

Микросервисы

на платформе .NET

Кристиан Хорсдал



Microservices in .NET Core

CHRISTIAN HORSDAL GAMMELGAARD



MANNING
Shelter Island

Кристиан Хорсдал

Микросервисы на платформе .NET



Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2018

Кристиан Хорсдал

Микросервисы на платформе .NET

Серия «Для профессионалов»

Перевел с английского *Е. Зазноба*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>О. Сивченко</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Рощина</i>
Художественный редактор	<i>С. Заматевская</i>
Корректоры	<i>О. Андриевич, Е. Павлович</i>
Верстка	<i>Г. Блинов</i>

ББК 32.988.02-018

УДК 004.738.5

Хорсдал К.

X82 Микросервисы на платформе .NET. — СПб.: Питер, 2018. — 352 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-496-03221-6

Создание микросервисов — удобная и надежная парадигма для программирования легких отказоустойчивых приложений. В этой книге подробно и интересно рассмотрены тонкости построения микросервисов на платформе .NET с применением таких популярных технологий, как Nancy и OWIN. Книга учитывает тонкости работы на платформе .NET Core и будет интересна всем, кому требуется эффективно и быстро решать нетривиальные задачи при работе с растущими системами.

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

ISBN 978-1617293375 англ.
ISBN 978-5-496-03221-6

© 2017 by Manning Publications Co. All rights reserved
© Перевод на русский язык ООО Издательство «Питер», 2018
© Издание на русском языке, оформление ООО Издательство «Питер», 2018
© Серия «Для профессионалов», 2018

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

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

Изготовлено в России. Изготовитель: ООО «Питер Пресс».

Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург, улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 12.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —

Книги печатные профессиональные, технические и научные.

Подписано в печать 07.12.17. Формат 70×100/16. Бумага офсетная. Усл. н. л. 28,380. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Краткое содержание

Предисловие	12
Благодарности	13
Об этой книге	14

Часть I. Знакомство с микросервисами

Глава 1. Первый взгляд на микросервисы	21
Глава 2. Простой микросервис для корзины заказов	52

Часть II. Создание микросервисов

Глава 3. Распознавание микросервисов и определение их области действия	81
Глава 4. Взаимодействие микросервисов	107
Глава 5. Хранение данных и их принадлежность	138
Глава 6. Проектирование устойчивых к ошибкам систем	164
Глава 7. Написание тестов для микросервисов	186

Часть III. Сквозная функциональность: создание платформы многоразового кода для микросервисов

Глава 8. Знакомство с OWIN: написание и тестирование промежуточного ПО OWIN... ..	217
Глава 9. Сквозная функциональность: мониторинг и журналирование	232
Глава 10. Обеспечение безопасности взаимодействия микросервисов	258
Глава 11. Создание платформы многоразового кода для микросервисов	284

Часть IV. Создание приложений

Глава 12. Создание приложений на основе микросервисов	309
--	-----

Приложения

Приложение А. Настройка среды разработки.....	340
Приложение Б. Развертывание для эксплуатации в производственной среде	345
Дополнительная литература.....	350
Список использованных в книге технологий	352

Оглавление

Предисловие.....	12
Благодарности	13
Об этой книге	14
Для кого предназначена книга	14
Структура издания.....	15
Условные обозначения и загрузка кода	16
Об авторе.....	17
Об иллюстрации на обложке	17

Часть I. Знакомство с микросервисами

Глава 1. Первый взгляд на микросервисы	21
1.1. Что такое микросервис	21
Какова архитектура микросервисов.....	23
Отличительные признаки микросервисов.....	23
1.2. Почему именно микросервисы?.....	29
Возможность непрерывной доставки ПО	29
Высокий уровень удобства сопровождения	31
Надежность и масштабируемость.....	32
1.3. Издержки и недостатки микросервисов.....	32
1.4. С нуля или на базе существующей системы?.....	33
1.5. Многократное использование кода	34
1.6. Обслуживание запросов пользователя: пример совместной работы микросервисов	35
Основная обработка пользовательского запроса.....	36
Побочные действия пользовательских запросов.....	37
Общая картина	38
1.7. стек технологий платформы .NET для микросервисов.....	39
Фреймворк Nancy.....	39
Стандарт OWIN	40
Настройка среды разработки	43
1.8. Пример простого микросервиса	44
Создание пустого приложения ASP.NET Core	45
Добавление в проект фреймворка Nancy.....	45

Добавление модуля Nancy с реализацией конечной точки	46
Добавляем промежуточное ПО OWIN.....	48
1.9. Резюме	50
Глава 2. Простой микросервис для корзины заказов	52
2.1. Обзор микросервиса Shopping Cart.....	53
2.2. Реализация микросервиса Shopping Cart	56
Создание пустого проекта	56
API, предоставляемый микросервисом Shopping Cart другим сервисам	57
Извлечение информации о товаре	65
Синтаксический разбор ответа с информацией о товаре	68
Добавление стратегии обработки ошибок	70
Реализация простейшей ленты событий.....	72
2.3. Выполнение кода	77
2.4. Резюме	77

Часть II. Создание микросервисов

Глава 3. Распознавание микросервисов и определение их области действия	81
3.1. Главный фактор определения области действия микросервисов: бизнес-возможности	82
Что такое бизнес-возможность.....	82
Выделение бизнес-возможностей	83
Пример: POS-система	84
3.2. Дополнительный фактор определения области действия микросервисов: вспомогательные технические возможности	91
Что такое вспомогательная техническая возможность	91
Примеры вспомогательных технических возможностей	91
Распознавание технических возможностей.....	96
3.3. Что делать, если сложно очертить область действия	97
Для начала чуть завышаем размер	97
Выделение новых микросервисов из существующих.....	101
Заранее планируем выделение нового микросервиса.....	102
3.4. Микросервисы с корректно заданной областью действия обладают типичным набором свойств.....	103
Сведение основных областей действия к бизнес-возможностям повышает качество микросервисов	103
Сведение дополнительных областей действия к вспомогательным техническим возможностям повышает качество микросервисов	104
3.5. Резюме	105
Глава 4. Взаимодействие микросервисов	107
4.1. Типы взаимодействия: команды, запросы и события.....	107
Команды и запросы: синхронное взаимодействие	109
События: асинхронное взаимодействие.....	113
Форматы данных.....	116

4.2. Реализация взаимодействия	117
Создание проекта для микросервиса Loyalty Program	119
Реализация команд и запросов	120
Реализация команд с помощью методов POST и PUT протокола HTTP	120
Реализация запросов с помощью метода GET протокола HTTP	124
Форматы данных	125
Реализация взаимодействия на основе событий	128
4.3. Резюме	137
Глава 5. Хранение данных и их принадлежность	138
5.1. У любого микросервиса имеется хранилище данных	138
5.2. Распределение данных по микросервисам	139
Правило 1: бизнес-возможности обуславливают принадлежность данных	139
Правило 2: репликация данных для ускорения работы системы и повышения ее надежности	141
Где микросервисы хранят свои данные	145
5.3. Реализация хранилища данных микросервиса	147
Хранение принадлежащих микросервису данных	148
Хранение порождаемых микросервисом событий	151
Установка заголовков кэширования в ответах Nancy	159
Чтение и использование заголовков кэширования	161
5.4. Резюме	163
Глава 6. Проектирование устойчивых к ошибкам систем	164
6.1. Ожидайте отказов	165
Наличие подробных журналов	166
Использование маркеров корреляции	168
Развертывание и свертывание	168
Не допускайте распространения сбоев	169
6.2. Ответственность за устойчивость к ошибкам на стороне клиента	171
Шаблон устойчивости к ошибкам «Повтор»	172
Шаблон устойчивости к ошибкам «Предохранитель»	174
6.3. Реализация шаблонов устойчивости к ошибкам	176
Реализация стратегии частых повторов с помощью библиотеки Polly	179
Реализация предохранителя с помощью библиотеки Polly	180
Реализация стратегии редких повторов с помощью библиотеки Polly	181
Заносим в журнал все необработанные исключения	184
6.4. Резюме	185
Глава 7. Написание тестов для микросервисов	186
7.1. Что и как тестировать	186
Пирамида тестов: что необходимо тестировать в системе микросервисов	187
Системные тесты: тестирование системы микросервисов в целом	188

Эксплуатационные тесты: тестирование микросервиса извне процесса.....	189
Модульные тесты: тестирование конечных точек изнутри процесса.....	192
7.2. Тестировочные библиотеки Nancy.Testing и xUnit	194
Познакомьтесь: библиотека Nancy.Testing	194
Познакомьтесь: фреймворк xUnit	195
Совместная работа xUnit и Nancy.Testing	195
7.3. Написание модульных тестов с помощью библиотеки Nancy.Testing	197
Создание проекта для модульного тестирования	198
Использование объекта Browser для модульного тестирования конечных точек.....	200
Использование настраиваемого загрузчика для внедрения имитаций в конечные точки.....	202
7.4. Написание эксплуатационных тестов	206
Создание проекта для эксплуатационного теста.....	207
Создание имитационных конечных точек	208
Запуск всех процессов тестируемого микросервиса.....	210
Выполнение тестового сценария для тестируемого микросервиса.....	212
7.5. Резюме.....	213

Часть III. Сквозная функциональность: создание платформы многоразового кода для микросервисов

Глава 8. Знакомство с OWIN: написание и тестирование промежуточного ПО OWIN ...	217
8.1. Реализация сквозной функциональности	217
8.2. Конвейер OWIN	220
8.3. Написание промежуточного ПО	225
Промежуточное ПО в виде лямбда-выражений	226
Классы промежуточного ПО	227
8.4. Тестирование промежуточного ПО и конвейеров	228
8.5. Резюме	231
Глава 9. Сквозная функциональность: мониторинг и журналирование	232
9.1. Мониторинг микросервисов	232
9.2. Журналирование микросервисов.....	236
9.3. Реализация промежуточного ПО для мониторинга	240
Реализация конечной точки для поверхностного мониторинга	241
Реализация конечной точки для углубленного мониторинга	242
Добавление промежуточного ПО для мониторинга в конвейер OWIN.....	244
9.4. Реализация промежуточного ПО для журналирования	246
Добавление маркеров корреляции во все журнальные сообщения.....	248
Добавление маркера корреляции во все исходящие HTTP-запросы	250
Журналирование запросов и показателей их производительности.....	253
Конфигурация конвейера OWIN с промежуточным ПО для маркеров корреляции и журналирования	255
9.5. Резюме	257

Глава 10. Обеспечение безопасности взаимодействия микросервисов	258
10.1. Безопасность микросервисов	258
Аутентификация пользователей на периферии	260
Авторизация пользователей в микросервисах	262
Насколько микросервисы должны доверять друг другу	262
10.2. Реализация безопасного обмена сообщениями между микросервисами.....	265
Познакомьтесь: IdentityServer	267
Реализация аутентификации с помощью промежуточного ПО IdentityServer..	273
Реализация авторизации при взаимодействии между микросервисами с помощью IdentityServer и промежуточного ПО.....	275
Реализация авторизации пользователей в модулях Nancy.....	278
10.3. Резюме.....	283
Глава 11. Создание платформы многоразового кода для микросервисов	284
11.1. Создание нового микросервиса должно быть быстрым и легким	284
11.2. Создание платформы многоразового кода для микросервисов	285
11.3. Упаковка и совместное использование промежуточного ПО с помощью NuGet	287
Создание пакета с промежуточным ПО для журналирования и мониторинга.	288
Создание пакета с промежуточным ПО для авторизации.....	295
Создание пакета с REST-фабрикой клиентов	298
Автоматическая регистрация фабрики HttpClientFactory в контейнере Nancy.....	301
Использование платформы микросервисов	303
11.4. Резюме.....	306

Часть IV. Создание приложений

Глава 12. Создание приложений на основе микросервисов	309
12.1. Приложения для систем микросервисов: одно или несколько приложений?	309
Универсальные приложения.....	310
Специализированные приложения	311
12.2. Шаблоны для построения приложений на основе микросервисов	312
Составные приложения: интеграция на фронтенде.....	312
Шлюз API.....	315
Бэкенд для фронтенда	317
Когда использовать каждый из шаблонов	319
Отрисовка на стороне клиента или на стороне сервера?	320
12.3. Пример: корзина товаров и список продуктов	321
Создание шлюза API	324
Создаем интерфейс списка продуктов.....	326
Создание интерфейса корзины.....	330
Позволяем пользователям добавлять продукты в корзину	334
Позволяем пользователям удалять продукты из корзины.....	336
12.4. Резюме.....	337

Приложения

Приложение А. Настройка среды разработки.....	340
А.1. Установка IDE.....	340
Visual Studio 2015.....	341
Visual Studio Code.....	341
Редактор ATOM	342
IDE JetBrains Rider.....	342
А.2. Установка интерфейса командной строки dotnet	342
А.3. Установка генератора Yeoman ASP.NET	343
А.4. Установка Postman.....	343
А.5. Установка SQL Server Express.....	344
Приложение Б. Развертывание для эксплуатации в производственной среде	345
В.1. Развертывание API для протокола HTTP	345
Для серверов с операционной системой Windows.....	346
Для серверов с операционной системой Linux.....	346
Azure Web Apps	347
Azure Service Fabric.....	347
В.2. Развертывание потребителей событий.....	347
Для серверов с операционной системой Windows.....	347
Для серверов с операционной системой Linux.....	348
Azure WebJobs.....	348
Функции Azure	348
Amazon Lambda.....	349
Дополнительная литература.....	350
Микросервисы	350
Проектирование программного обеспечения и архитектуры в целом.....	350
Список использованных в книге технологий	352

Предисловие

Когда в издательстве «Мэннинг» мне предложили написать книгу, речь шла о книге, посвященной фреймворку Nancy. С одной стороны, я был рад снова рассказать о Nancy, поскольку это удивительный фреймворк, но я уже написал о нем одну книгу, поэтому мне хотелось, чтобы новое издание было чем-то большим. Я был убежден, что Nancy заслуживает не просто описания и демонстрации, но демонстрации в таком контексте, который позволил бы показать, *почему* Nancy такой удачный фреймворк. Для меня важна простота работы с ним. Nancy дает вам возможность создавать то, что хотите. В то же время это фреймворк с огромным потенциалом, развивающийся по мере роста потребностей пользователей. После некоторых размышлений и обсуждения с издательством «Мэннинг» стало понятно, что необходимый мне контекст для демонстрации Nancy — *микросервисы*. Микросервисы позволяют быстро создавать «легкие» приложения — как раз это я со временем научился так ценить. Они также подчеркивают необходимость в «легковесных», но вместе с тем обладающих широкими возможностями технологиях, таких как Nancy. На этой стадии все мои идеи о том, какой должна быть книга, начали становиться на свои места: я хотел написать книгу, посвященную скорее проектированию и реализации микросервисов, а не какой-то конкретной технологии, но в то же время демонстрирующую с наилучшей стороны «легковесные» технологии платформы .NET. В итоге получилась книга, которую вы сейчас читаете, и я надеюсь, что вы не только узнаете из нее, как преуспеть в разработке микросервисов, но и осознаете важность тщательного выбора тех библиотек и фреймворков, которые дорожат простотой, не мешают вам и с которыми просто приятно работать.

Что такое микросервис? Микросервис — узконаправленный сервис со следующими свойствами:

- ☐ отвечает за отдельный фрагмент функциональности;
- ☐ его можно установить отдельно;
- ☐ состоит из одного или нескольких процессов;
- ☐ имеет собственное хранилище данных;
- ☐ небольшая команда разработчиков может поддерживать несколько микросервисов;
- ☐ микросервисы допускают замену.

Эти свойства можно использовать в качестве инструкции по проектированию и реализации микросервисов, а также чтобы распознать микросервисы, столкнувшись с ними.

Благодарности

Написание книги требует времени — много времени. Так что первые слова благодарности — моей жене Джейн Хорсдэл Гаммельгард (Jane Horsdal Gammelgaard), поддерживавшей меня все это время. Ты потрясающая, Джейн!

Я хотел бы поблагодарить моего редактора Дэна Магарри (Dan Maharry). Благодаря его замечательным советам, мягкому, а иногда и жесткому подталкиванию в нужную сторону и постоянному акцентированию внимания на создании высококачественного произведения моя книга получилась намного лучше, чем могла бы быть без него. Огромная благодарность также техническому редактору Микаэлю Лунду (Michael Lund) за внимательный анализ кода и предложения по его улучшению, а также за критику моих ценочек рассуждений в тех случаях, когда они были мало-вразумительны. Отдельная благодарность Карстену Стребекку (Karsten Strøbæk) за тщательную вычитку текста.

Моя благодарность также замечательной группе технических экспертов-рецензентов: Энди Киру (Andy Kirsch), Брайану Расмуссену (Brian Rasmussen), Джемре Менгу (Cemre Mengu), Гаю Мэттью Лакроссу (Guy Matthew LaCrosse), Джеймсу Макгинну (James McGinn), Джеффу Смиту (Jeff Smith), Джиму Макгинну (Jim McGinn), Мэтту Р. Коулу (Matt R. Cole), Мортену Херману Лангъяру (Morten Herman Langkjær), Нестору Нарваэсу (Nestor Narvaez), Нику Макгиннессу (Nick McGinness), Ронни Хегелунду (Ronnie Hegelund), Самуэлю Бошу (Samuel Bosch) и Шахиду Игбалу (Shahid Iqbal). Они предлагали мне различные темы и другие способы изложения существующих тем, находили ошибки и опечатки в коде и терминологии. Каждый этап процесса рецензирования стал этапом обретения этой книгой ее нынешних очертаний.

Наконец, я хотел бы сказать спасибо сотрудникам издательства «Мэннинг», благодаря которым книга появилась на свет: издателю Маржан Бэйс (Marjian Bace), редактору Грегу Вайлду (Greg Wild), а также всем членам редакционно-издательской команды, включая Тиффани Тэйлор (Tiffany Taylor), Кэти Теннант (Katie Tennant), Мелоди Долаб (Melody Dolab) и Гордана Салиновича (Gordan Salinovic).

Об этой книге

«Микросервисы на платформе .NET» — практическое руководство по написанию микросервисов на основе платформы .NET с помощью простых в использовании облегченных технологий, таких как веб-фреймворк Nancy и промежуточное ПО, созданное по стандарту OWIN (Open Web Interface for .NET — открытый веб-интерфейс для платформы .NET). Я приложил максимум усилий, чтобы представить материал книги в таком виде, который позволил бы вам сразу же начать применять прочитанное на практике. Поэтому я старался везде пояснять, почему делаю что-то именно так, равно как и показывать в подробностях, как именно.

Веб-фреймворк Nancy, используемый на протяжении всей книги, создан Андреасом Хоканссоном (Andreas Håkansson), который по сей день руководит проектом. Через какое-то время к Андреасу присоединился Стивен Роббинс (Steven Robbins), и вдвоем они сделали из Nancy потрясающий фреймворк. В настоящее время его продвигают Андреас, Стивен и их помощники Кристиан Хелланг (Kristian Hellang), Джонатан Шеннон (Jonathan Channon), Дэмиен Хики (Damian Hickey), Филипп Хэйдон (Phillip Haydon) и ваш покорный слуга, а также многочисленное сообщество других разработчиков. Полный список участвующих в разработке и поддержке Nancy можно найти по адресу <http://nancyfx.org/contribs.html>.

OWIN — открытый стандарт для интерфейсов между веб-серверами и веб-приложениями. Работа над OWIN началась в конце 2010 года усилиями Райана Райли (Ryan Riley), Бенджамина ван дер Веена (Benjamin van der Veen), Маурисио Шеффера (Mauricio Scheffer) и Скотта Куна (Scott Koon). С тех пор множество специалистов внесли свой вклад в разработку спецификаций стандарта OWIN — сначала через группу Google, а потом с помощью GitHub-репозитория OWIN (<https://github.com/owin/owin>) — и в его реализацию.

Для кого предназначена книга

«Микросервисы на платформе .NET» — книга прежде всего для разработчиков, но дизайнеры и другие специалисты тоже могут многое почерпнуть из нее. При ее написании я ориентировался на .NET-разработчиков, которые хотят начать создавать распределенные серверные системы вообще и микросервисы в частности, поэтому основное внимание уделено тому, что необходимо знать и делать разработчику для написания кода для системы микросервисов. Предполагается, что читатели обладают практическими знаниями в области языка программирования C# и — хотя бы в небольшом объеме — протокола HTTP.

Структура издания

Книга состоит из 12 глав, разделенных на четыре части.

Часть I кратко знакомит вас с микросервисами, объясняя, что они собой представляют и чем интересны. В этой части речь идет также об основных используемых в книге технологиях — фреймворке Nancy и стандарте OWIN.

- ❑ Глава 1 рассказывает, что такое микросервисы и почему они важны. В ней перечислены шесть отличительных признаков микросервисов, которыми я руководствуюсь при их проектировании и реализации. В конце главы мы познакомимся с фреймворком Nancy и стандартом OWIN.
- ❑ Глава 2 — комплексный пример написания микросервиса с помощью фреймворка Nancy и стандарта OWIN, а также библиотеки Polly и фреймворка .NET Core. К концу этой главы мы создадим законченный, хотя и простой микросервис.

Часть II охватывает вопросы разбиения системы на микросервисы и реализации функциональности в системе микросервисов.

- ❑ В главе 3 изучается вопрос о том, как распознать микросервисы и определить, какую функциональность поместить в каждый из них. Эта глава посвящена проектированию системы микросервисов в целом.
- ❑ Глава 4 демонстрирует проектирование и реализацию совместной работы микросервисов. В этой главе обсуждаются различные способы взаимодействия микросервисов между собой и показано, как это взаимодействие реализовать.
- ❑ В главе 5 изучается вопрос о том, где в системе микросервисов должны храниться данные, а также рассматривается возможность дублирования части данных между несколькими микросервисами.
- ❑ Глава 6 объясняет и демонстрирует реализацию некоторых важных методов обеспечения надежности систем микросервисов.
- ❑ В главе 7 подробно рассматривается тестирование системы микросервисов, включая тестирование системы в целом, тестирование каждого микросервиса и тестирование кода внутри микросервисов.

В части III показано, как ускорить разработку новых микросервисов путем построения единой платформы микросервисов, рассчитанной специально для конкретной системы. Подобная платформа обеспечивает реализацию грунны важных элементов функциональности, пронизывающих всю систему микросервисов, таких как журналирование, мониторинг и безопасность. В этой части книги мы создадим такую платформу и посмотрим, как применять ее для быстрого создания микросервисов.

- ❑ Глава 8 детально познакомит вас со стандартом OWIN. В ней вы прочтете о методах построения промежуточного ПО OWIN и узнаете, насколько хорошо промежуточное ПО OWIN подходит для различной сквозной функциональности.
- ❑ Глава 9 разъясняет, почему в системе микросервисов так важны мониторинг и журналирование. На базе полученных в главе 8 знаний о стандарте OWIN мы

создадим промежуточное ПО для поддержки мониторинга и журналирования микросервисов.

- ❑ В главе 10 обсуждается безопасность в системе микросервисов. Сильно распределенная природа системы микросервисов ставит перед нами связанные с безопасностью задачи, которые мы здесь обсудим. Мы также поговорим о возможностях использования промежуточного ПО OWIN в целях реализации различных средств безопасности в микросервисах.
- ❑ Глава 11 описывает создание на основе материала глав 9 и 10 платформы микросервисов. Эта платформа создается путем унаковки взятого из предыдущих глав промежуточного ПО OWIN. Такие пакеты NuGet уже готовы для совместного использования микросервисами. Глава включает также пример создания нового микросервиса с помощью этой платформы.

Часть IV состоит из главы 12, завершающей книгу описанием некоторых подходов к созданию приложений для конечных пользователей системы микросервисов. В ней также рассматривается небольшое приложение на базе некоторых микросервисов из предыдущих глав.

В целом эти 12 глав научат вас проектировать и программировать микросервисы с помощью несложных технологий на основе платформы .NET.

Условные обозначения и загрузка кода

В большинстве глав приводятся примеры программ. Их все можно найти в разделе загрузок для данной книги на сайте издательства «Мэннинг» по адресу <https://www.manning.com/books/microservices-in-net-core> или в репозитории Git на GitHub по адресу <https://github.com/horsdal/microservices-in-dotnetcore>.

Код основан на фреймворке .NET Core, так что для запуска вам понадобится установить .NET Core, а также соответствующие утилиты командной строки и интегрированную среду разработки (IDE). Информацию по их установке и настройке можно найти в приложении А.

В этой книге я также использую несколько сторонних библиотек с открытым исходным кодом, в частности веб-фреймворк Nancy. Платформа .NET Core сильно отличается от традиционной .NET, так что возникает необходимость в переносе и тщательном тестировании существующих библиотек, прежде чем можно будет говорить о полной поддержке ими .NET Core. Если на момент выхода книги появятся более стабильные выпуски различных библиотек для платформы .NET Core, я рекомендую применять их во время работы с примерами кода.

В репозитории GitHub, расположенном по адресу <https://github.com/horsdal/microservices-in-dotnetcore>, ветка `master` содержит код в том виде, в котором он приведен в книге. При выходе стабильных версий библиотек для платформы .NET Core я планирую создать ветку `current` и хранить в ней копию кода, который будет поддерживаться в соответствующем текущим версиям библиотек виде на протяжении нескольких лет после выхода издания.

В книге вы найдете множество примеров исходного кода, как в пронумерованных листингах, так и внутри обычного текста. Исходный код листингов показан:

вот таким шрифтом отдельными строками,

а код в тексте — моноширинным шрифтом, чтобы он выделялся среди обычного текста. Иногда код будет набран **жирным** шрифтом, чтобы показать изменения по сравнению с предыдущими этапами, например, как в случае добавления нового элемента в существующую строку кода.

Во многих случаях форматирование исходного кода было изменено: я добавил разрывы строк и поменял отступы, чтобы код влез в ширину полосы нечатной книги. В редких случаях, когда даже этого оказывалось недостаточно, листинги включают символы продолжения строки (↵). Кроме того, я часто удалял из листингов комментарии, если код описывается в тексте. Многие листинги сопровождаются пояснениями к коду, подчеркивающими важные моменты.

Об авторе

Кристиан — независимый консультант с многолетним опытом в сфере создания веб-приложений и распределенных систем как на .NET, так и на других платформах. Он состоит в команде поддержки фреймворка Napsu, а также является ключевым сотрудником корпорации Microsoft в отделе поддержки платформы .NET.

Об иллюстрации на обложке

Рисунок на обложке книги называется «Император Китая в парадном облачении, 1700 год». Иллюстрация взята из четырехтомной книги Томаса Джеффериса (Thomas Jefferys) *Collection of the Dresses of Different Nations, Ancient and Modern* («Сборник платьев различных народов, старинных и современных»), изданной в Лондоне между 1757 и 1772 годом. Титульная страница сообщает, что это раскрашенные вручную гравюры на меди, с усилением цветности гуммиарабиком.

Томаса Джеффериса (1719–1771) называли Географом Его Величества Георга III. Он был английским картографом, гравировал и неچатал карты для нравительства и различных официальных органов, а также издавал различные коммерческие карты и атласы, большинство из которых — Северной Америки. Его работа пробудила интерес к особенностям одежды стран, которые он исследовал и картографировал. Эти особенности блестяще отражены в упомянутом сборнике.

Путешествия ради удовольствия были относительно новым явлением в то время, и сборники, подобные книге Томаса Джеффериса, были очень популярны, знакомя и настоящих, и «диванных» туристов как с другими регионами мира, так и с местными костюмами того времени. Разнообразие рисунков в книгах Джеффериса отражает уникальность и индивидуальность городов и регионов стран 200-летней давности. Изолированные друг от друга, люди говорили на разных диалектах

и языках. На улицах или в сельской местности было несложно определить, где человек жил и чем занимался, каков был его социальный статус, просто по тому, как он говорил или какую одежду носил. С тех времен дресс-код изменился, и нет такого регионального разнообразия, столь богатого в тот период. Сейчас зачастую непросто даже определить, с какого континента человек. Наверное, если посмотреть на это с оптимизмом, мы променяли культурное и визуальное многообразие на более разнообразную личную жизнь — или, возможно, более разнообразную и интересную интеллектуальную и техническую жизнь.

В наши времена, когда непросто отличить одну книгу по компьютерам от другой, издательство «Мэннинг» прославляет изобретательность и инициативу компьютерного дела обложками книг, отражающими богатое разнообразие региональной жизни два столетия назад, возвращенное к жизни иллюстрациями Джеффериса.

Часть I

Знакомство с микросервисами

Из этой части вы узнаете, что такое микросервисы и почему они вам нужны. Я начну с обсуждения шести отличительных признаков, которые можно использовать для распознавания микросервисов и в качестве руководящих принципов их проектирования. По ходу дела мы рассмотрим выгоды и издержки применения микросервисов.

Ближе к концу главы 1 мы бегло рассмотрим стек технологий, используемый на протяжении всей книги. Этот стек состоит из фреймворка .NET Core, веб-фреймворка Nancy и стандарта OWIN. В главе 2 вы попробуете создать собственный микросервис. Вы также узнаете о дополнительных возможностях фреймворка Nancy.

1

Первый взгляд на микросервисы

В этой главе:

- ❑ понятие микросервиса и его основные отличительные признаки;
- ❑ плюсы и минусы микросервисов;
- ❑ пример совместной работы микросервисов при обслуживании запроса пользователя;
- ❑ использование веб-фреймворка Nancy в простом приложении.

В этой главе я расскажу вам о микросервисах и продемонстрирую, чем они интересны. Мы также рассмотрим шесть отличительных признаков микросервиса. Наконец, я познакомлю вас с двумя важнейшими из используемых в этой книге технологий: с основанным на платформе .NET веб-фреймворком Nancy и конвейером промежуточного ПО OWIN.

1.1. Что такое микросервис

Микросервис (microservice) — сервис, предоставляемый удаленным API для использования остальной системой, обладающий одной и только одной чрезвычайно узконаправленной функциональной возможностью. Например, рассмотрим систему управления складом. Если разбить ее возможности на составные части, получится следующий список.

1. Принимаем поступающий на склад товар.
2. Определяем, где должен храниться новый товар.
3. Рассчитываем маршруты размещения внутри склада, чтобы поместить товар в нужные хранилища.
4. Распределяем маршруты размещения товара между работниками склада.
5. Получаем заказы.
6. Рассчитываем маршруты изъятия товара со склада для составления заказов.
7. Распределяем маршруты изъятия товара между работниками склада.

Рассмотрим, как реализовать первую из этих возможностей — приемку прибывшего на склад товара — в качестве микросервиса. Мы назовем этот микросервис *Receive Stock* («Принять товар»).

1. По HTTP поступает запрос на приемку нового товара и занесение его в журнал. Он может исходить от другого микросервиса или, возможно, от веб-страницы, используемой диспетчером для регистрации поступившего товара. Микросервис *Receive Stock* должен зарегистрировать новый товар в собственном хранилище данных.
2. Микросервис отправляет ответ в качестве подтверждения приемки товара.

Рисунок 1.1 демонстрирует получение микросервисом *Receive Stock* запроса от другого микросервиса, работающего совместно с ним.



Рис. 1.1. Микросервис *Receive Stock* при поступлении нового товара предоставляет для использования API, к которому могут обращаться другие микросервисы

Каждая маленькая возможность в системе реализуется в виде отдельного микросервиса. Любой микросервис в системе:

- ☐ работает в собственном отдельном процессе;
- ☐ может быть развернут отдельно, независимо от других микросервисов;
- ☐ имеет собственное хранилище данных;
- ☐ взаимодействует с другими микросервисами для выполнения своих заданий.

Важно также отметить, что микросервисы могут взаимодействовать с другими микросервисами, не обязательно написанными на том же языке программирования (C#, Java, Erlang и т. д.). Все, что им требуется знать, — как обмениваться сообщениями друг с другом. Некоторые могут обмениваться сообщениями через служебную шину или двоичный протокол, например Thrift, в зависимости от системных требований, но гораздо чаще микросервисы обмениваются сообщениями по протоколу HTTP.

ПРИМЕЧАНИЕ

Эта книга посвящена созданию микросервисов на платформе .NET с помощью языка программирования C# и веб-фреймворка Nancy. Микросервисы, которые я буду демонстрировать, представляют собой маленькие узкоспециализированные приложения Nancy, взаимодействующие по HTTP.

Какова архитектура микросервисов

Основное внимание в этой книге уделяется проектированию и реализации отдельных микросервисов, но стоит отметить, что термин «*микросервисы*» можно использовать также для описания *архитектурного стиля* всей системы, состоящей из множества микросервисов. Микросервисы как архитектурный стиль — упрощенный вариант сервис-ориентированной архитектуры (service-oriented architecture (SOA)), в которой сервисы ориентированы на выполнение одного действия каждый, и выполнения качественного. Система с архитектурой микросервисов представляет собой распределенную систему с, вероятно, значительным количеством совместно работающих микросервисов.

Микросервисный архитектурный стиль быстро обретает все большую популярность при создании и поддержке сложных серверных комплексов программ. Причина этого очевидна: микросервисы обеспечивают множество потенциальных преимуществ по сравнению как с более традиционными сервис-ориентированными подходами, так и с монолитными архитектурами. Микросервисы при надлежащей реализации демонстрируют пластичность, масштабируемость и отказоустойчивость вдобавок к небольшой длительности производственного цикла от начала создания до внедрения в производство.

Отличительные признаки микросервисов

Как я упоминал, микросервис — это *сервис с одной узконаправленной функциональной возможностью*. Что эта фраза значит? Поскольку методология микросервисов все еще находится в стадии развития (по состоянию на начало 2016 года), пока не существует общепринятого определения того, что именно представляет собой микросервис¹. Можно, однако, посмотреть, что отличает микросервис в целом. Я обнаружил, что существует шесть основных отличительных признаков микросервиса.

1. Микросервис отвечает за одну функциональную возможность.
2. Микросервисы можно развертывать по отдельности.
3. Микросервис состоит из одного или нескольких процессов.
4. У микросервиса имеется собственное хранилище данных.
5. Небольшая команда разработчиков может сопровождать несколько микросервисов.
6. Микросервис можно легко заменить.

Этот список отличительных признаков поможет вам при столкновении с правильно настроенным микросервисом распознать его, а также определить сферу действия и реализации своих собственных микросервисов. Приняв за основу эти

¹ Для дальнейшего обсуждения отличительных признаков микросервисов я рекомендовал бы вам обратиться к статье на эту тему: *Fowler M., Lewis J. Microservices: A Definition of This New Architectural Term*, 2014. — March 25 [Электронный ресурс]. — Режим доступа: <http://martinfowler.com/articles/microservices.html>.

отличительные признаки, вы вступите на нуть извлечения максимальной выгоды из своих микросервисов и создания в результате *пластичной, масштабируемой и отказоустойчивой* системы. На протяжении всей книги я буду демонстрировать, что эти отличительные признаки означают в смысле устройства микросервисов и как писать необходимый для их вонложения код. Тенерь рассмотрим каждый из признаков по отдельности.

Ответственность за одну возможность

Микросервис *отвечает за одну и только одну функциональную возможность* из всей системы. Разделим это утверждение на две составные части.

- ❑ На микросервис возлагается ровно одна обязанность.
- ❑ Эта обязанность состоит в реализации отдельной возможности.

Принцип единственной обязанности формулируется несколькими различными способами. Одна из традиционных его формулировок: «У класса должна быть только одна причина для изменения»¹. Хотя здесь говорится именно о *классе*, этот принцип может применяться и вне контекста классов в объектно-ориентированных языках программирования. В случае микросервисов принцип единственной обязанности применяется на уровне сервисов.

Более современная формулировка принципа единственной обязанности тоже обязана своим появлением дядюшке Бобу: «Собирайте воедино вещи, меняющиеся по схожим причинам. Отделяйте друг от друга вещи, меняющиеся по различным причинам»². Эта формулировка данного принципа применима к микросервисам: микросервис должен реализовывать ровно одну функциональную возможность. Таким образом, микросервис должен изменяться только при изменении этой возможности. Более того, вы должны постараться реализовать в микросервисе эту возможность максимально полно, чтобы при ее изменении нужно было менять только один микросервис.

Существует два вида возможностей в системе микросервисов.

- ❑ *Бизнес-возможность* — выполняемое системой действие, которое способствует реализации предназначения системы. Например, это может быть отслеживание корзины заказов пользователей или расчет цен. Предметно-ориентированное проектирование — отличный способ разделения системы на отдельные бизнес-возможности.
- ❑ *Техническая возможность* — нечто необходимое для нескольких других микросервисов, например для интеграции с какой-то сторонней системой. Технические возможности нельзя назвать главными движущими силами при разбиении системы на микросервисы, они выделяются лишь в случае, когда оказывается, что нескольким микросервисам для их бизнес-возможностей требуется одна и та же техническая возможность.

¹ Martin R. C. SRP: The Single Responsibility Principle [Электронный ресурс]. — Режим доступа: <http://mng.bz/zQyz>.

² Martin R. C. The Single Responsibility Principle [Электронный ресурс]. — Режим доступа: <http://mng.bz/RZgU>.

ПРИМЕЧАНИЕ

Определение области действия и обязанностей микросервисов рассматривается в главе 3.

Развертываемость по отдельности

Микросервисы должны допускать *развертываемость по отдельности*. При изменении микросервиса должна существовать возможность ввести этот измененный микросервис в эксплуатацию, не развертывая (вообще не затрагивая) какие-либо другие части системы. Остальные микросервисы в системе должны выполняться и работать во время развертывания измененного микросервиса и продолжать выполняться после того, как новая версия будет развернута.

Рассмотрим сайт интернет-магазина. При изменении микросервиса Shopping Cart («Корзина заказов») должна быть обеспечена возможность развертывания только этого микросервиса (рис. 1.2).

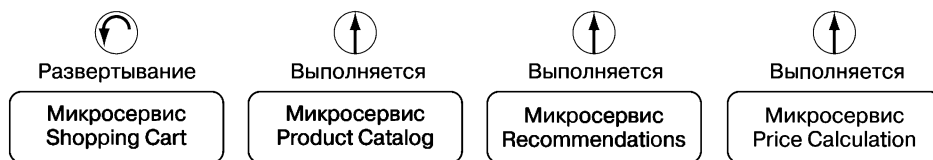


Рис. 1.2. Во время развертывания микросервиса Shopping Cart остальные микросервисы продолжают работать

Тем временем микросервисы Price Calculation («Расчет цены»), Recommendations («Рекомендации»), Product Catalog («Каталог товаров») и другие продолжают работать и обслуживать запросы пользователей.

Возможность индивидуального развертывания каждого из микросервисов важна, поскольку в системе их может быть очень много и каждый может взаимодействовать с несколькими другими. Одновременно с этим выполняется развертывание некоторых или всех микросервисов. Если бы пришлось развертывать все микросервисы или их группы в строгом порядке, управление развертываниями быстро стало бы громоздким, что привело бы к нечастым и объемным рискованным развертываниям. Несомненно, вместо этого желательно иметь возможность частого развертывания небольших изменений во всех микросервисах, что означает небольшие развертывания с низким уровнем риска.

Чтобы иметь возможность развертывать отдельный микросервис в тот момент, когда остальная система продолжает работать, процесс сборки должен быть настроен с учетом следующего.

- ☐ Все микросервисы должны быть встроены в отдельные артефакты или пакеты.
- ☐ Процесс развертывания должен быть настроен таким образом, чтобы обеспечить поддержку индивидуального развертывания при продолжении функционирования остальных микросервисов. Например, можно использовать процесс нлавающего развертывания, при котором микросервис развертывается на одном сервере за один раз, чтобы снизить время простоя.

То, что микросервисы желательно разворачивать по отдельности, влияет на способ их взаимодействия. Изменения в интерфейсе микросервиса обычно должны быть обратно совместимыми, чтобы другие микросервисы могли продолжать работать с новой версией точно так же, как работали со старой. Более того, способ взаимодействия микросервисов должен быть устойчивым к ошибкам в том смысле, что каждый микросервис должен быть готов к периодическим сбоям других микросервисов и должен при этом продолжать работать по мере возможности. Сбой одного микросервиса, например, из-за простоя во время разворачивания не должен приводить к сбою других микросервисов, а лишь к сокращению функциональности или чуть более длительному времени обработки.

ПРИМЕЧАНИЕ

Взаимодействие и устойчивость к ошибкам микросервисов мы рассмотрим в главах 4, 5 и 7.

Состоит из одного или нескольких процессов

Микросервис должен выполняться в отдельном процессе или отдельных процессах, чтобы оставаться как можно более независимым от других микросервисов той же системы, а также чтобы сохранять возможность отдельного разворачивания. Разбив это положение на составные части, получаем два пункта.

- ❑ Все микросервисы должны работать в отдельных от других микросервисов процессах.
- ❑ У любого микросервиса может быть более одного процесса.

Вернемся к микросервису Shopping Cart. Если он работает в том же процессе, что и микросервис Product Catalog (рис. 1.3), код микросервиса Shopping Cart может вызывать побочные эффекты в микросервисе Product Catalog. Это означает сильную (и нежелательную) связанность между микросервисами Shopping Cart и Product Catalog: работа одного может приводить к простоям или ошибкам другого.

Проблемы на границе процесса.

Микросервисы должны

выполняться в отдельных процессах

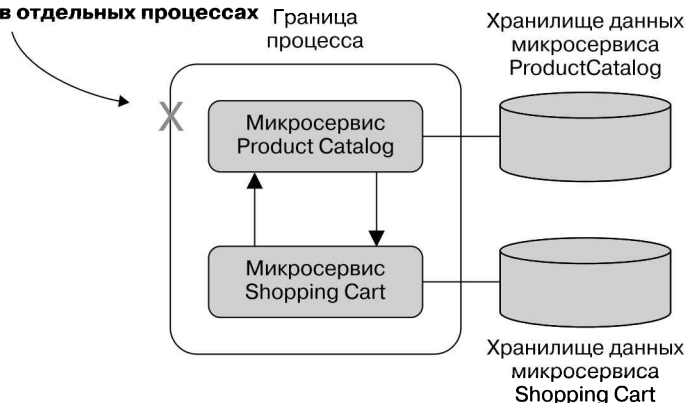


Рис. 1.3. Выполнение нескольких микросервисов в одном процессе ведет к сильной связанности

Теперь рассмотрим развертывание новой версии микросервиса Shopping Cart. Оно или повлечет за собой повторное развертывание микросервиса Product Catalog, или потребует какой-нибудь разновидности динамической загрузки кода, которая бы позволила заменить код микросервиса Shopping Cart в работающем процессе. Первый вариант напрямую противоречит положению об индивидуальной развертываемости микросервисов. Второй же сложен и как минимум ставит микросервис Product Catalog под угрозу сбоя вследствие развертывания микросервиса Shopping Cart.

Говоря о сложности: почему микросервис может состоять более чем из одного процесса? Мы ведь в конце концов стремимся к максимальному упрощению микросервисов.

Рассмотрим микросервис Recommendations. Он реализует и выполняет алгоритмы выдачи рекомендаций для сайта интернет-магазина, а также содержит базу данных, в которой хранятся необходимые для этого данные. Алгоритмы выполняются в одном процессе, а база данных работает в другом. Зачастую микросервису необходимо два или более процесса для реализации всего того (например, хранилища данных и фоновой обработки), что требуется ему для обеспечения системе соответствующей функциональной возможности.

Наличие собственного хранилища данных

У микросервиса должно быть собственное хранилище, в котором находятся необходимые ему данные. Это еще одно следствие того, что сфера действия микросервиса — завершенная функциональная возможность. Большей части бизнес-возможностей требуется какое-либо хранилище данных. Например, микросервису Product Catalog необходимо хранить определенную информацию о каждом из товаров. Сохранение слабой связанности микросервиса Product Catalog с другими микросервисами обеспечивается тем, что содержащее информацию о товарах хранилище данных полностью принадлежит ему. Только микросервис Product Catalog определяет, как и когда сохраняется информация о товарах. Как показано на рис. 1.4, другие микросервисы, например микросервис Shopping Cart, могут обращаться к информации о товарах только через интерфейс микросервиса Product Catalog, а не напрямую в хранилище данных Product Catalog.

То, что у каждого микросервиса имеется собственное хранилище данных, позволяет использовать различные технологии баз данных для различных микросервисов в зависимости от потребностей конкретного микросервиса. Например, микросервис Product Catalog может использовать для хранения информации о товарах СУБД SQL Server, микросервис Shopping Cart — хранить содержимое корзины заказов всех пользователей в СУБД Redis, а микросервис Recommendations — использовать для выдачи рекомендаций индекс Elasticsearch. Выбираемые для микросервиса технологии баз данных — часть реализации, невидимая другим микросервисам.

Такой подход дает каждому микросервису возможность использовать оптимальную для его задач базу данных, что приносит выигрыш в смысле времени разработки, производительности и масштабируемости. Очевидный недостаток — необходимость администрирования и сопровождения нескольких баз данных, а также работы с ними, если вы спроектировали свою систему подобным образом. Базы данных часто оказываются сложными технологически, и научиться использовать каждую

из них на уровне эксплуатации в производственной среде непросто. Следует принимать во внимание это соотношение выгод и потерь при выборе базы данных для микросервиса. Но одно из преимуществ наличия у каждого микросервиса своего хранилища данных — возможность замены одной базы данных на другую.

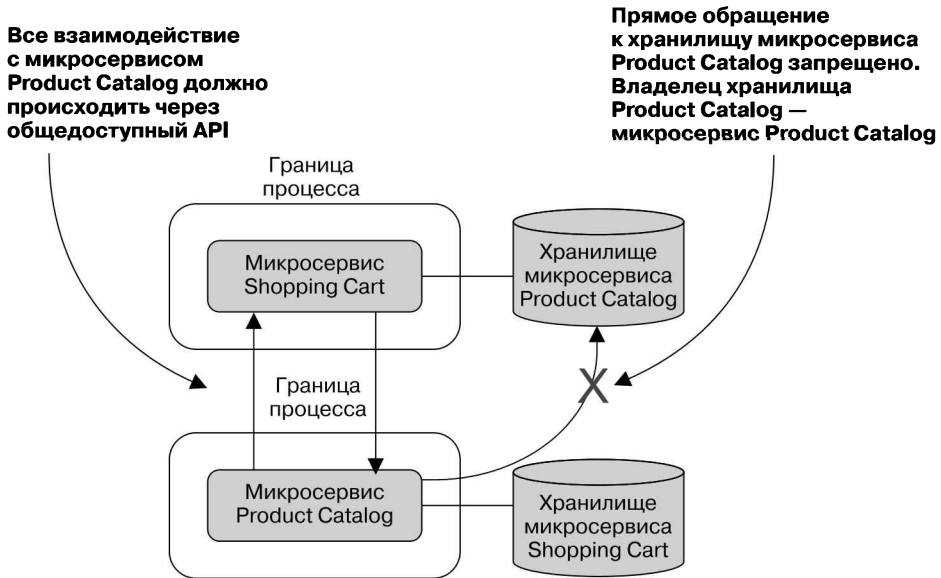


Рис. 1.4. Один микросервис не может обращаться к хранилищу другого

ПРИМЕЧАНИЕ

Мы рассмотрим владение данными, доступ к ним и их хранение в главе 5.

Возможность сопровождения небольшой командой разработчиков

До сих пор я практически не упоминал о размере микросервиса, хотя корень «микро» говорит о том, что микросервисы невелики. Я не думаю, что имеет смысл обсуждать, сколько строк кода должно быть в микросервисе или каково количество реализуемых им технических требований, сценариев использования или функциональных точек. Все это зависит от сложности обеспечиваемой микросервисом функциональной возможности.

А вот обсудить объем работ по сопровождению микросервиса имеет смысл. В качестве руководящего принципа при выборе размера микросервисов можно использовать следующее эмпирическое правило: *небольшая команда разработчиков — скажем, пятеро — должна быть в состоянии сопровождать как минимум пять микросервисов*. Термин «сопровождать» означает все аспекты поддержания работоспособности микросервиса и его пригодности для использования: разработку новой функциональности, выделение новых микросервисов из слишком разросшихся старых, обслуживание при эксплуатации в производственной среде, мониторинг, тестирование, исправление ошибок — и все, что только потребуется.

Заменяемость

Заменяемость микросервиса означает возможность перенести его с нуля за разумное время. Другими словами, *сопровождающая микросервис команда должна быть способна заменить текущую реализацию совершенно новой, причем сделать это в обычном рабочем режиме*. Этот отличительный признак накладывает еще одно ограничение на размер микросервиса: замена слишком большого микросервиса обойдется недешево, а заменить маленький вполне реально.

По какой же причине может возникнуть необходимость замены микросервиса? Возможно, код пришел в беспорядок и сопровождать его стало слишком сложно. Может быть, на производстве он демонстрирует не слишком хорошую производительность. Обе ситуации нежелательны, но изменения в технических требованиях с течением времени часто приводят к тому, что имеет смысл заменить код, а не поддерживать. Если размер микросервиса достаточно мал для того, чтобы переписать его за разумный промежуток времени, в периодическом появлении таких ситуаций нет ничего страшного. Команда разработчиков переписывает его, учитывая опыт создания текущей реализации и всех новых технических требований.

Мы познакомились с отличительными признаками микросервисов. Обсудим теперь выгоды от их использования, затраты и другие соображения.

1.2. Почему именно микросервисы?

Создание системы микросервисов, соответствующих описанному в предыдущем разделе принципам, несет очевидные выгоды: они масштабируемы и отказоустойчивы, и время от начала их разработки до внедрения в производство невелико. Эти выгоды реальны, поскольку надлежащим образом созданные микросервисы:

- ❑ обеспечивают возможность непрерывной доставки ПО;
- ❑ обеспечивают эффективность процесса разработки в силу чрезвычайной легкости их сопровождения;
- ❑ изначально выполнены устойчивыми к ошибкам;
- ❑ масштабируются в сторону увеличения или уменьшения независимо друг от друга.

Обсудим эти пункты подробнее.

Возможность непрерывной доставки ПО

Архитектурный стиль микросервисов учитывает необходимость непрерывной доставки ПО. Это делается благодаря акценту на сервисах, которые:

- ❑ можно быстро разработать и изменить;
- ❑ можно всесторонне тестировать с помощью автоматизированных тестов;
- ❑ можно развертывать независимо друг от друга;
- ❑ работают эффективно.

Эти свойства делают возможной непрерывную доставку ПО, но непрерывная доставка отнюдь не вытекает из использования архитектуры микросервисов. Их взаимосвязь носит более сложный характер: реализация непрерывной доставки упрощается благодаря микросервисам по сравнению с более традиционной сервис-ориентированной архитектурой. В то же время только при надежном и эффективном развертывании сервисов появляется возможность полностью реализовать концепцию микросервисов. Непрерывная доставка ПО и микросервисы взаимодополняют друг друга.

Выгоды от непрерывной доставки ПО хорошо известны. Это новышенная адаптивность на бизнес-уровне, надежность выпуска ПО, снижение рисков и новышение качества программного продукта.

Что такое непрерывная доставка ПО

Непрерывная доставка ПО — применяемая при разработке практика обеспечения возможности быстрого развертывания программного обеспечения в производственной среде в любой момент. Развертывание в производственной среде остается бизнес-решением, но практикующие непрерывную доставку команды предпочитают выполнять его часто и развертывать свежеразработанное программное обеспечение вскоре после попадания его в систему контроля исходных кодов.

Существуют два основных требования к непрерывной доставке. Во-первых, программное обеспечение всегда должно находиться в полностью рабочем состоянии. Чтобы добиться этого, команда должна уделять особое внимание контролю качества. Это ведет к повышению уровня автоматизации тестов и разбиению разработки на небольшие куски. Во-вторых, процесс развертывания должен быть надежным, повторяемым и быстрым, чтобы можно было часто развертывать программное обеспечение в производственной среде. Это достигается полной автоматизацией процесса развертывания и высокой степенью мониторинга состояния производственной среды.

Хотя непрерывная доставка ПО требует немалых технических навыков, это в большей степени вопрос организации процесса и культуры труда. Подобный уровень качества, автоматизации и мониторинга требует тесного сотрудничества между всеми вовлеченными в разработку и эксплуатацию программного обеспечения сторонами, включая предпринимателей, разработчиков, экспертов по информационной безопасности и системных администраторов. Другими словами, он требует культуры DevOps, когда те, кто занимается разработкой и эксплуатацией, сотрудничают и учатся друг у друга.

Непрерывная доставка ПО — неотъемлемый спутник микросервисов. При невозможности быстрого и дешевого развертывания отдельных микросервисов реализация системы микросервисов быстро стала бы очень затратной. В отсутствие автоматизации развертывания микросервисов количество необходимой при развертывании полной системы микросервисов ручной работы было бы ошеломляющим.

Непрерывной доставке сопутствует культура DevOps, также необходимая для микросервисов. Для успешной работы системы микросервисов нужно, чтобы каждый участник процесса вносил свой вклад в обеспечение беспроблемной работы сервиса и максимальной прозрачности состояния производственной среды. Это тре-

бует сотрудничества различных специалистов: с опытом эксплуатации, опытом разработки, опытом работы в сфере безопасности, а также со знанием предметной области, помимо прочего.

Эта книга не посвящена непрерывной доставке ПО или DevOps, но мы предполагаем, что в среде, в которой вы разрабатываете микросервисы, используется непрерывная доставка. Создаваемые в данной книге сервисы можно развертывать в локальных дата-центрах или в облаке с применением любого количества технологий автоматизации развертывания, допускающих использование платформы .NET. Эта книга охватывает последствия применения непрерывной доставки ПО и DevOps для отдельных микросервисов. В части III книги мы подробнее рассмотрим создание платформы, принимающей на себя немало эксплуатационных проблем, с которыми приходится иметь дело всем микросервисам. Кроме того, в приложении Б рассмотрим основные возможности ввода в производственную среду разработанных на протяжении всей книги микросервисов.

Высокий уровень удобства сопровождения

Грамотно спроектированные и разработанные микросервисы легки в сопровождении с нескольких точек зрения. *С точки зрения разработчика*, в обеспечении удобства сопровождения микросервисов играют роль несколько факторов.

- ❑ Каждый грамотно спроектированный микросервис обеспечивает *единственную функциональную возможность*. Не две и не три — только одну.
- ❑ У микросервиса имеется собственное хранилище данных. Никакие другие сервисы не могут взаимодействовать с хранилищем данных микросервиса. Этот факт, а также типичный объем кода микросервиса означают, что можно охватить весь микросервис одним взглядом и сразу понять, что он собой представляет.
- ❑ Для грамотно написанных микросервисов могут и должны быть созданы всесторонние автоматизированные тесты.

С точки зрения эксплуатации в обеспечении удобства сопровождения микросервисов играют роль два фактора.

- ❑ Небольшая команда разработчиков может сопровождать несколько микросервисов. Микросервисы должны быть спроектированы в расчете на эффективную эксплуатацию, из чего следует, что должна существовать возможность легко определять текущее состояние любого микросервиса.
- ❑ Все микросервисы можно разворачивать по отдельности.

Из этого следует, что необходимо своевременно обнаруживать и решать возникающие при эксплуатации в производственной среде проблемы, например, масштабированием соответствующего микросервиса или развертыванием его новой версии. То, что у микросервиса в соответствии с одним из отличительных признаков имеется собственное хранилище данных, также новышает удобство его сопровождения при эксплуатации в производственной среде, поскольку область сопровождения хранилища данных ограничивается его микросервисом-владельцем.

Тенденция к облегченности

В силу того что каждый микросервис отвечает только за одну функциональную возможность, микросервисы по своей природе невелики как по области действия, так и по размеру кода. Основное преимущество микросервисов как раз и состоит в их простоте, вызванной ограниченностью области действия.

При разработке микросервисов важно избегать усложнения их кода вследствие использования больших, сложных фреймворков, библиотек и программных продуктов, даже если вам кажется, что их функциональность может пригодиться в будущем. Весьма вероятно, что она не понадобится, так что лучше предпочесть меньшие, облегченные технологии, выполняющие только то, что необходимо микросервису прямо сейчас. Помните: микросервисы заменяемы, так что, если первоначальные технологии перестанут удовлетворять ваши потребности, всегда можно полностью переписать микросервис в рамках разумного бюджета.

Надежность и масштабируемость

Распределенная архитектура на основе микросервисов дает возможность масштабировать каждый из сервисов отдельно с учетом узких мест. Более того, микросервисы предпочитают асинхронное взаимодействие с помощью событий и придают особое значение отказоустойчивости везде, где требуется синхронное взаимодействие. Эти свойства при подлежащей реализации обеспечивают высокую доступность и хорошую масштабируемость системы.

1.3. Издержки и недостатки микросервисов

Выбор архитектуры микросервисов влечет за собой существенные издержки, которые не следует игнорировать.

- ❑ Системы микросервисов — распределенные системы. Связанные с выбором распределенной архитектуры издержки хорошо известны. Такие системы сложнее для осмысления и тестирования, чем монолитные, и обмен сообщениями между процессами или по сети происходит на порядки медленнее, чем вызовы методов внутри процесса.
- ❑ Системы микросервисов состоят из множества микросервисов, и необходимо разработать и развернуть каждый из них, а также обеспечить управление им при эксплуатации в производственной среде. Это влечет за собой большое количество развертываний и сложную схему установки на производстве.
- ❑ У каждого микросервиса — собственная база кода. Следовательно, рефакторинг с перемещением кода из одного микросервиса в другой требует больших усилий. Необходимо заранее позаботиться о правильном определении области действия всех микросервисов.

Прежде чем пахать построение системы микросервисов, желательно оценить, настолько реализуемая система сложна, чтобы оправдать падающие расходы.

Насколько производительны микросервисы

В спорах о том, использовать ли микросервисы, часто всплывает вопрос: будет ли основанная на них система столь же производительна, как и не основанная? Основной довод против: каждый запрос в созданной из множества взаимодействующих микросервисов системе будет затрагивать несколько микросервисов, а их взаимодействие потребует выполнения между ними удаленных обращений. Что произойдет при поступлении пользовательского запроса? Будет ли выполняться длинная цепочка из удаленных обращений от одного микросервиса к другому? Учитывая, что удаленные вызовы выполняются на порядки медленнее, чем вызовы методов внутри процесса, это выглядит довольно медленным.

Проблема с этим доводом состоит в том, что в подобной системе будут выполняться примерно те же обращения между различными частями системы, как если бы все происходило в одном процессе. Во-первых, взаимодействие между микросервисами будет намного более «мелкозернистым», чем обычно бывают вызовы внутри процесса. Во-вторых, как мы обсудим в главах 4 и 5, основанное на событиях асинхронное взаимодействие предпочтительнее, чем выполнение синхронных удаленных вызовов, и мы будем хранить копии одних и тех же данных в нескольких микросервисах, чтобы гарантировать их доступность там, где они нужны. В общем, эти методы разительно снижают необходимость заставлять пользователя ждать во время выполнения удаленных вызовов. Более того, «мелкозернистая» структура микросервисов позволяет масштабировать отдельные перегруженные части системы.

Невозможно дать однозначный ответ «да» или «нет» на вопрос, насколько производительны микросервисы. Но я могу сказать, что производительности хорошо спроектированной системы микросервисов более чем достаточно для многих, если не всех, систем.

1.4. С нуля или на базе существующей системы?

Следует ли вводить микросервисы в новый проект с самого начала, или они подходят только для уже существующих больших систем? Этот вопрос часто всплывает при обсуждении микросервисов.

Микросервисный архитектурный стиль обязан своим появлением тому факту, что системы многих предприятий вначале были небольшими, но с течением времени разрослись. Многие из этих систем состоят из одного огромного приложения — монолита, демонстрирующего широко известные недостатки больших монолитных систем.

- ❑ Сильная связанность внутри базы кода.
- ❑ Скрытая связанность между субкомпонентами, которая не видна компилятору, поскольку основана на выраженных в неявной форме знаниях о форматировании определенных строк, о том, как используются определенные столбцы в базе данных, и т. д.
- ❑ Развертывание приложения — длительный процесс, который может потребовать вовлечения нескольких специалистов и длительного простоя системы.
- ❑ Архитектура такой системы рассчитана на все случаи жизни, на работу с наиболее сложными компонентами. Если потребовать единого образа архитектуры всего

мополита, наименее сложные части системы окажутся технически переусложненными. Это относится к разбиению на слои, выбору технологий и паттернов (шаблонов) и т. д.

Микросервисная архитектура возникла как результат решения этих проблем в существующих мополитных системах. Многократно разбивая субкомпоненты мополита на все меньшие и легче управляемые части, вы в конце концов придете к созданию микросервисов¹.

В то же время постоянно возникают новые проекты. Подходят ли микросервисы для этих создаваемых с нуля проектов? Все зависит от обстоятельств. Вот вопросы, которые следует задать себе в таком случае.

- ☐ Пойдет ли этой системе на пользу возможность отдельного развертывания подсистем?
- ☐ Сможете ли вы создать довольно сильно автоматизированную систему развертывания?
- ☐ Достаточно ли хорошо вы знакомы с предметной областью, чтобы правильно распознать и разделить различные независимые бизнес-возможности системы?
- ☐ Достаточно ли широка сфера действия системы, чтобы оправдать сложность распределенной архитектуры?
- ☐ Достаточно ли широка сфера действия системы, чтобы оправдать затраты на создание системы автоматизации развертывания?
- ☐ Просуществует ли проект достаточно долго, чтобы оправдать предварительные вложения средств в автоматизацию и развертывание?

Некоторые создаваемые с нуля проекты вполне удовлетворяют этим критериям, а значит, могут выиграть от того, что будут использовать архитектуру микросервисов с самого начала.

1.5. Многократное использование кода

Использование архитектуры микросервисов ведет к появлению в системе множества сервисов, у каждого — своя база кода, которую придется сопровождать. Задача было бы применять код многократно для различных сервисов, чтобы снизить затраты на сопровождение, но, хотя в повторном использовании кода есть потенциальные преимущества, перенос кода из сервиса в отдельную многократно используемую библиотеку влечет за собой скрытые издержки.

- ☐ У сервиса становится одной зависимостью больше, и для того, чтобы понять, как он работает, необходимо будет разобраться с этой зависимостью. Это не означает

¹ Некоторые приверженцы микросервисов утверждают, что наиболее правильный способ создания микросервисов — многократное применение шаблона Strangler («Дроссель») к различным субкомпонентам монолита. См. *Fowler M. Monolith First*, 2015. — June 3 [Электронный ресурс]. — Режим доступа: <http://martinfowler.com/bliki/MonolithFirst.html>.

увеличения количества кода, в котором пужно разобраться, по, перемещая код из сервиса в библиотеку, вы разносите его «географически», усложняя навигацию по коду и его измеппение.

- ❑ Находящийся в повой библиотеке код необходимо разрабатывать и поддерживать с учетом мпожества сцепариев использования. Это потребуе памного бóльших усилий, чем разработка в расчете только па одип сценарий использования.
- ❑ Совместно используемая библиотека привпосит определепную связапность между примепяющими ее сервисами. Вызваппые сервисом А измеппения в библиотеке, возможно, не требуются в сервисе В. Нужно ли будет меппать сервис В в соответствии с новой версией библиотеки, хотя ему опа, строго говоря, и не требуется? Если модифицировать сервис В, в нем появится не используемый им код и, что еще хуже, сервис В будет подвергаться риску возникповепия вызываемых этим кодом ошибок. Если же не модифицировать, то у вас будет песколько версий библиотеки в производственной среде, что еще больше затруднит ее сопровождение. Оба сцепария приводят к определепным сложпостям или в сервисе В, или в общем ландшафте сервиса.

Эти момепты особепно важпы для бизнес-кода. Никогда не следует использовать бизнес-код повторно в различных микросервисах, так как это ведет к пагубной связапности микросервисов.

Учитывая все это, следует повторно применять код осторожно и осмотрительно. Однако существуют веские доводы в пользу повторного использования кода иппфраструктуры, который воплощает техпические возможности.

Чтобы размер сервисов оставался пебольшим и опи были запяты реализацией одной фуппкциональной возможности, часто лучше паписать повый сервис с нуля, а не добавлять фуппкциональность в существующий. Главное — сделать это быстро и безболезненно, для чего и пригодится повторное использование кода сервисами. Как во всех подробностях изложено в главе 3, существует мпожество техпических возможностей, которые должны реализовывать все сервисы, чтобы хорошо вписываться в общий ландшафт архитектуры сервисов. Этот код не пужно писать для каждого сервиса отдельно, его можно использовать мпожескратно в разных сервисах ради единообразия обработки всех техпических вопросов и для спижепия объема работ по созданию пового сервиса.

1.6. Обслуживание запросов пользователя: пример совместной работы микросервисов

Чтобы понять, как работает микросервисная архитектура, рассмотрим пример: посетитель ипптерпет-магазина добавляет товар в корзину заказов. С точки зрения клиентского кода в систему прикладной части через микросервис API Gateway (API-шлюз) передается запрос Ajax, а возвращается обновлеппая корзина с информацией о цене. Это простое взаимодействие показано на рис. 1.5. Мы вернемся к разговору об API-шлюзах в главе 12.

Ничего удивительного или захватывающего в этом нет. Самое интересное заключается во взаимодействиях с целью выполнения запроса, протекающих «под капотом» микросервиса API Gateway. Чтобы добавить новый товар в корзину заказов, микросервис API Gateway использует несколько других микросервисов. Каждый микросервис представляет собой отдельный процесс, и в данном примере они взаимодействуют посредством HTTP-запросов.



Рис. 1.5. Код клиентской части, выполняя запрос для добавления товара в корзину заказов, обменивается сообщениями только с микросервисом API-шлюза. Происходящее «за кулисами» шлюза не видно

Основная обработка пользовательского запроса

Все микросервисы и их взаимодействия, происходящие в процессе выполнения пользовательского запроса по добавлению товара в корзину заказов, показаны на рис. 1.6.

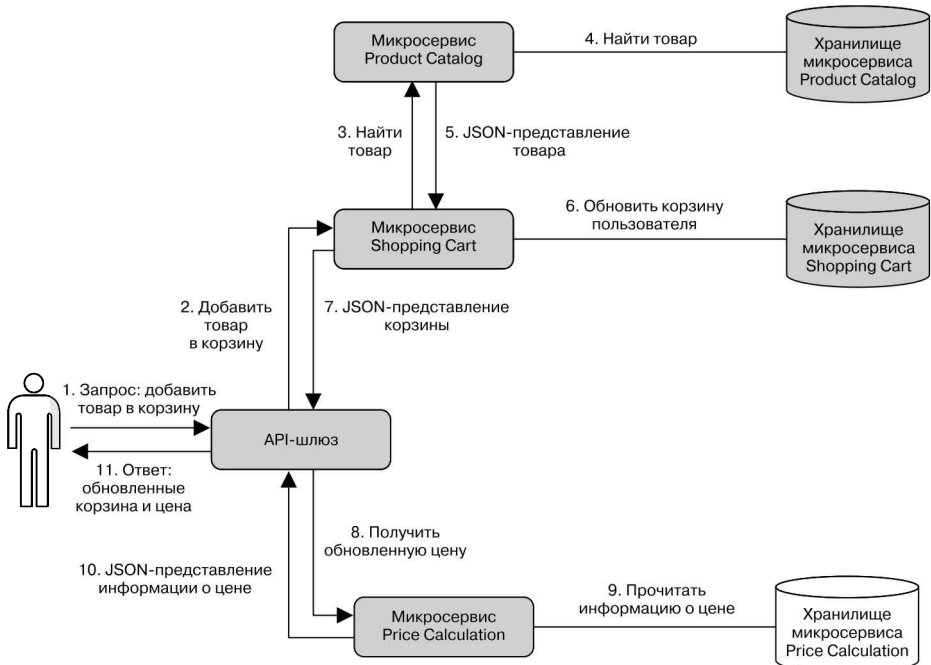


Рис. 1.6. Клиент видит только микросервис API Gateway, но это лишь тонкий наружный слой системы микросервисов. Стрелки обозначают обращения между различными частями системы, а номера на них показывают последовательность вызовов

Запрос на добавление товара в корзину заказов разбит на меньшие задачи, каждую из которых выполняет отдельный микросервис.

- ❑ Микросервис API-шлюза отвечает только за беглую проверку входящего запроса. После проверки выполнение задачи передается сначала микросервису Shopping Cart, а затем микросервису Price Calculation.
- ❑ Микросервис Shopping Cart использует другой микросервис — Product Catalog — для поиска нужной информации о добавляемом в корзину товаре. После этого Shopping Cart сохраняет информацию о корзине заказов пользователя в своем хранилище данных и возвращает микросервису API Gateway представление обновленной корзины. Ради повышения производительности и большей устойчивости к ошибкам микросервис Shopping Cart, вероятно, будет кэшировать поступающие от микросервиса Product Catalog ответы.
- ❑ Микросервис Price Calculation использует текущие бизнес-правила интернет-магазина для расчета общей цены товаров в корзине заказов пользователя с учетом всех применимых в данном случае скидок.

Каждый из этих совместно выполняющих запрос пользователя микросервисов предназначен для решения конкретной узкой задачи и знает об остальных микросервисах настолько мало, насколько это возможно. Например, микросервис Shopping Cart ничего не знает оцепообразовании или микросервисе Price Calculation, а также о том, как товары хранятся в микросервисе Product Catalog. Это основная идея микросервисов: на каждый из них возлагается только одна обязанность.

Побочные действия пользовательских запросов

На этом сайте интернет-магазина, когда пользователь добавляет в свою корзину товар, выполняются несколько действий, помимо самого добавления товара в корзину.

1. Рекомендательный механизм обновляет внутреннюю модель, чтобы отразить факт проявления пользователем высокой степени интереса к конкретному товару.
2. Сервис отслеживания действий пользователя фиксирует факт добавления пользователем товара в корзину в своей базе данных. Потом можно будет воспользоваться этой информацией для создания отчетов или других целей, связанных с бизнес-аналитикой.

Нет необходимости совершать эти действия в контексте пользовательского запроса — их можно с таким же успехом произвести после окончания выполнения запроса, когда пользователь уже получит ответ и не будет ждать отклика от серверной части.

Оба действия можно рассматривать как побочные для пользовательских запросов. Они не являются непосредственными следствиями выполнения запроса с требованием обновить пользовательскую корзину заказов, это вторичные действия, происходящие в результате добавления товара в корзину. На рис. 1.7 детально проиллюстрированы побочные действия добавления товара в корзину.

Выполнение побочных действий запускает публикуемое микросервисом Shopping Cart событие `ItemAddedtoCart`. Два других микросервиса, подписанных на исходящие

от микросервиса Shopping Cart события, выполняются соответствующие действия при наступлении событий, таких как `ItemAddedtoCart`. Эти два подписчика реагируют на события асинхронно — вне контекста исходного запроса, так что побочные действия могут выполняться параллельно с основным потоком обработки запроса или после его завершения.

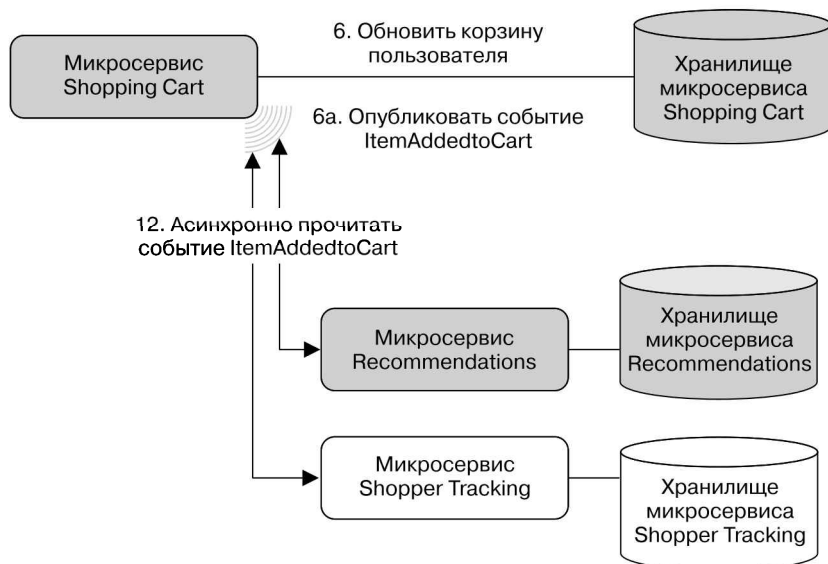


Рис. 1.7. Микросервис Shopping Cart публикует события, а другие подписанные на него микросервисы реагируют на них

ПРИМЕЧАНИЕ

Реализация подобного взаимодействия, основанного на потоках событий, рассматривается в главе 4.

Общая картина

В целом в обработке запроса на добавление товара в корзину заказов участвуют шесть микросервисов (рис. 1.8). Ни один из них не знает ничего о внутреннем устройстве остальных. У пяти микросервисов есть собственные хранилища данных, обслуживающие только их. Часть обработки происходит синхронно в контексте пользовательского запроса, а часть — асинхронно.

Такова типичная система микросервисов. Запросы обрабатываются при взаимодействии нескольких микросервисов, на каждый из которых, независимый от других настолько, насколько это возможно, возлагается только одна обязанность.

После подробного рассмотрения конкретного примера обработки пользовательского запроса системой микросервисов пришло время ненадолго остановиться на стеке технологий платформы .NET для микросервисов.

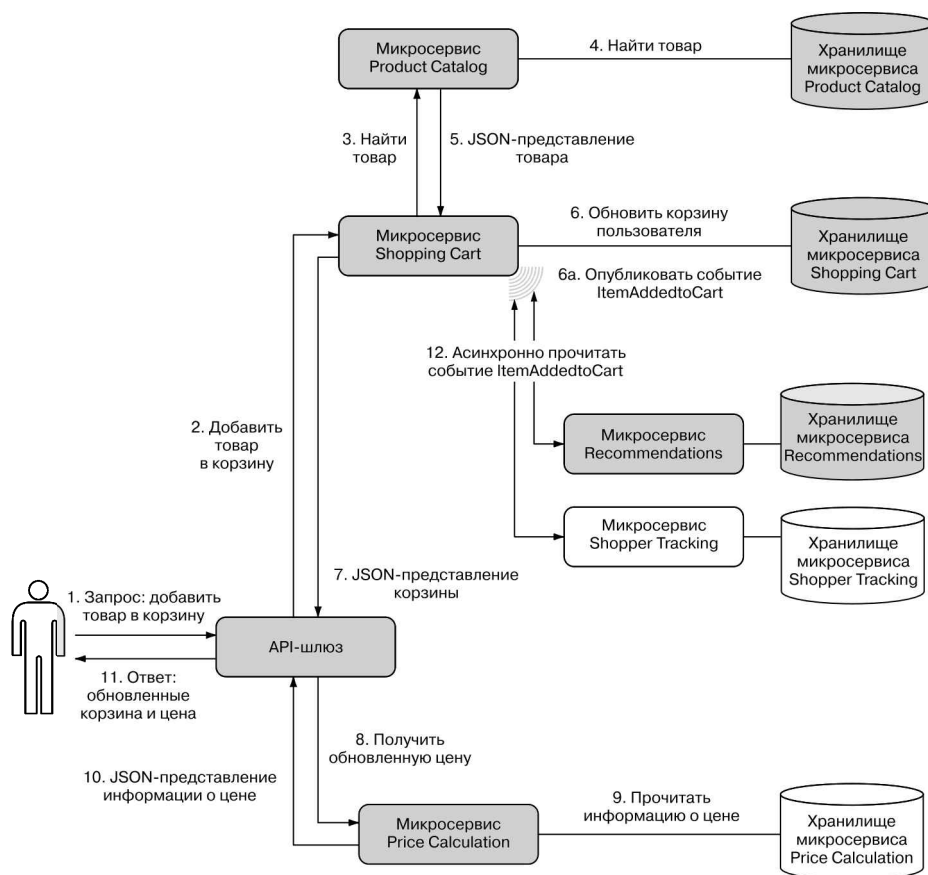


Рис. 1.8. При добавлении пользователем товара в свою корзину заказов клиентская часть делает запрос к микросервису API gateway, который совместно с другими микросервисами выполняет запрос. Во время обработки микросервисы могут порождать события, на которые другие микросервисы могут подписываться и затем обрабатывать их асинхронно

1.7. Стек технологий платформы .NET для микросервисов

Самое время познакомиться с двумя технологиями, которые будут использоваться в нашей книге чаще всего: Nancy и OWIN.

Фреймворк Nancy

Nancy (<http://nancyfx.org/>) — основанный на платформе .NET веб-фреймворк с открытым исходным кодом, явным образом декларирующий своей целью предоставление разработчикам супер-пупер-крутого способа разработки веб-приложений и сервисов.

Термин «*супер-пупер-крутой способ*» в данном случае описывает базовые принципы фреймворка Nancy.

- ❑ *Nancy просто работает.* В нем заранее заданы разумные значения по умолчанию для всего, чего нужно, так что перед началом работы не требуется никаких или почти никаких предварительных настроек или формальностей. При этом приложения можно расширять, чтобы рационально использовать различные аспекты фреймворка Nancy. Все работает логично и разумно прямо «из коробки».
- ❑ *Nancy можно легко адаптировать к конкретной задаче.* Если вы наткнетесь на ситуацию, при которой значения по умолчанию фреймворка Nancy не подходят для вашего приложения, вам не составит труда построить Nancy под свои потребности. Если обычной постройки недостаточно, абсолютно все в фреймворке Nancy разбито на компоненты и может быть заменено вашей собственной реализацией.
- ❑ *Фреймворк Nancy не требует особых формальностей и не мешает вам работать.* При создании приложения Nancy фреймворк не становится на вашем пути. API спроектированы гибкими, они позволяют писать код так, как вам удобно. Другими словами, при использовании фреймворка Nancy можно сосредоточиться на коде своего приложения, а не разбираться с Nancy.
- ❑ *Приложения Nancy легко тестируются.* Сам по себе фреймворк Nancy спроектирован с возможностью тестирования. Nancy обеспечивает также удобный поток разработки ваших собственных приложений Nancy с ориентацией на тестирование. Помимо того что API фреймворка Nancy спроектированы с учетом тестируемости, Nancy поставляется с сопутствующей библиотекой Nancy.Testing, облегчающей написание тестов для приложений Nancy.

Благодаря этим базовым принципам и обеспечиваемому ими супер-пупер-крутому способу разработки Nancy — мой излюбленный веб-фреймворк для создания микросервисов на платформе .NET. Мы будем использовать Nancy на протяжении всей книги, и по мере описания трудностей реализации микросервисов я продемонстрирую множество его возможностей. В последнем разделе этой главы вы увидите небольшой фрагмент кода приложения Nancy. Однако сначала я хотел бы познакомить вас со стандартом OWIN.

Стандарт OWIN

Открытый веб-интерфейс для платформы .NET (Open Web Interface for .NET, OWIN, <http://owin.org/>) — это открытый стандарт, описывающий интерфейс между веб-серверами .NET и веб-приложениями .NET. Интерфейс OWIN отцепляет веб-сервер и веб-приложение друг от друга. OWIN-совместимый веб-сервер, получив HTTP-запросы из сети, передает их на обработку веб-приложению через стандартный интерфейс OWIN. Такой веб-сервер не знает никаких подробностей о веб-приложении. Все, что ему известно и важно для него, — это то, что веб-приложение может получать запросы через интерфейс OWIN. Аналогично веб-приложение не знает ничего о веб-сервере. Оно знает только, что через интерфейс OWIN поступают запросы.

Как показано на рис. 1.9, это достигается использованием адаптера, реализующего поверх веб-сервера интерфейс OWIN, и OWIN-совместимого веб-фреймворка для реализации веб-приложения. Все поступающие запросы передаются веб-сервером через OWIN-адаптер. Затем OWIN-адаптер и веб-фреймворк обмениваются сообщениями через интерфейс OWIN. В результате веб-приложение расцепляется с веб-сервером, что обеспечивает веб-приложениям определенную переносимость.

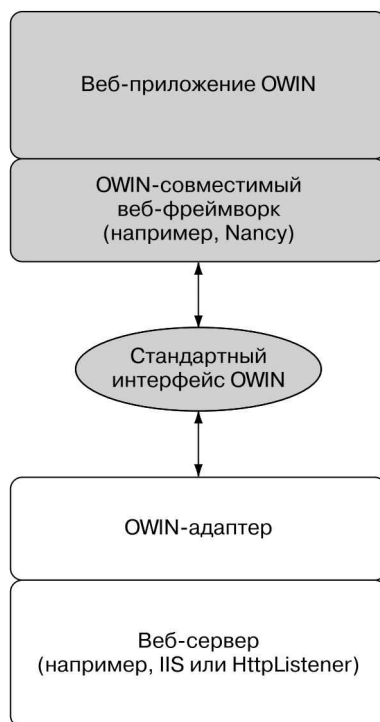


Рис. 1.9. Реализованное поверх интерфейса OWIN веб-приложение обменивается сообщениями с веб-сервером через стандартизованный интерфейс OWIN. Сервер может реализовывать интерфейс OWIN непосредственно или с помощью адаптера, осуществляющего взаимодействие между интерфейсом OWIN и интерфейсом веб-сервера

Поскольку у веб-серверов и веб-приложений есть информация только об интерфейсе OWIN, появляется возможность вставлять компоненты между веб-сервером и веб-приложением без каких-либо изменений в них. Эти компоненты носят название «*промежуточное ПО OWIN*» (OWIN middleware). На рис. 1.10 показан веб-сервер OWIN с журналирующим запросы промежуточным ПО OWIN и веб-приложением OWIN поверх него. У веб-сервера нет никакой возможности узнать, что обмен сообщениями происходит не напрямую с веб-приложением, — промежуточное ПО использует тот же интерфейс, так что оно выглядит для веб-сервера приложением, а для приложения — веб-сервером.

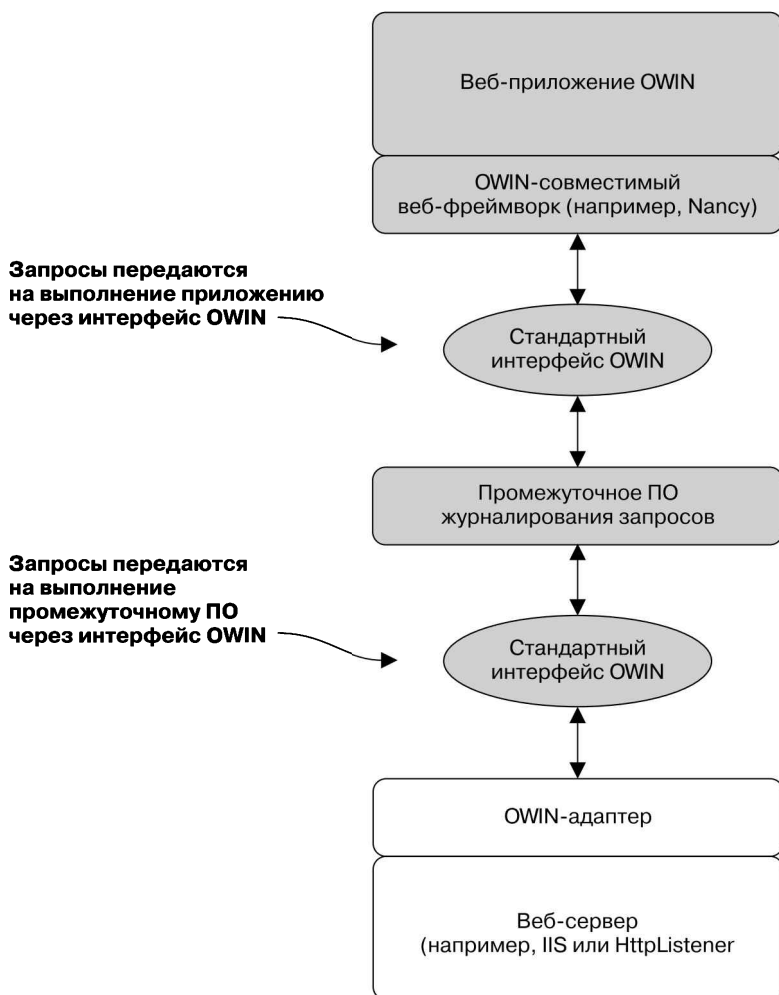


Рис. 1.10. Веб-сервер OWIN с промежуточным ПО OWIN и веб-приложением OWIN поверх него. Веб-сервер передает входящий запрос вышележащим слоям — в данном случае промежуточному ПО журналирования запросов, заносящему в журнал запись о запросе, а в дальнейшем передающему его на выполнение веб-приложению. Промежуточное ПО для веб-сервера OWIN выглядит совместимым приложением, а для приложения — OWIN-совместимым веб-сервером

На рис. 1.10 стек включает только один элемент промежуточного программного обеспечения, но это не мешает вам добавить еще. Один элемент промежуточного ПО может передавать обязанности другому, так же как и приложению. Если все элементы используют интерфейс OWIN, вы можете формировать стек из такого количества элементов, какое вам нужно.

Фреймворк Nancy — OWIN-совместимый, так что создаваемые с помощью него микросервисы могут служить OWIN-приложениями в копвейере OWIN. В даппой

книге будем использовать промежуточное ПО для того, чтобы реализовать несколько элементов сквозной функциональности, которые плохо вписываются в код приложения. В части III мы детальнее рассмотрим реализацию поддержки мониторинга, журналирование показателей производительности, журналирование запросов и процедуры безопасности в качестве элементов промежуточного программного обеспечения, которые можно повторно использовать во всех микросервисах.

Два основных преимущества использования промежуточного ПО: сквозная функциональность аккуратно отделяется от логики приложения, промежуточное ПО можно применять повторно, что позволяет легко создавать новые микросервисы, не тратя времени на повторную упаковку того же кода для мониторинга, журналирования и обеспечения безопасности.

Настройка среды разработки

Чтобы писать и программировать свой первый микросервис, вам понадобятся правильные инструменты. Чтобы следить за примерами из данной книги, нужна среда разработки для создания приложений на платформе ASP.NET Core. «Разработка приложений ASP.NET» и Visual Studio обычно воспринимаются как синонимы, но в случае кросс-платформенной Core-версии платформы ASP.NET есть и другие возможности.

- ❑ Наиболее распространённое IDE для работающих в операционной системе Windows, вероятно, все же Visual Studio 2015 с плагином Web Tools Extension для ASP.NET Core.
- ❑ Пользователи, работающие в Linux, OS X или Windows, могут также применять Visual Studio Code, редактор Atom с плагином OmniSharp или JetBrains Rider.

Все они — Visual Studio 2015, Visual Studio Code, редактор Atom с плагином OmniSharp и JetBrains Rider — поддерживают разработку и выполнение приложений ASP.NET Core. Все они включают удобный редактор кода на языке C# с поддержкой технологии автодополнения IntelliSense и рефакторинга. Все они разработаны с учетом ASP.NET Core и умеют запускать приложения на платформе ASP.NET Core. Кроме того, все они бесплатны, даже версия Community среды Visual Studio.

СОВЕТ

На момент написания данной книги только Visual Studio и Visual Studio Code предоставляют возможность отладки приложений ASP.NET Core.

После установки IDE нужно будет установить также утилиту командной строки ASP.NET Core. Для этого следуйте инструкциям по установке .NET Core с сайта <http://dot.net/>. Таким образом вы получите утилиту командной строки dotnet, которую сможете использовать для множества различных связанных с микросервисами задач, включая создание и восстановление пакетов NuGet, а также выполнение микросервисов.

Помимо IDE и утилиты dotnet, вам понадобится утилита для выполнения HTTP-запросов. Я рекомендую использовать Postman, но утилиты Fiddler и curl тоже неплохи и весьма популярны. Чтобы следить за примерами из этой книги, подойдет любая из них.

ПРИМЕЧАНИЕ

В приложении А вы найдете информацию о скачивании, установке и краткую информацию об использовании среды Visual Studio, редактора Visual Code, редактора Atom с плагином OmniSharp, среды JetBrains Rider и утилиты Postman. Самое время установить и настроить те из них, которые вам нравятся больше всего, — они пригодятся во время чтения книги.

1.8. Пример простого микросервиса

После окончания установки и запуска среды разработки наступает время примеров микросервисов а-ля Hello World. Воспользуемся фреймворком Nancy для создания микросервиса с одной конечной точкой API. Обычно у микросервиса несколько конечных точек, но для этого примера хватит и одной. Конечная точка будет возвращать текущие дату и время по Гринвичу (UTC) в формате JSON или XML в зависимости от заголовков запроса (рис. 1.11). Мы также добавим в пример элемент промежуточного ПО OWIN, журналирующий входящий запрос в консоль.

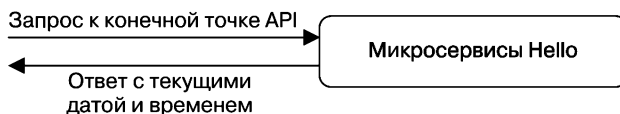


Рис. 1.11. Микросервис в стиле Hello World, возвращающий текущие дату и время

ПРИМЕЧАНИЕ

Говоря о конечной точке API, или конечной точке HTTP, или просто конечной точке, я имею в виду URL, по которому один из микросервисов реагирует на HTTP-запросы.

Для реализации этого примера необходимо пройти следующие четыре этапа.

1. Создать пустое приложение ASP.NET Core.
2. Добавить в это приложение фреймворк Nancy.
3. Добавить модуль Nancy с реализацией конечной точки.
4. Добавить промежуточное ПО OWIN, журналирующее все запросы в консоль.

В следующих разделах мы рассмотрим каждый из этих этапов подробнее.

Создание пустого приложения ASP.NET Core

Прежде всего необходимо создать пустое приложение ASP.NET Core с названием *HelloMicroservices*. Если вы решили использовать Visual Studio, то создать проект можно, выбрав из меню пункты **File** ► **New** ► **Project** (Файл ► Создать ► Проект). Выберите в диалоговом окне **ASP.NET Web Application** (Веб-приложение), затем **Empty** (Пустое) в **ASP.NET Core Templates**.

Если вы выбрали вариант установки Visual Studio Code или Atom, следуя инструкциям из приложения А, то вы установили и утилиту для скаффолдинга Yeoman. Можете воспользоваться ею для создания пустого приложения ASP.NET Core, введя команду `yo aspnet` в командной оболочке, а затем выбрав из меню **Empty Web Application** (Пустое веб-приложение).

Когда вы создадите пустое приложение ASP.NET Core и назовете его *HelloMicroservices*, у вас должен получиться проект, содержащий следующие файлы:

- ❑ `Hellomicroservices\Program.cs`;
- ❑ `Hellomicroservices\Startup.cs`;
- ❑ `Hellomicroservices\project.json`.

В проекте будут и другие файлы, но нас интересуют прежде всего эти три.

Это полноценное приложение, готовое к запуску. Оно будет отвечать на любой HTTP-запрос строкой *Hello World*. Запустить приложение из командной строки можно, перейдя в каталог, в котором находится файл `project.json`, и набрав команду `dotnet run`.

Наше приложение выполняется на порте 5000 (обратите внимание, что при запуске его из Visual Studio порт может оказаться другим). Перейдя в браузере по адресу `http://localhost:5000`, вы получите ответ *Hello World*.

Добавление в проект фреймворка Nancy

Включить фреймворк Nancy в проект как пакет NuGet можно, добавив его в раздел `dependencies` файла `project.json`, являющегося частью нашего пустого приложения. Нам также понадобится пакет `Microsoft.AspNetCore.Owin`, так что добавим и его. Раздел `dependencies` файла `project.json` выглядит примерно так, как показано в листинге 1.1.

Листинг 1.1. Раздел `dependencies` файла `project.json`

```
"dependencies": {
  "Microsoft.NETCore.App": {
    "version": "1.0.0",
    "type": "platform"
  },
  "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
  "Microsoft.AspNetCore.Owin": "1.0.0",
  "Nancy": "2.0.0-barneyrubble"
},
```

При сохранении файла `project.json` в Visual Studio, Visual Studio Code или Atom с OmniSharp IDE будет восстанавливать пакет, то есть скачивать любые указанные в разделе `dependencies` новые пакеты. По окончании восстановления пакета Nancy будет частью нашего проекта¹.

После добавления пакета Nancy системы управления пакетами NuGet пужно сообщить ASP.NET Core о необходимости применять фреймворк Nancy. Это можно сделать в файле `Startup.cs`, который уже включен в проект. В нем уже содержится код, который следует заменить кодом из листинга 1.2. Этот код говорит ASP.NET Core о необходимости использования OWIN, после чего вставляет Nancy в конвейер OWIN.

Листинг 1.2. Настройка ASP.NET Core с помощью файла `Startup.cs`

```
namespace Hellomicroservices
{
    using Microsoft.AspNetCore.Builder;
    using Nancy.Owin;

    public class Startup
    {
        public void Configure(IApplicationBuilder app)
        {
            app.UseOwin(buildFunc =>
            {
                buildFunc.UseNancy();
            });
        }
    }
}
```

ASP.NET Core вызывает этот метод во время запуска приложения

Настройка ASP.NET Core для использования OWIN. `buildFunc` можно использовать для организации конвейера OWIN

Добавляет Nancy в конвейер OWIN. Это дает Nancy возможность обрабатывать входящие HTTP-запросы

Теперь у нас есть приложение с фреймворком Nancy на платформе ASP.NET Core. Но оно пока не может обрабатывать запросы, поскольку мы не сконфигурировали в Nancy никаких маршрутов. Если перезапустить приложение и снова перейти по адресу `http://localhost:5000` в браузере, будет возвращена ошибка 404 Not Found (404 Не найдено). Давайте это исправим.

Добавление модуля Nancy с реализацией конечной точки

Сейчас мы добавим модуль Nancy с реализацией одной конечной точки API. *Модуль Nancy* — это класс, наследующий класс `NancyModule`; он используется для описания обрабатываемых приложением конечных точек и для реализации поведения каждой конечной точки. Фреймворк Nancy автоматически находит при запуске все классы-потомки `NancyModule` и регистрирует все описанные в модулях Nancy маршруты. Описание маршрутов в модулях Nancy производится посредством внутреннего DSL для работы с протоколом HTTP фреймворка Nancy. Для создания модуля Nancy

¹ Работая из командной строки, вам необходимо будет выполнить команду `dotnet restore` для этой цели. — *Примеч. пер.*

создадим файл под названием `CurrentDateTimeModule.cs` и вставим в него следующий код (листинг 1.3).

Листинг 1.3. Модуль Nancy

```
namespace Hellomicroservices
{
    using System;
    using Nancy;

    public class CurrentDateTimeModule
    {
        : NancyModule
        {
            public CurrentDateTimeModule()
            {
                Get("/", _ => DateTime.UtcNow);
            }
        }
    }
}
```

Объявляем модуль Nancy

Объявляем маршрут и обработчик маршрута. Устанавливаем запросы для возвращения текущей даты и времени в виде JSON или XML

В этом модуле мы объявили маршрут для пути `/` с помощью выражения `Get("/", ...?)`. Мы также сообщили фреймворку Nancy, что все HTTP-запросы к `/` необходимо обрабатывать с помощью лямбда-выражения `_ ? DateTime.UtcNow`. Всякий раз, когда на маршрут `/` поступает запрос, возвращается ответ с текущими датой и временем.

ПРИМЕЧАНИЕ

По традиции я применяю символ `_` в качестве наименования для параметров лямбда-выражения, не используемых справа от стрелки лямбда-выражения.

Теперь можно перезапустить приложение и снова перейти в браузере по адресу `http://localhost:5000`. Браузер обратится к маршруту в модуле Nancy и отобразит ошибку. Почему? Потому что Nancy не может найти представление для маршрута `/`. Это нормально. Предназначение этого маленького приложения состоит не в выдаче HTML браузеру, а в выдаче данных в формате JSON или XML. Чтобы проверить, что наше приложение может это сделать, воспользуемся Postman или аналогичной утилитой для выполнения HTTP-запроса GET к корневому маршруту приложения с заголовком `Accept`, имеющим значение `application/json`. Этот тест с помощью Postman проиллюстрирован на рис. 1.12.

А вот и сам запрос:

```
Сервер, к которому выполняется запрос → GET / HTTP/1.1
Host: localhost:5000
Accept: application/json
```

HTTP-метод (GET), путь (/) и протокол HTTP/1.1, используемые в запросе

Список заголовков запроса. В данном случае это только заголовок `Accept` со значением `application/json`

Ответ на этот запрос — текущие дата и время по UTC, сериализованные в виде JSON:

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

"2016-06-06T19:50:09.2556094Z"

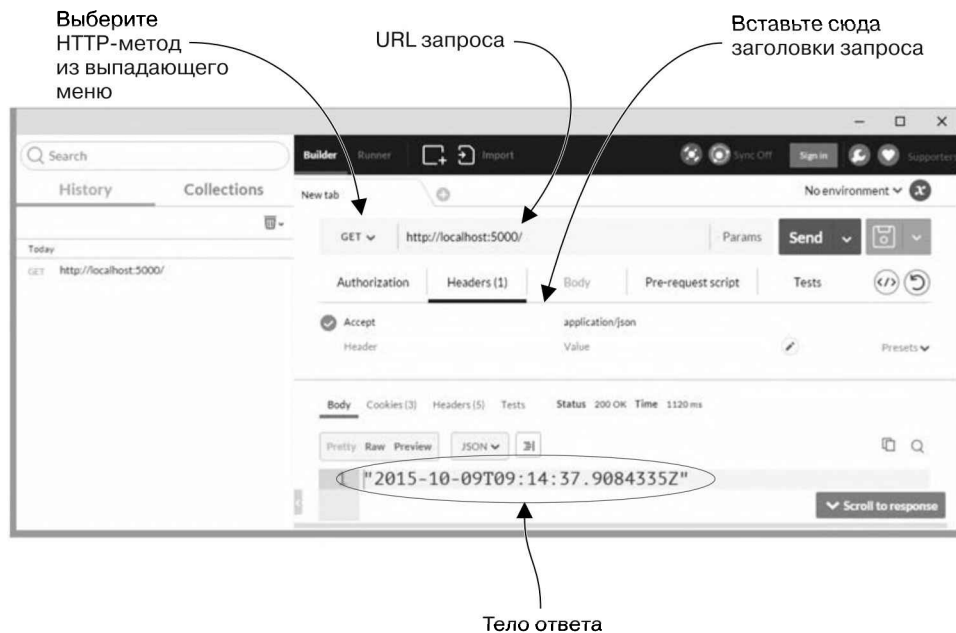


Рис. 1.12. Postman облегчает отправку HTTP-запросов и управление их нюансами, например заголовками и HTTP-методом

Если поменять заголовок `Accept` в предыдущем запросе на `application/xml`, ответ будет сериализован как XML. Фреймворк Nancy поддерживает как JSON-, так и XML-сериализацию и учитывает заголовок `Accept` при сериализации ответа. Мы реализовали нашу первую конечную точку API с помощью фреймворка Nancy.

Добавляем промежуточное ПО OWIN

Теперь, после создания и запуска минимального приложения Nancy, можно воспользоваться элементом промежуточного ПО OWIN для добавления простейшего журналирования запросов. В нашем приложении уже есть копвейер OWIN, хотя и очень короткий. Он содержит только один компонент — фреймворк Nancy. Код в листинге 1.4, который мы вставим в класс `Startup`, добавляет в этот копвейер компонент, который выводит в консоль текст `Got request` при каждом поступлении запроса. Затем он передает запрос следующему компоненту в конвейере — Nancy.

Листинг 1.4. Промежуточное ПО OWIN, выводящее в консоль строку

```

app.UseOwin(buildFunc =>
{
    buildFunc(next => env =>
    {
        System.Console.WriteLine("Got request");
        return next(env);
    });
    buildFunc.UseNancy();
});

```

Добавляем элемент промежуточного ПО (лямбда-выражение) в конвейер OWIN

Это промежуточное ПО выводит строку в стандартный поток вывода при каждом поступлении запроса

Вызываем следующее промежуточное ПО в конвейере OWIN и передаем ему все данные запроса

Если этот код кажется вам пемпого страпным, не беспокойтесь: я вернусь к OWIN в главах 9–11 и объясню, как он работает и как использовать его сильные стороны в микросервисах. Пока вам достаточно знать, что можно сформировать конвейер OWIN в классе *Startup* и что промежуточное ПО представляет собой лямбда-выражение:

```

next => env =>
{
    System.Console.WriteLine("Got request");
    return next(env);
}

```

С этим журналирующим запросы промежуточным ПО OWIN после запуска приложения и выполнения нескольких запросов по адресу <http://localhost:5000> вывод консоли нашего маленького микросервиса будет выглядеть следующим образом:

```

PS> dotnet run
Application started. Press Ctrl+C to shut down.
Got request
Got request
Got request
Got request
Got request
Got request

```

ПРИМЕЧАНИЕ

В этой книге мы будем использовать промежуточное ПО OWIN для сквозной функциональности, например для журналирования запросов и мониторинга.

На этом наш пример завершен. С помощью столь малого количества кода вам удалось создать свой первый простой микросервис с одной конечной точкой, возвращающей текущие дату и время по Гринвичу в формате JSON или XML. Более того, у этого микросервиса есть простейшее журналирование запросов в виде вывода текста в консоль при каждом поступлении запроса.

1.9. Резюме

- ❑ *Микросервисы* — многозначный термин, используемый как для микросервисного архитектурного стиля, так и для отдельных микросервисов в системе микросервисов.
- ❑ Микросервисный архитектурный стиль — особая разновидность SOA, при которой каждый сервис имеет небольшой размер и только одну бизнес-возможность.
- ❑ Микросервис — сервис с единовременной узконаправленной функциональной возможностью.
- ❑ Я буду часто ссылаться в этой книге на шесть отличительных признаков микросервисов, а именно:
 - микросервис отвечает за одну функциональную возможность;
 - микросервисы можно развертывать по отдельности. Каждый микросервис можно развернуть отдельно, не затрагивая никаких других частей системы;
 - микросервис выполняется в одном или нескольких процессах отдельно от других микросервисов;
 - относящиеся к обеспечиваемой микросервисом функциональной возможности данные принадлежат ему и содержатся в хранилище данных, к которому у него и только у него имеется доступ;
 - микросервис достаточно мал для того, чтобы небольшая команда разработчиков примерно из пяти человек могла сопровождать полдесятка или чуть больше микросервисов;
 - микросервис можно легко заменить. Команда разработчиков должна иметь возможность переписать микросервис с нуля за короткий период времени, если, скажем, база его кода стала слишком запутанной.
- ❑ Микросервисы идут рука об руку с непрерывной доставкой.
- ❑ Небольшой размер и индивидуальное развертывание микросервисов облегчают непрерывную доставку.
- ❑ Возможность автоматического, быстрого и надежного развертывания микросервисов упрощает развертывание и сопровождение системы микросервисов.
- ❑ Построенная на микросервисах система обеспечивает отказоустойчивость и возможность масштабирования.
- ❑ Построенная на микросервисах система пластична: ее можно легко изменить в соответствии с бизнес-потребностями. Каждый микросервис сам по себе в высокой степени легок в сопровождении, кроме того, можно быстро создать новые микросервисы для обеспечения новых возможностей.
- ❑ Микросервисы взаимодействуют между собой ради предоставления конечному пользователю пущей функциональности.
- ❑ Каждый микросервис предоставляет удаленный общедоступный API, который могут использовать другие микросервисы.

- ❑ Микросервис может предоставлять поток событий, на который могут подписываться другие микросервисы. События обрабатываются в микросервисах-подписчиках асинхронно, но оставляют подписчикам возможность быстрого реагирования на них.
- ❑ Nancy — облегченный фреймворк на платформе .NET, удобный для начала работы с микросервисами.
- ❑ Модули фреймворка Nancy используются для создания и настройки конечных точек в приложениях Nancy.
- ❑ Стандарт OWIN позволяет создавать конвейеры промежуточного программного обеспечения, выполняемые при каждом запросе и отлично подходящие для сквозной функциональности.
- ❑ Большинство микросервисов выдают в конечных точках не HTML, а данные в форматах JSON или XML. Для тестирования подобных конечных точек хорошо подходят такие приложения, как Postman и Fiddler.

2

Простой микросервис для корзины заказов

В этой главе:

- ❑ практически полнофункциональная реализация микросервиса Shopping Cart;
- ❑ создание конечных точек с помощью фреймворка Nancy;
- ❑ реализация запроса от одного микросервиса к другому;
- ❑ реализация с помощью фреймворка Nancy простой ленты событий для микросервиса.

В главе 1 мы изучили, как микросервисы работают и каковы их отличительные признаки. Мы также построили простой стек технологий C#/Nancy/OWIN, что дало возможность с легкостью создавать микросервисы, после чего взглянули на простой микросервис корзины заказов. В этой главе с помощью фреймворка Nancy реализуем четыре основные составные части этого микросервиса.

- ❑ Простейший API на основе протокола HTTP, с помощью которого клиенты смогут просматривать корзину заказов, удалять ее и добавлять в нее товары. Каждый из этих методов будет виден клиентам в качестве конечной точки HTTP, например `http://myservice/add/{item_number}`.
- ❑ Обращение одного микросервиса к другому за дополнительной информацией. В данном случае микросервис Shopping Cart будет обращаться к микросервису Product Catalog за информацией о ценах, исходя из `item_number` добавленного в корзину товара.
- ❑ Лента событий, посредством которой сервис будет публиковать события для остальной системы. Создание ленты событий для корзины заказов обеспечит другим сервисам (например, рекомендательному механизму) возможность обновления их собственных данных и совершенствования своих функциональных возможностей.
- ❑ Предметная логика для реализации поведения корзины заказов.

Ради простоты мы не будем полностью реализовывать этот микросервис в данной главе, а завершим его разработку в следующих частях книги. Рассмотрим следующие вопросы.

- ❑ У микросервиса Shopping Cart должно быть собственное хранилище данных, но мы не будем реализовывать его или код доступа к находящимся в нем данным. Этот вопрос рассмотрен в главе 5.
- ❑ Любой готовый для эксплуатации в производственной среде микросервис должен включать поддержку мониторинга и журналирования. Если микросервис

не обеспечивает постоянного доступа к информации о своем рабочем состоянии, становится сложнее поддерживать непрерывную работу системы в целом. Но эти функции не реализуют непосредственно бизнес-возможность, так что отложим обсуждение возможностей мониторинга и журналирования до главы 9.

Приступим.

ПРИМЕЧАНИЕ

Убедитесь, что вы полностью выполнили установку и настройку среды разработки. В приложении А вы найдете информацию по скачиванию, установке и — вкратце — использованию IDE, которые можно применять для работы с примерами кода из этой книги. В этой главе содержится масса кода, так что, если вы еще не установили среду разработки, самое время это сделать.

2.1. Обзор микросервиса Shopping Cart

В главе 1 мы видели, как сайт интернет-магазина, основанный на микросервисах, обрабатывает пользовательский запрос по добавлению товара в корзину заказов. Повторим полную картину обработки этого пользовательского запроса на рис. 2.1.

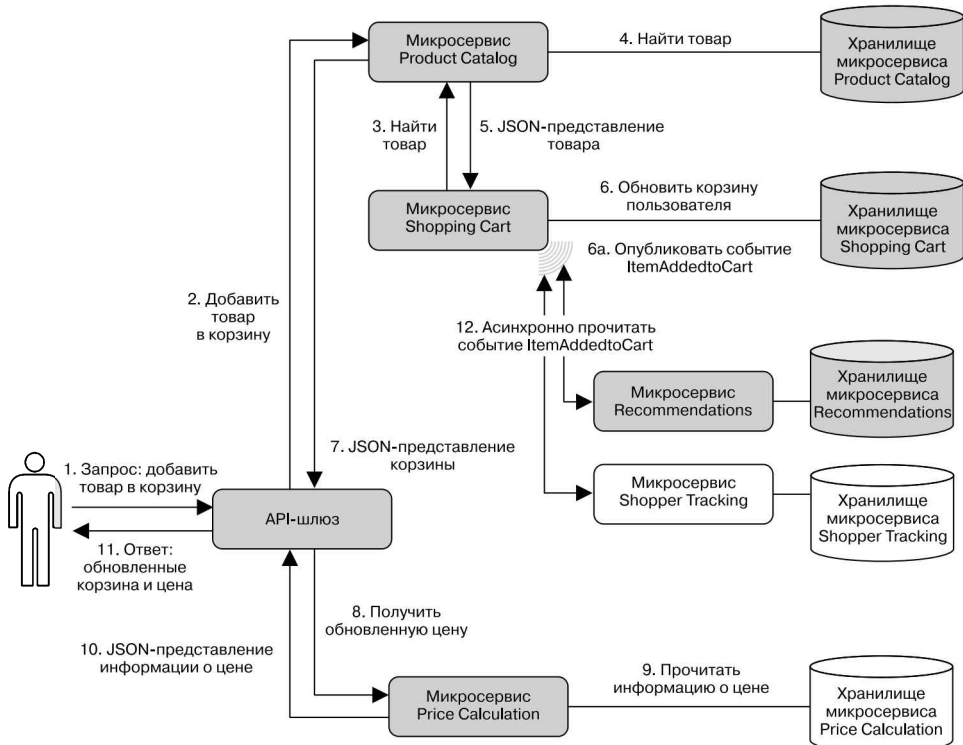


Рис. 2.1. Микросервис Shopping Cart обеспечивает возможности обращения других микросервисов к корзине заказов, добавления в нее товаров и их удаления и подписки на генерируемые микросервисом Shopping Cart события

Микросервис Shopping Cart играет основную роль при добавлении пользователем товара в корзину заказов. Но это не единственный процесс, в котором он участвует. Микросервис Shopping Cart не менее важен для просмотра корзины заказов и удаления из нее товаров. Shopping Cart должен поддерживать этот процесс посредством своего API для протокола HTTP аналогично тому, как он поддерживает добавление товара в корзину.

На рис. 2.2 показаны взаимодействия между микросервисом Shopping Cart и другими микросервисами системы.



Рис. 2.2. Обзор добавления товара в корзину заказов на сайте интернет-магазина, основанном на микросервисах

Микросервис Shopping Cart поддерживает три типа синхронных запросов:

- ☐ чтение корзины заказов;
- ☐ добавление товара в корзину заказов;
- ☐ удаление товара из корзины заказов.

Кроме этого, он предоставляет ленту событий, на которую могут подписываться другие микросервисы. Теперь, когда мы увидели общую картину функциональности микросервиса Shopping Cart, можно углубиться в его реализацию.

Компоненты микросервиса Shopping Cart

Рассмотрим, что представляет собой микросервис Shopping Cart. Далее перечислены его составные части (рис. 2.3).

- ☐ Небольшая модель предметной области микросервиса Shopping Cart, отвечающая за реализацию относящихся к корзинам заказов бизнес-правил.
- ☐ Компонент API для протокола HTTP, отвечающий за обработку всех входящих HTTP-запросов. Этот компонент состоит из двух модулей: один обрабатывает запросы от других микросервисов, а второй предоставляет ленту событий.
- ☐ Два компонента хранилища данных, EventStore и ShoppingCartStore. Они отвечают за взаимодействие с хранилищем данных (ShoppingCartStore):
 - EventStore занимается сохранением событий в хранилище данных и чтением их из него;

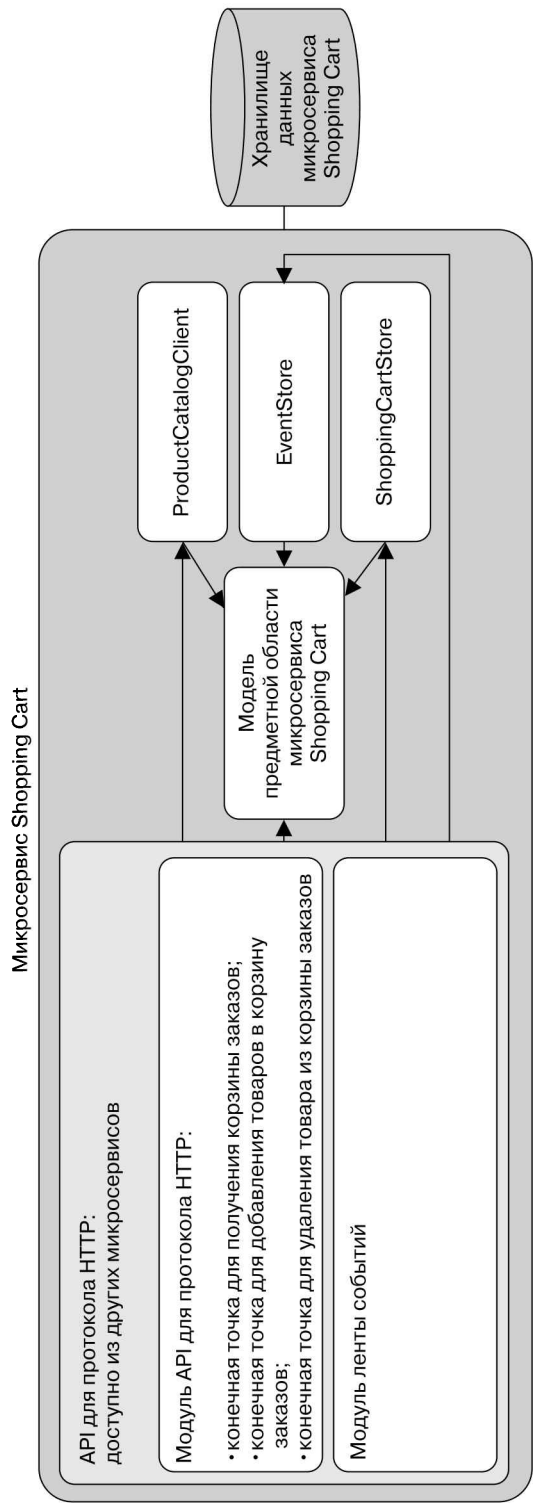


Рис. 2.3. Микросервис Shopping Cart — небольшая база кода из нескольких компонентов, обеспечивающая одну конкретную бизнес-возможность

- `ShoppingCartStore` занимается чтением и обновлением корзины заказов в хранилище данных. Замечу, что корзины заказов и события могут храниться в разных базах данных (мы вернемся к этому вопросу в главе 5).
- Компонент `ProductCatalogClient` отвечает за взаимодействие с микросервисом `Product Catalog`, показанным на рис. 2.1. Размещение кода этого взаимодействия в компоненте `ProductCatalogClient` служит нескольким целям:
- информация об API других микросервисов инкапсулируется в одном месте;
 - в этом же месте инкапсулируются детали выполнения HTTP-запроса;
 - в этом же месте инкапсулируется кэширование результатов обращения к другим микросервисам;
 - в этом же месте инкапсулируется обработка ошибок, возвращаемых другими микросервисами.

В этой главе приведены код для модели предметной области, API для протокола HTTP и простейшая реализация компонента `ProductCatalogClient`, но здесь нет кода ни `EventStore`, ни `ShoppingCartStore`, ни хранилища данных. Кроме того, ради краткости опущен код обработки ошибок. В главах 4 и 5 мы подробнее рассмотрим реализацию API микросервисов с помощью фреймворка Nancy. В главе 5 вернемся к вопросу хранения данных в микросервисе. Глава 6 будет посвящена более детальному изучению обеспечения надежности таких клиентов, как `ProductCatalogClient`.

2.2. Реализация микросервиса Shopping Cart

Теперь, разобравшись с компонентами микросервиса Shopping Cart, мы можем заняться его кодом.

НОВЫЕ ТЕХНОЛОГИИ

В этой главе мы начнем использовать две новые технологии.

- `HttpClient` — встроенный тип платформы .NET Core, предназначенный для выполнения HTTP-запросов. Он предоставляет API для создания и отправки HTTP-запросов, а также чтения возвращаемых ответов.
- `Polly` — библиотека, облегчающая реализацию одной из наиболее распространенных стратегий обработки происходящих при удаленных вызовах сбоев. Она прямо «из коробки» поддерживает различные стратегии повторов отправки (retry) и предохранителей (circuit breaker). Мы обсудим предохранители в главе 6.

Создание пустого проекта

Первое, что необходимо сделать, — создать проект Nancy точно так же, как в главе 1. Создайте пустое приложение ASP.NET Core под названием `ShoppingCart` и добавьте в этот проект пакет Nancy системы управления пакетами NuGet. Далее добавьте Nancy в приложение в классе `Startup` (листинг 2.1).

Листинг 2.1. Класс Startup, запускающий Nancy

```
namespace ShoppingCart
{
    using Microsoft.AspNet.Builder;
    using Nancy.Owin;

    public class Startup
    {
        public void Configure(IApplicationBuilder app)
        {
            app.UseOwin(buildFunc => buildFunc.UseNancy());
        }
    }
}
```

В файл Startup.cs
нужно добавить только
одну эту строку



Теперь у нас есть пустое приложение Nancy, готовое к работе.

API, предоставляемый микросервисом Shopping Cart другим сервисам

В этом разделе мы реализуем API для протокола HTTP микросервиса Shopping Cart, выделенное на рис. 2.4. Этот API состоит из трех частей, каждая из которых реализована в виде конечной точки HTTP.

- ❑ Конечная точка HTTP GET, в которой остальные микросервисы могут получить корзину заказов пользователя путем указания его ID. Ответ представляет собой корзину заказов, сериализованную в формат JSON или XML.
- ❑ Конечная точка HTTP POST, в которой остальные микросервисы могут добавлять в корзину заказов товары. Добавляемые товары передаются в конечную точку в виде массива идентификаторов товаров. Этот массив может быть в формате XML или JSON и должен являться телом запроса.
- ❑ Конечная точка HTTP DELETE, в которой остальные микросервисы могут удалять товары из корзины заказов пользователя. Удаляемые товары передаются в теле запроса в виде XML- или JSON-массива идентификаторов товаров.

В каждом из следующих трех разделов мы реализуем по одной из этих конечных точек.

Получение корзины заказов

Первая часть API для протокола HTTP, которую мы реализуем, — конечная точка, предоставляющая другим микросервисам возможность извлечения пользовательской корзины заказов. На рис. 2.5 показано применение этой конечной точки другими микросервисами для получения корзины заказов.

Эта конечная точка принимает запрос типа HTTP GET. Ее URL включает идентификатор пользователя, чья корзина заказов необходима другому микросервису, а тело ответа представляет собой XML- или JSON-сериализацию этой корзины заказов. Запрос должен включать заголовок *Accept*, который определяет, в каком формате, XML или JSON, должно быть тело ответа.

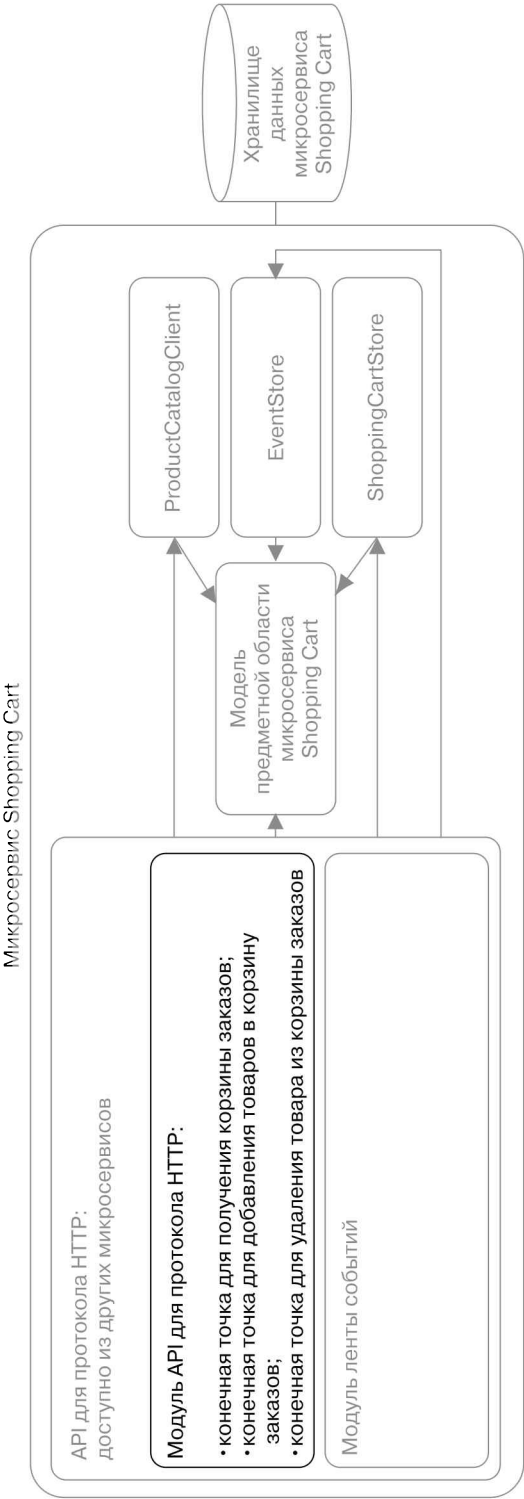


Рис. 2.4. Реализация компонента API для протокола HTTP

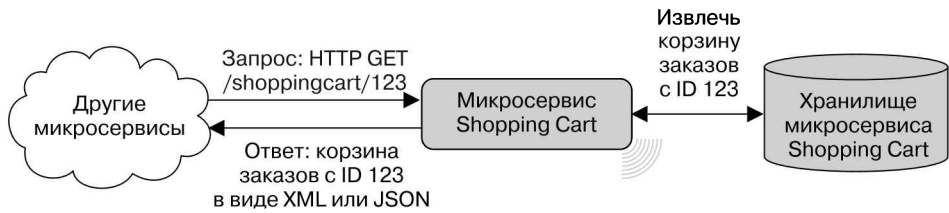


Рис. 2.5. Другие микросервисы могут использовать конечную точку микросервиса Shopping Cart для получения корзины заказов в формате XML или JSON

Например, пусть шлюзу API с рис. 2.1 необходима корзина заказов для пользователя с ID 123. Для ее получения шлюз посылает HTTP-запрос:

```

HTTP GET /shoppingcart/123 HTTP/1.1
Host: shoppingcart.my.company.com
Accept: application/json
  
```

Это запрос к URL `shoppingcart/123` микросервиса Shopping Cart, в котором часть URL `123` представляет собой идентификатор пользователя.

Для обработки подобных запросов необходимо добавить в проект ShoppingCart новый модуль Nancy под названием `ShoppingCartModule`. Как упоминалось в главе 1, модули фреймворка Nancy представляют собой классы — потомки класса `NancyModule` и используются для реализации конечных точек в приложениях Nancy. Поместите следующий код (листинг 2.2) в новый файл с названием `ShoppingCartModule.cs`.

Листинг 2.2. Конечная точка для доступа к корзине заказов по ID пользователя

```

Сообщаем Nancy, что все маршруты
в этом модуле будут начинаться с /shoppingcart

namespace ShoppingCart.ShoppingCart
{
    using Nancy;
    using Nancy.ModelBinding;

    public class ShoppingCartModule : NancyModule
    {
        public ShoppingCartModule(IShoppingCartStore shoppingCartStore)
        : base("/shoppingcart")
        {
            Get("/{userid:int}", parameters =>
            {
                var userId = (int) parameters.userid;
                return shoppingCartStore.Get(userId);
            });
        }
    }
}

Объявляем класс
ShoppingCartModule
как потомок класса
NancyModule.
Фреймворк Nancy
автоматически находит
все модули Nancy
при загрузке

Объявляем конечную точку
для обработки запросов
к URL /shoppingcart/{userid},
например /shoppingcart/123

Возвращаем корзину заказов пользователя.
Фреймворк Nancy сериализует его в формат XML
или JSON перед отправкой клиенту

Устанавливаем идентификатор пользователя
равным сегменту userid URL запроса
  
```

Наверное, вы заметили здесь некоторые важные составные части фреймворка Nancy в действии. Разобьем этот код на составные части.

Выражение `Get("/{userid:int}", ...?)` представляет собой объявление маршрута, таким образом можно объявить о необходимости обработки HTTP-запросов типа `GET` к конечным точкам, соответствующим указанному в скобках шаблону. Шаблон может быть строковым литералом, например `"/shoppingcart"`, или содержать сегменты, соответствующие определенным частям URL запроса, например `{userid:int}`. Сегмент `{userid:int}` называется `userid` и соответствует только целочисленным значениям.

За объявлением маршрута следует лямбда-выражение:

```
parameters =>
{
    var userId = (int) parameters.userid;
    return shoppingCartStore.Get(userId);
};
```

Это обработчик маршрута, представляющий собой фрагмент кода, выполняемый всякий раз при получении микросервисом Shopping Cart запроса к URL, соответствующему объявлению маршрута. Например, когда шлюз API запрашивает корзину заказов через URL `shoppingcart/123`, именно этот код обрабатывает запрос.

Обработчик запроса принимает на входе один аргумент, `parameters`, обеспечивающий доступ ко всем сегментам URL запроса, которым нашлось соответствие. Объект `parameters` — динамический, и к этим сегментам можно обращаться так, как будто они являются его свойствами. Именно поэтому выражение `parameters.userid` работает, но тип `parameters.userid` — `dynamic`, так что перед применением приходится преобразовывать его в `int`.

Обработчик маршрута использует также объект `shoppingCartStore`, принимаемый в качестве аргумента конструктором класса `ShoppingCartModule`:

```
public ShoppingCartModule(IShoppingCartStore shoppingCartStore)
```

Поскольку обработка маршрута происходит в модуле `ShoppingCartModule`, аргумент `shoppingCartStore` находится в области видимости обработчика маршрута.

Тип аргумента конструктора — интерфейс `IShoppingCartStore`. Фреймворк Nancy автоматически отыщет реализацию интерфейса `IShoppingCartStore` и, если не обнаружит неоднозначности, подставит вместо него экземпляр `ShoppingCartStore`. Я не стану включать код хранилища данных в эту главу, но прилагаемый к книге код содержит интерфейс `IShoppingCartStore` и пустую ее реализацию.

Обработчик маршрута возвращает объект `ShoppingCart`, полученный им от объекта `shoppingCartStore`:

```
return shoppingCartStore.Get(userId);
```

Тип `ShoppingCart` создан специально для микросервиса Shopping Cart, так что фреймворк Nancy не может ничего о нем знать. Но Nancy терпимо отнесется к возвращаемым от обработчиков маршрутов данным. Можно вернуть любой пугный объект, и Nancy обработает такую ситуацию вполне адекватно. В этом случае мы хотели бы сериализовать объект `ShoppingCart` и вернуть данные вызывающей стороне. Именно это Nancy и делает.

В листинге 2.3 показан пример ответа на запрос к URL `shoppingcart/123`.

Листинг 2.3. Пример ответа от микросервиса Shopping Cart

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8 ← Тело ответа в формате JSON

```
539      ← Длина тела ответа
{
  "userId": 42,
  "items": [
    {
      "productcatalogId": 1,
      "productName": "Basic t-shirt",
      "description": "a quiet t-shirt",
      "price": {
        "currency": "eur",
        "amount": 40
      }
    },
    {
      "productcatalogId": 2,
      "productName": "Fancy shirt",
      "description": "a loud t-shirt",
      "price": {
        "currency": "eur",
        "amount": 50
      }
    }
  ]
}
```

Корзина заказов, сериализованная в формат JSON

Как видите, благодаря фреймворку Nancу с помощью небольшого объема кода можно создать значительную функциональность и добиться ее работоспособности.

Добавление товаров в корзину заказов

Вторая конечная точка, которую нам нужно включить в состав микросервиса Shopping Cart, служит для добавления товаров в корзину заказов пользователя. На рис. 2.6 показано применение этой конечной точки другими микросервисами.



Рис. 2.6. Другие микросервисы могут добавлять товары в корзину заказов с помощью HTTP-запроса POST, содержащего в теле массив идентификаторов товаров

Аналогично конечной точке HTTP GET из предыдущего раздела эта новая конечная точка получает в URL ID пользователя. На этот раз конечная точка принимает HTTP-запросы типа POST вместо HTTP-запросов GET и запрос должен содержать в теле список товаров. Например, следующий запрос добавляет два товара в корзину заказов пользователя 123 (листинг 2.4).

Листинг 2.4. Добавление двух товаров в корзину заказов

Данные в теле запроса должны быть в формате JSON

POST /shoppingcart/123/items HTTP/1.1

Host: shoppingcart.my.company.com

Accept: application/json

Content-Type: application/json

URL включает ID корзины заказов: 123

Формат ответа должен быть JSON

[1, 2]

Тело запроса представляет собой JSON-массив идентификаторов товаров

Для обработки подобных запросов нам понадобится добавить еще одно объявление маршрута в класс `ShoppingCartModule`. Новый обработчик маршрута будет читать товары из тела запроса, находить для каждого из них информацию о товаре, добавлять их в соответствующую корзину заказов и возвращать обновленную корзину.

Объявление нового маршрута показано в листинге 2.5. Добавьте его в конструктор класса `ShoppingCartModule`.

Листинг 2.5. Обработчик для маршрута добавления товаров в корзину заказов

```
public class ShoppingCartModule : NancyModule
{
    public ShoppingCartModule(
        IShoppingCartStore shoppingCartStore,
        IProductcatalogClient productcatalog,
        IEventStore eventStore)
        : base("/shoppingcart")
    {
        Get("/{userid:int}", parameters => { ... });

        Post("/{userid:int}/items",
            async (parameters, _) =>
        {
            var productcatalogIds = this.Bind<int[]>();
            var userId = (int) parameters.userid;

            var shoppingCart = shoppingCartStore.Get(userId);
            var shoppingCartItems = await
                productcatalog
                    .GetShoppingCartItems(productcatalogIds)
                    .ConfigureAwait(false);
```

Объявляем конечную точку HTTP POST для URL /shoppingcart/{userid}/item

Читаем и десериализуем массив идентификаторов товаров из тела HTTP-запроса

Извлекаем информацию о товарах из микросервиса ProductCatalog

```

shoppingCart.AddItems(shoppingCartItems, eventStore);
shoppingCartStore.Save(shoppingCart);
return shoppingCart;
});
}
}

```

Добавляем товары в корзину

Сохраняем обновленную корзину в хранилище данных

Возвращаем обновленную корзину

Здесь вступают в игру две новые возможности Nancy. Во-первых, новый обработчик событий асинхронен. Это можно видеть по объявлению лямбда-выражения `async`:

```

Post("/{userid:int}/items",
    async (parameters, _) =>

```

Этот обработчик объявлен как асинхронный потому, что выполняет удаленное обращение к микросервису Product Catalog. Выношение этого внешнего вызова асинхронно экономит ресурсы в Shopping Cart. Nancy может работать полностью асинхронно, что дает возможность коду приложений эффективно использовать ключевые слова `async/await` языка программирования C#.

Во-вторых, тело запроса содержит JSON-массив идентификаторов товаров, которые необходимо добавить в корзину заказов. Обработчик события использует привязку моделей фреймворка Nancy для их чтения в C#-массив:

```

var productcatalogIds = this.Bind<int[]>();

```

Привязка моделей Nancy поддерживает все сериализуемые объекты C#. Зачастую лучше применять более структурированный объект, чем «нлоский» JSON-массив для отправки данных в конечную точку, причем чтение в этом случае тоже упрощается. При этом достаточно будет просто менять тип `int[]` в параметре типа из выражения `this.Bind<int[]>()` на другой. «Из коробки» фреймворк Nancy поддерживает привязку к данным в форматах JSON и XML, но, как мы увидим в главе 4, добавление других форматов также не представляет сложностей.

Новый обработчик маршрута использует два объекта, нока отсутствующих в `ShoppingCartModule`. Мы снова наложимся на фреймворк Nancy и передадим их через аргументы конструктора. Механизм внедрения зависимостей фреймворка Nancy автоматически обеспечивает зависимости при создании экземпляров модулей (листинг 2.6).

Листинг 2.6. Добавление зависимостей модуля в качестве аргументов конструктора

```

public ShoppingCartModule(
    IShoppingCartStore shoppingCartStore,
    IProductcatalogClient productCatalog,
    IEventStore eventStore)

```

Применяется только для передачи значения в метод `AddItems`, где будет использовано позднее

Теперь остальные микросервисы тоже могут добавлять товары в корзину заказов. Соответственно, им должна быть предоставлена возможность удаления товаров из корзины заказов.

Краткий обзор возможностей ключевых слов `async/await`

В версии 5 языка C# появились два новых ключевых слова, `async` и `await`, предназначенные для упрощения асинхронного выполнения методов. Простейший метод `async` выглядит следующим образом:

```

public async Task<int> WaitForANumber()
{
    await Task.Delay(1000);
    .ConfigureAwait(false);
    return 10;
}

```

Объявляет метод как асинхронный

Откладывает выполнение текущего потока до завершения задания

Разрешает возобновление выполнения с другим контекстом потока

Поскольку этот метод — асинхронный, возвращаемое значение автоматически оборачивается в объект Task

При вызове этого метода поток выполнения идет обычным образом до момента, указанного в `await`. Ключевое слово `await` используется совместно с *ожидаемыми типами* (*awaitables*), самый распространенный из которых — `System.Threading.Tasks.Task<T>`, при этом происходит асинхронное ожидание завершения ожидаемого метода. Это значит, что, когда поток выполнения достигает `await`, происходят две вещи.

- Оставшаяся часть метода отправляется в очередь на выполнение после того, как ожидаемый метод будет выполнен, — в данном случае поток выполнения вернется из метода `Task.Delay(1000)`. После завершения ожидаемого метода выполняется оставшаяся часть «наружного» метода, вероятно, в новом потоке, но с тем же состоянием, которое было до `await`.
- Текущий поток выполнения возвращается из асинхронного метода и продолжает выполняться в вызывающей программе.

Вызов `ConfigureAwait(false)` в предыдущем фрагменте кода указывает объекту `Task`, что сохранять текущий контекст потока не требуется. Вследствие этого контекст потока не восстанавливается при продолжении выполнения метода. А раз методу не нужен контекст потока, его сохранение и восстановление можно опустить.

В серверном коде, как и в микросервисах, для выполнения многих запросов требуются операции ввода/вывода (I/O), такие как обращение к хранилищу данных или другому микросервису. Возможность асинхронного выполнения операций ввода/вывода вместо блокирования потока на время ожидания их завершения позволяет экономить серверные ресурсы. В этой книге мы будем часто использовать ключевые слова `async/await` и объекты `Task` ради экономии серверных ресурсов и лучшей масштабируемости.

Удаление товаров из корзины заказов

Третья, и последняя, конечная точка HTTP DELETE обеспечивает остальным микросервисам возможность удаления товаров из корзины заказов (рис. 2.7). К настоящему времени вы уже должны неплохо представлять себе процесс добавления конечных точек в модули фреймворка Nancy. Вам необходимо реализовать конечную точку HTTP DELETE, которая получала бы на входе массив идентификаторов товаров и удаляла эти товары из корзины.

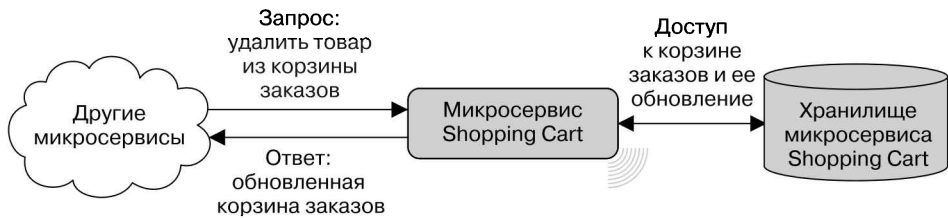


Рис. 2.7. Другие микросервисы могут удалять товары из корзины заказов с помощью HTTP-запроса DELETE путем передачи массива идентификаторов товаров в теле запроса

Добавьте следующие строки в конструктор класса `ShoppingCartModule` (листинг 2.7).

Листинг 2.7. Конечная точка для удаления товаров из корзины заказов

```
public class ShoppingCartModule : NancyModule
{
    public ShoppingCartModule(
        IShoppingCartStore shoppingCartStore,
        IProductCatalogClient productCatalog,
        IEventStore eventStore)
        : base("/shoppingcart")
    {
        Get("/{userid:int}", parameters => { ... });

        Post("/{userid:int}/items",
            async (parameters, _) => { ... });

        Delete("/{userid:int}/items", parameters =>
        {
            var productCatalogIds = this.Bind<int[]>();
            var userId = (int)parameters.userid;

            var shoppingCart = shoppingCartStore.Get(userId);
            shoppingCart.RemoveItems(productCatalogIds, eventStore);
            shoppingCartStore.Save(shoppingCart);

            return shoppingCart;
        });
    }
}
```

Вполне допустимо использовать один шаблон маршрута для двух объявлений маршрутов, если они используют различные HTTP-методы

Объект `eventStore` будет далее использоваться в методе `RemoveItems`

На этом создание `ShoppingCartModule`, потребовавшее менее 50 строк кода, завершено. Именно поэтому я считаю Nancy облегченным фреймворком.

Извлечение информации о товаре

Теперь, когда мы реализовали предоставляемый микросервисом Shopping Cart API, перейдем к другому вопросу и рассмотрим получение информации о товарах из микросервиса Product Catalog. Компонент `ProductCatalogClient`, который мы собираемся реализовать в этом разделе, на рис. 2.8 выделен.

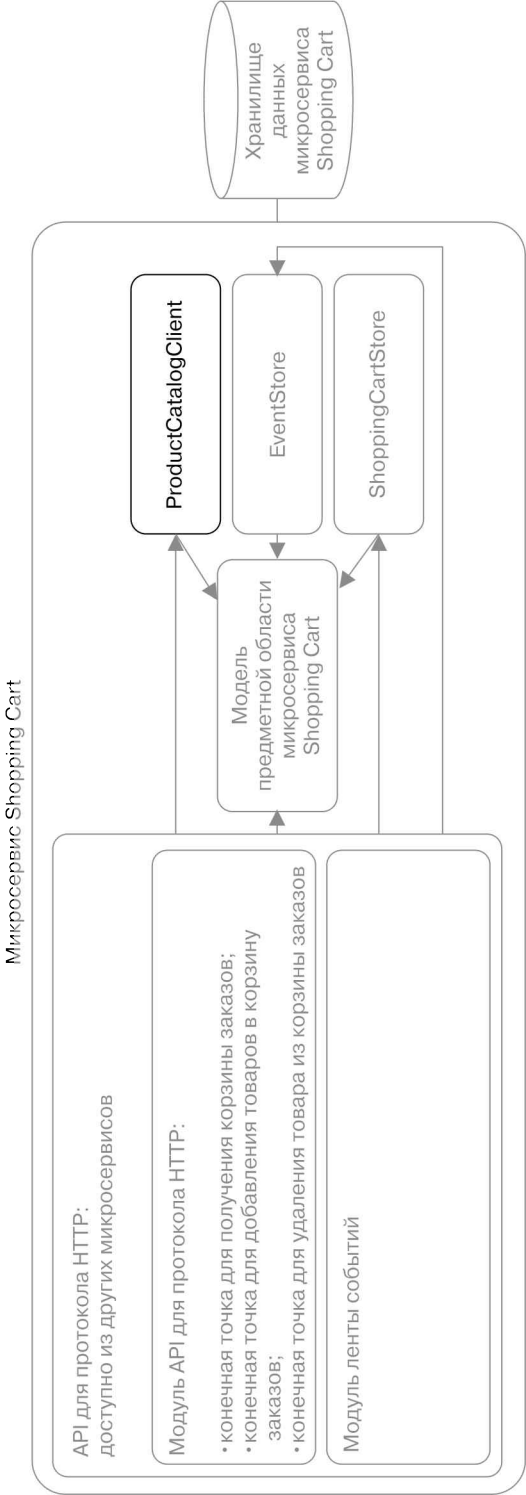


Рис. 2.8. Компонент ProductCatalogClient

Product Catalog и Shopping Cart — разные микросервисы, работающие в отдельных процессах, возможно даже на отдельных серверах. Микросервис Product Catalog предоставляет API для протокола HTTP, который использует микросервис Shopping Cart. Информацию о каталоге товаров можно получить путем HTTP-запроса GET к конечной точке микросервиса Product Catalog.

Для реализации HTTP-запроса к микросервису Product Catalog необходимо:

- ❑ реализовать HTTP-запрос GET;
- ❑ выполнить синтаксический разбор ответа конечной точки микросервиса Product Catalog и адантировать его к предметной области микросервиса Shopping Cart;
- ❑ реализовать стратегию обработки неудачных запросов к микросервису Product Catalog.

В последующих разделах рассмотрим выполнение этих пунктов подробнее.

Реализация HTTP-запроса типа GET

Микросервис Product Catalog предоставляет конечную точку, находящуюся по пути `/products`. Она принимает на входе массив идентификаторов товаров в виде параметра строки запроса и возвращает информацию о товаре для каждого из них. Например, следующий запрос служит для извлечения информации о товарах с ID 1 и 2:

```
HTTP GET /products?productIds=[1,2] HTTP/1.1
```

Host: productcatalog.my.company.com

Accept: application/json

Для выполнения запроса воспользуемся типом `HttpClient`. Вместо настоящего микросервиса `Product Catalog` эта реализация использует созданный с помощью сервиса *Apiary* (<https://apiary.io>). *Apiary* — онлайн-сервис, который, помимо прочего, предоставляет удобную возможность создания конечных точек, возвращающих жестко зашитые ответы. Следующий код (листинг 2.8) с помощью подобной фиктивной конечной точки выполняет HTTP-запрос типа `GET` к микросервису `Product Catalog`.

Листинг 2.8. HTTP-запрос типа GET к микросервису Product Catalog

URL фиктивного микросервиса Product Catalog

```
private static string productCatalogBaseUrl =
    @"http://private-05cc8-chapter2productcatalogmicroservice
    ➡ .apiary-mock.com";
private static string getProductPathTemplate =
    "/products?productIds={0}";
private static async Task<HttpResponseMessage>
    RequestProductFromProductCatalogue(int[] productCatalogueIds)
{
    var productsResource = string.Format(
        getProductPathTemplate, string.Join(",", productCatalogueIds));
```

```

using (var httpClient = new HttpClient())
{
    httpClient.BaseAddress = new Uri(productCatalogueBaseUrl);
    return await
        httpClient.GetAsync(productsResource).ConfigureAwait(false);
}

```

Указываем httpClient,
что HTTP-запрос GET
необходимо выполнять асинхронно

Создаем клиент
для выполнения
HTTP-запроса типа GET

Все очень просто. Единственное, что не помешает отметить: благодаря асинхронному выполнению HTTP-запроса GET текущий поток освобождается для обработки во время обработки запроса в Product Catalog, других заданий — в микросервисе Shopping Cart. Это хорошая практика, позволяющая экономить ресурсы в микросервисе Shopping Cart, благодаря чему он становится немного менее требовательным к ресурсам и лучше масштабируемым.

Синтаксический разбор ответа с информацией о товаре

Микросервис Product Catalog возвращает информацию о товаре в виде JSON-массива. В массиве имеется запись для каждого из запрошенных товаров, как показано в листинге 2.9.

Листинг 2.9. Возвращаемый список товаров в формате JSON

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

543

```

[
  {
    "productId": "1",
    "productName": "Basic t-shirt",
    "productDescription": "a quiet t-shirt",
    "price": { "amount" : 40, "currency": "eur" },
    "attributes" : [
      {
        "sizes": [ "s", "m", "l"],
        "colors": ["red", "blue", "green"]
      }
    ]
  },
  {
    "productId": "2",
    "productName": "Fancy shirt",
    "productDescription": "a loud t-shirt",
    "price": { "amount" : 50, "currency": "eur" },
    "attributes" : [
      {

```

```

        "sizes": [ "s", "m", "l", "xl"],
        "colors": ["ALL", "Batique"]
    }
}
]

```

Необходимо десериализовать этот JSON и прочесть из него информацию, необходимую для создания списка элементов типа `ShoppingCart`. API микросервиса `Product Catalog` форматирует возвращаемый им массив. Избежать тесной связанности между микросервисами помогает то, что только компонент `ProductCatalogClient` знает что-либо про API микросервиса `Product Catalog`. Это значит, что `ProductCatalogClient` несет ответственность за преобразование полученных от микросервиса данных в подходящие для проекта `ShoppingCart` типы. В данном случае нам нужен список объектов типа `ShoppingCartItem`. В листинге 2.10 приведен код для десериализации и преобразования данных ответа.

Листинг 2.10. Извлечение данных из ответа

```

                                Используем фреймворк Json.NET для десериализации
                                поступивших от микросервиса Product
                                Catalog данных в формате JSON
private static async Task<IEnumerable<ShoppingCartItem>>
    ConvertToShoppingCartItems(HttpResponseMessage response)
{
    response.EnsureSuccessStatusCode();
    var products =
        JsonConvert.DeserializeObject<List<ProductCatalogueProduct>>(
            await response.Content.ReadAsStringAsync().ConfigureAwait(false));
    return
        products
            .Select(p => new ShoppingCartItem(
                int.Parse(p.ProductId),
                p.ProductName,
                p.ProductDescription,
                p.Price
            ));
}

                                Создаем экземпляр
                                класса ShoppingCartItem
                                для каждого товара в ответе

                                Используем закрытый
                                класс для представления
                                данных по товарам
private class ProductCatalogProduct
{
    public string ProductId { get; set; }
    public string ProductName { get; set; }
    public string ProductDescription { get; set; }
    public Money Price { get; set; }
}

```

Если вы сравните листинги 2.9 и 2.10, то можете заметить, что в ответе больше свойств, чем было в классе `ProductCatalogProduct`. Дело в том, что микросервису `Shopping Cart` не требуется вся информация, так что нет причин читать остальные свойства, это только привело бы к излишней связанности. Мы вернемся к этому вопросу в главах 4, 5 и 7.

Листинг 2.11 сочетает в себе код запроса информации о товаре и код синтаксического разбора ответа. Этот метод выполняет HTTP-запрос GET и адаптирует ответ к предметной области микросервиса Shopping Cart.

Листинг 2.11. Извлечение товаров и преобразование их в элементы корзины заказов

```
private async Task<IEnumerable<ShoppingCartItem>>
    GetItemsFromCatalogueService(int[] productCatalogueIds)
{
    var response = await
        RequestProductFromProductCatalogue(productCatalogueIds)
        .ConfigureAwait(false);
    return await ConvertToShoppingCartItems(response)
        .ConfigureAwait(false);
}
```

Компонент `ProductCatalogClient` практически закончен. Осталось только добавить код для обработки сбоев HTTP-запросов.

Добавление стратегии обработки ошибок

Удаленные вызовы *могут* завершаться неудачно. И не только могут, но при работе большой распределенной системы они зачастую завершаются неудачно. Вероятно, обращение микросервиса Shopping Cart к микросервису Product Catalog не будет завершаться неудачно часто, но в целой системе микросервисов неудачные завершения удаленных вызовов не будут редкостью.

Удаленные вызовы завершаются неудачно по многим причинам: сбой сети, плохо сформированный вызов, ошибка в удаленном микросервисе, сбой сервера, на котором происходит обработка вызова, или удаленный микросервис находится в процессе развертывания новой версии. В системе микросервисов следует ожидать возможных сбоев и заранее проектировать отказоустойчивость выполнения всех удаленных вызовов. Это важная тема, к которой мы вернемся в главе 6.

Требующийся при конкретном удаленном вызове уровень отказоустойчивости зависит от бизнес-требований к выполняющему вызов микросервису. Обращение микросервиса Shopping Cart к микросервису Product Catalog играет важную роль: без информации о товарах пользователь не сможет добавить товары в свою корзину заказов, а значит, интернет-магазин не сможет продать пользователю этот товар. В то же время информация о товарах меняется редко, так что ее можно кэшировать в микросервисе Product Catalog и запрашивать из него только тогда, когда в кэше не окажется нужной информации. Кэширование — это хорошая стратегия, поскольку:

- ❑ обеспечивает устойчивость микросервиса Shopping Cart к сбоям микросервиса Product Catalog;
- ❑ микросервис Shopping Cart будет работать эффективнее при наличии в кэше информации о товарах;
- ❑ меньшее количество запросов от микросервиса Shopping Cart означает меньшую нагрузку на микросервис Product Catalog.

Мы пока еще не реализовали кэширование. К реализации кэширования для большей надежности вернемся в главе 6.

Некоторые обращения микросервиса Shopping Cart к микросервису Product Catalog выполняются даже в случае кэширования. В подобных случаях оптимальной стратегией обработки неудачных обращений представляется следующее: повторить обращение несколько раз, после чего отказаться от дальнейших попыток и признать действие по добавлению товаров в корзину заказов завершившимся неудачно. В этой главе мы реализуем простую стратегию повторной отправки для обработки случаев неудавшихся запросов. Мы воспользуемся библиотекой Polly, которую включим в проект Shopping Cart в виде пакета NuGet.

ПРИМЕЧАНИЕ

Более подробно библиотека Polly и стратегии обработки ошибок описываются в главе 6.

Использование стратегии Polly включает два шага.

1. Объявление стратегии.
2. Применение стратегии для выполнения удаленного вызова.

Как можно видеть из листинга 2.12, API библиотеки Polly сильно упрощает оба шага.

Листинг 2.12. Стратегия обработки ошибок микросервиса

```
private static Policy exponentialRetryPolicy =  
    Policy.  
        .Handle<Exception>()  
        .WaitAndRetryAsync(  
            3,  
            attempt => TimeSpan.FromMilliseconds(100 * Math.Pow(2, attempt)));  
  
public async Task<IEnumerable<ShoppingCartItem>>  
    GetShoppingCartItems(int[] productCatalogIds) =>  
        exponentialRetryPolicy  
            .ExecuteAsync(async () =>  
                await GetItemsFromcatalogService(productCatalogIds) ←  
                    .ConfigureAwait(false));
```

Используем API библиотеки Polly
для создания стратегии повторной отправки
с экспоненциальной отсрочкой отправки

Оборачиваем обращения
к микросервису Product Catalog
в стратегию повторной отправки

Стратегия, обернутая вокруг обращения к микросервису Product Catalog, проста: в случае неудачи обращение повторяется минимум три раза. При каждой неудаче время ожидания перед следующей попыткой удваивается.

На этом реализация компонента ProductCatalogClient завершена. Хотя он и насчитывает менее 100 строк кода, объем выполняемой им работы довольно велик: он формирует HTTP-запрос GET и выполняет его. Он осуществляет синтаксический разбор ответа, полученного от микросервиса Product Catalog, и адаптирует ответ к предметной области корзины заказов. Он также включает используемую для этих вызовов стратегию повторной отправки. Далее займемся лентой событий.

Реализация простейшей ленты событий

Микросервис Shopping Cart уже умеет хранить корзины заказов и добавлять в них товары. Описанием такого товара служит информация, полученная от микросервиса Product Catalog. Микросервис Shopping Cart также предоставляет другим микросервисам API, обеспечивающий возможность добавления товаров в корзины заказов и удаления из них, а также чтения содержимого корзины.

Пока нам не хватает ленты событий. Микросервису Shopping Cart необходимо публиковать события об изменениях в корзинах заказов, чтобы другие микросервисы могли подписываться на эти события и при необходимости реагировать на них. Скопированный из главы 1 рис. 2.9 иллюстрирует, что в случае добавления товаров в корзину заказов часть функциональности микросервисов Recommendations и Shopper Tracking основана на поступающих от микросервиса Shopping Cart событиях.

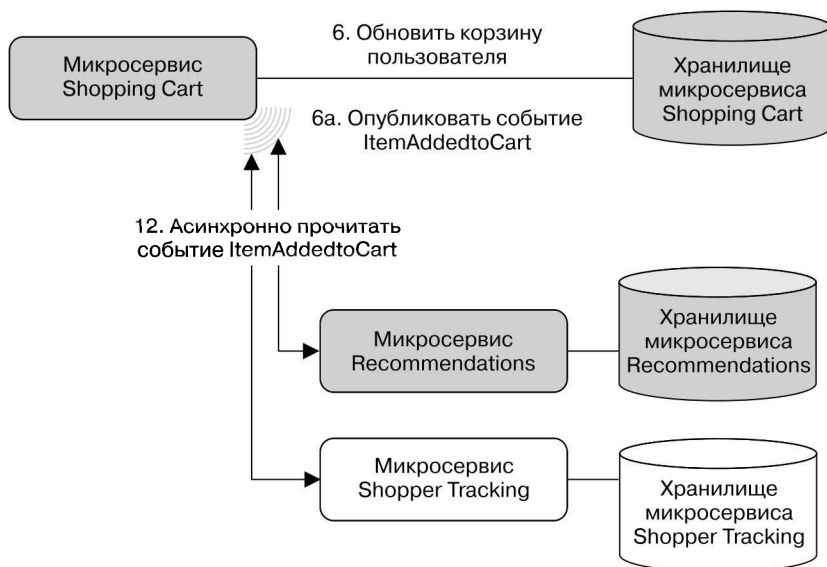


Рис. 2.9. Микросервис Shopping Cart публикует события изменений в корзинах заказов в ленте событий. Микросервисы Recommendations и Shopper Tracking подписываются на эти события и реагируют на их поступление

В этом разделе мы реализуем следующие компоненты (выделены на рис. 2.10): модель предметной области Shopping Cart и EventFeed. (В главе 4 мы вернемся к реализации лент событий и подписчиков на события.) Модель предметной области отвечает за генерацию событий, а компонент EventFeed предоставляет другим микросервисам возможность чтения публикуемых микросервисом Shopping Cart событий.

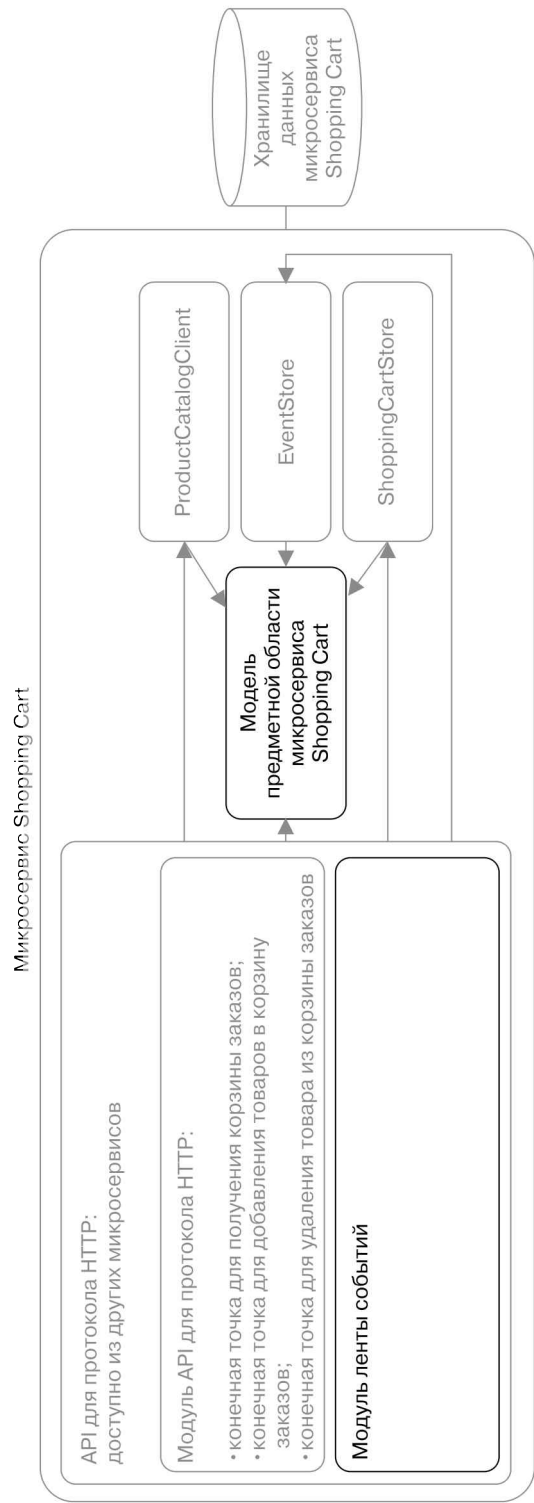


Рис. 2.10. Лента событий микросервиса Shopping Cart публикует события для остального интернет-магазина

Реализация ленты событий включает следующие этапы.

- ❑ *Генерация событий.* Код из модели предметной области Shopping Cart генерирует событие, когда происходит что-то важное (в соответствии с бизнес-правилами). Важные события наблюдаются при добавлении новых товаров в корзину заказов или удалении из нее старых.
- ❑ *Хранение событий.* Генерируемые моделью предметной области Shopping Cart события хранятся в хранилище данных микросервиса.
- ❑ *Публикация событий.* Благодаря реализации ленты событий другие микросервисы могут подписываться на них путем частых опросов (polling).

Мы займемся ими по очереди.

Генерация события

Прежде чем опубликовать событие, его необходимо сгенерировать. Обычно в микросервисах этим занимается предметно-ориентированный код, и Shopping Cart не исключение. При добавлении товаров в корзину заказов предметно-ориентированный объект класса `ShoppingCart` генерирует событие путем вызова метода `Raise` для объекта, реализующего интерфейс `IEventStore`, предоставляя необходимые для этого события данные (листинг 2.13).

Листинг 2.13. Генерация событий

```
public void AddItems(
    IEnumerable<ShoppingCartItem> shoppingCartItems,
    IEventStore eventStore)
{
    foreach (var item in shoppingCartItems)
        if (this.items.Add(item))
            eventStore.Raise(
                "ShoppingCartItemAdded",
                new { UserId, item });
}
```

Генерирует событие для каждого товара с помощью объекта eventStore

С точки зрения предметно-ориентированного кода генерация события — всего лишь вопрос вызова метода `Raise` объекта, реализующего интерфейс `IEventStore`. Предметно-ориентированный объект класса `ShoppingCart` генерирует событие и при удалении товара. Код генерации этого события почти такой же, так что я оставляю его реализацию в качестве упражнения читателю.

Хранение события

Генерируемые предметно-ориентированным кодом события не публикуются для других микросервисов сразу же. Вместо этого они сохраняются, а затем публикуются асинхронно. Другими словами, при генерации события объект `eventStore` только сохраняет событие в базе данных (рис. 2.14). Я оставляю этот код на волю

вашего воображения, как и весь остальной код текущей главы, предназначенный для работы с базами данных. Здесь важно лишь понимать, что каждое событие сохраняется как отдельная запись в базе данных — хранилище событий и получает номер из равномерно возрастающей целочисленной последовательности.

Листинг 2.14. Сохранение данных событий в базе данных

```
public void Raise(string eventName, object content)
{
    var seqNumber = database.NextSequenceNumber(); ← Получаем для события номер
    database.Add(                                     из последовательности
        new Event(
            seqNumber,
            DateTimeOffset.UtcNow,
            eventName,
            content));
}
```

Объект `eventStore` сохраняет все ностунающие события и отслеживает порядок их ностунления. Мы вернемся к вопросу хранения событий в главе 5, в которой будем рассматривать их реализацию подробнее.

Простая лента событий

После сохранения события оказываются готовыми к публикации. В определенном смысле они *уже* опубликованы. И хотя микросервисы *подписываются* на события других микросервисов, лента событий функционирует так, что подписчикам приходится периодически спрашивать, не появились ли новые события. Поскольку за опрос о появлении новых событий отвечают подписчики, в микросервисе `Shopping Cart` достаточно просто добавить конечную точку, которая предоставляла бы подписчикам возможность спрашивать о новых событиях. Например, для получения всех событий с порядковым номером, большим чем 100, подписчик может отправить следующий запрос:

```
GET /events?start=100 HTTP/1.1
Host: shoppingcart.my.company.com
Accept: application/json
```

А если подписчику нужно ограничить количество ностунающих за один вызов событий, он может добавить в запрос аргумент `end`:

```
GET /events?start=100&end=200 HTTP/1.1
Host: shoppingcart.my.company.com
Accept: application/json
```

Поместим реализацию конечной точки `/events` в новый модуль `Nancy`, как показано в листинге 2.15. Эта конечная точка принимает на входе необязательные начальное и конечное значения, что дает возможность остальным микросервисам запрашивать определенный диапазон событий.

Листинг 2.15. Делаем события видимыми другим микросервисам

```

namespace ShoppingCart.EventFeed
{
    using Nancy;

    public class EventsFeedModule : NancyModule
    {
        public EventsFeedModule(IEventStore eventStore) : base("/events")
        {
            Get("/", _ =>
            {
                long firstEventSequenceNumber, lastEventSequenceNumber;
                if (!long.TryParse(this.Request.Query.start.Value,
                    out firstEventSequenceNumber))
                    firstEventSequenceNumber = 0;
                if (!long.TryParse(this.Request.Query.end.Value,
                    out lastEventSequenceNumber))
                    lastEventSequenceNumber = long.MaxValue;

                return
                    eventStore.GetEvents(
                        firstEventSequenceNumber,
                        lastEventSequenceNumber);
            });
        }
    }
}

```

Читаем начальное и конечное значения из параметра строки запроса

Возвращаем первичный список событий. Сериализацию событий в тело ответа берет на себя фреймворк Nancy

В основном класс `EventsFeedModule` использует уже знакомые вам возможности фреймворка Nancy. Единственная новинка здесь — то, что значения `start` и `end` читаются из параметров строки запроса. Как и для сегментов нути URL, фреймворк Nancy предоставляет с помощью динамического объекта `Request.Query` удобный способ доступа к параметрам строки запроса.

Класс `EventsFeedModule` использует хранилище событий, чтобы отфильтровать те события, которые находятся между полученными от клиента значениями `start` и `end`. Хотя оптимальной является фильтрация на уровне базы данных, следующая простая реализация иллюстрирует ее довольно хорошо (листинг 2.16).

Листинг 2.16. Фильтрация событий на основе значений `start` и `end`

```

public IEnumerable<Event> GetEvents(
    long firstEventSequenceNumber,
    long lastEventSequenceNumber) =>
    database
        .Where(e =>
            e.SequenceNumber >= firstEventSequenceNumber &&
            e.SequenceNumber <= lastEventSequenceNumber)
        .OrderBy(e => e.SequenceNumber);

```

Благодаря конечной точке `/events` микросервисы могут при необходимости подписаться на события микросервиса Shopping Cart путем онросов этой конечной точки. Подписчики могут — и должны — использовать параметры строки запроса

`start` и `end` для получения только новых событий. Если микросервис `Shopping Cart` не функционирует во время выполнения подписчиком опроса, подписчик может запросить те же события позже. Аналогично, если подписчик не работал в течение некоторого времени, он может паверстать упущенное, запрашивая у `Shopping Cart` события, начиная с последнего видеппого им. Как уже упоминалось, паша реализация лепты событий не полнофункциональна, но она демонстрирует главное: микросервисы могут подписываться на события и необходимый для этого код прост.

Вы завершили версию 1 реализации вашего первого микросервиса. Как можно видеть, микросервис представляет собой небольшую узкоаправлеппую программу: он обеспечивает лишь одну бизнес-возможность. Вы также могли заметить, что код микросервиса прост и легок для поппимания. Именно поэтому можно падеяться на возможность быстрого создания новых микросервисов и замены старых.

2.3. Выполнение кода

Теперь, когда код микросервиса `Shopping Cart` завершен, вы можете запустить его на выполнение аналогично тому, как запускали пример из главы 1, — из `Visual Studio` или командной строки с помощью утилиты `dotnet`. Проверить все конечные точки можно с помощью `Postman` или любой аналогичной утилиты. Когда вы в первый раз попытаетесь прочитать корзину заказов с помощью HTTP-запроса `GET` к конечной точке `/shoppingcart/123`, корзина окажется пустой. Добавьте в нее какие-нибудь товары с помощью HTTP-запроса `POST` к конечной точке `/shoppingcart/123/items`, после чего прочитайте ее снова. Ответ должен содержать добавленные товары. Можете также взглянуть на ленту событий по адресу `/events`, в которой вы должны увидеть события для каждого добавленного товара.

ПРЕДУПРЕЖДЕНИЕ

Я не привел в этой главе реализаций компонентов `EventStore` или `ShoppingCartStore`. Если вы не создадите свои собственные их реализации, ваш микросервис не будет функционировать.

2.4. Резюме

- ❑ Реализация законченного микросервиса не требует большого количества кода. Микросервис `Shopping Cart` состоит всего лишь из:
 - двух коротких модулей `Nancy`;
 - простого предметно-ориентированного класса `ShoppingCart`;
 - класса клиента для обращения к микросервису `Product Catalog`;
 - двух простых классов доступа к данным: `ShoppingCartDataStore` и `EventStore` (не приводятся в этой главе).
- ❑ Фреймворк `Nancy` сильно упрощает реализацию API для протокола HTTP. Предоставляемый им API описания маршрутов упрощает добавление в микросервис

копечных точек. Достаточно добавить описание маршрута и обработчик, например, `Get("/hello", _ ? "world")`, в конструктор модуля Nancy, как Nancy автоматически обнаружит его и начнет маршрутизировать запросы к обработчику.

- ❑ Фреймворк Nancy автоматически сериализует данные ответа и десериализует данные запроса. В коде приложения можно возвращать из обработчика любой сериализуемый объект или использовать привязку модели для десериализации данных запроса.
- ❑ Nancy поддерживает прямо «из коробки» форматы XML и JSON как для запросов, так и для ответов.
- ❑ Всегда следует учитывать возможность того, что другие микросервисы окажутся недоступными. Чтобы не плодить ошибки, желательно оборачивать все удаленные вызовы в стратегии обработки ошибок.
- ❑ Библиотека Polly очень удобна для реализации стратегий обработки ошибок и оборачивания в них удаленных вызовов.
- ❑ Реализация простой ленты событий не представляет сложностей и дает другим микросервисам возможность реагировать на события. Реализованная в этой главе лента событий «для бедных» — всего лишь крошечный модуль Nancy.
- ❑ За генерацию событий, сохраняемых затем в хранилище событий и публикуемых с помощью ленты событий, обычно отвечает предметно-ориентированный код.

Часть II

Создание микросервисов

В этой части нашей книги вы научитесь проектировать и программировать микросервисы. Все последующие разнообразные вопросы касаются проектирования и программирования надежных, легко сопровождаемых микросервисов.

- ❑ В главе 3 объясняется, как разбить систему на набор взаимосвязанных микросервисов.
- ❑ Глава 4 демонстрирует обеспечение плавной работой конечным пользователям функциональности посредством совместной работы микросервисов. Вы познакомитесь с тремя разновидностями взаимодействия и узнаете, когда следует использовать каждую из них.
- ❑ Глава 5 исследует данные в системах микросервисов и обсуждает, какие микросервисы за какие данные должны отвечать.
- ❑ В главе 6 представлены некоторые методики повышения устойчивости системы микросервисов к ошибкам. С помощью этих методик вы сможете создавать системы, работающие даже в случаях сбоев сети или отдельных микросервисов.
- ❑ Глава 7 посвящена тестированию. Вы научитесь создавать автоматизированные наборы тестов для системы микросервисов, начиная от общих тестов уровня всей системы и заканчивая узконаправленными модульными тестами.

К концу части II вы научитесь проектировать микросервисы, а также использовать платформу .NET Core и фреймворк Nancy для их программирования.

3

Распознавание микросервисов и определение их области действия

В этой главе:

- ❑ определение области действия микросервисов, реализующих бизнес-возможности;
- ❑ определение области действия микросервисов, предназначенных для вспомогательных технических возможностей;
- ❑ сложные случаи определения области действия микросервисов;
- ❑ выделение новых микросервисов из существующих.

Чтобы преуспевать в работе с микросервисами, важно уметь правильно оценивать область действия каждого из них. Если ваши микросервисы слишком велики, производственный цикл создания новых возможностей и исправления ошибок оказывается слишком долгим. Если же микросервисы слишком малы, то усиливается связанность между ними. Если же они имеют должный размер, но неправильные границы, связанность тоже усиливается, а более сильная связанность приводит к более длительному производственному циклу. Другими словами, если вы не знаете, как правильно определить область действия микросервисов, то лишитесь большей части выгод от их использования. В этой главе я научу вас искусству определения правильных областей действия для каждого микросервиса так, чтобы они оставались слабо связанными.

Главный фактор распознавания и определения области действия микросервисов — бизнес-возможности, второй по степени важности — вспомогательные технические возможности. Их учет в ходе работы позволяет создать микросервисы, чьи характеристики хорошо согласуются со списком отличительных признаков из главы 1.

- ❑ Микросервис отвечает за одну возможность.
- ❑ Микросервисы можно развертывать по отдельности.
- ❑ Микросервис состоит из одного или нескольких процессов.
- ❑ У микросервиса имеется собственное хранилище данных.

- ❑ Небольшая команда разработчиков может сопровождать несколько микросервисов.
- ❑ Микросервис можно легко заменить.

Первые два и последние два из этих отличительных признаков удастся реализовать только при удачном определении области действия микросервиса. Необходимо учитывать также некоторые нюансы, относящиеся к вопросам реализации, но неправильная область действия сразу сделает невозможным соответствие этим четырем признакам.

3.1. Главный фактор определения области действия микросервисов: бизнес-возможности

Один микросервис должен реализовывать одну возможность. Например, микросервис Shopping Cart должен отслеживать товары в корзине заказов пользователя. Основной способ распознавания возможностей для микросервисов состоит в анализе бизнес-задач и определении бизнес-возможностей. Каждую бизнес-возможность должен реализовывать отдельный микросервис.

Что такое бизнес-возможность

Бизнес-возможность — что-либо, выполняемое организацией и способствующее достижению ее бизнес-целей. Например, обработка корзины заказов в интернет-магазине — это бизнес-возможность, помогающая достичь более широкой бизнес-цели, состоящей в обеспечении возможности покупки товаров пользователями. У каждого коммерческого предприятия есть некоторое множество бизнес-возможностей, которые вместе образуют его общую бизнес-функцию.

Если говорить о задании соответствия бизнес-возможности микросервису, то микросервис моделирует бизнес-возможность. В некоторых случаях он реализует бизнес-возможность в целом и полностью автоматизирует ее выполнение. В других микросервис реализует только часть бизнес-возможности и, таким образом, автоматизирует ее выполнение лишь частично. В обоих случаях область действия микросервиса равна бизнес-возможности.

Бизнес-возможности и ограниченные контексты

Предметно-ориентированное проектирование (domain-driven design) — подход к проектированию программного обеспечения, основанный на моделировании предметной области. Важный этап этого процесса — выяснение используемого специалистами по данной предметной области языка. Оказывается, далеко не всегда специалисты по предметной области говорят на одном языке.

В разных частях предметной области внимание уделяется различным вещам, так что конкретное слово, например «клиент», может иметь разные оттенки смысла в разных частях предметной области. В отделе продаж фирмы, занимающейся продажей копировальных

устройств, клиентом могут считать фирму, покупающую несколько таких устройств и представленную в основном сотрудником, занимающимся закупками. В отделе обслуживания клиентов клиентом может считаться конечный пользователь, у которого возникли проблемы во время работы с копировальным устройством. При моделировании предметной области фирмы по продаже копировальных устройств в разных частях модели слово «клиент» означает разное.

Ограниченным контекстом (bounded context) в предметно-ориентированном проектировании называется некоторая часть предметной области, в пределах которой слова сохраняют свое значение. Ограниченные контексты связаны с бизнес-возможностями, но несколько отличаются от них. Ограниченный контекст определяет часть предметной области, внутри которой язык единообразен. Бизнес-возможности же связаны с тем, что необходимо выполнить предприятию. В пределах одного ограниченного контекста предприятию может потребоваться выполнять несколько действий, каждое из которых, вероятно, представляет собой бизнес-возможность.

Выделение бизнес-возможностей

Чтобы понимать работу бизнеса, необходимо хорошо знать предметную область. Для понимания его функционирования вы должны уметь распознавать бизнес-возможности, из которых и состоит бизнес, и процессы, связанные с реализацией этих возможностей. Другими словами, для распознавания бизнес-возможностей необходимо изучить предметную область. Подобные знания можно получить, беседуя с теми, кто лучше всего знает предметную область: бизнес-аналитиками, конечными пользователями программного обеспечения и т. п. — всеми теми, кто непосредственно занимается ежедневной рутинной, движущей бизнес.

Организация коммерческого предприятия обычно отражает его предметную область. За различные части предметной области отвечают разные группы людей, причем каждая из них отвечает за реализацию конкретных бизнес-возможностей. Эта организационная структура может помочь в определении областей действия микросервисов. Возлагаемая на микросервис обязанность должна относиться к сфере ответственности только одной группы. Если она выходит за границу между группами, вероятно, область действия была определена слишком широко, а значит, обязанность не является единым целым, что приведет к плохой сопровождаемости. Эти наблюдения хорошо согласуются с правилом, известным как *закон Конвея*¹: *структура проекта системы (в широком смысле этого слова), создаваемой какой-либо организацией, будет в точности отражать структуру взаимодействий в этой организации.*

Иногда встречаются части предметной области, в которых организация и предметная область не сочетаются. Для подобных случаев существуют два подхода, при которых соблюдается закон Конвея. Во-первых, можно признать как должное то, что система не вполне будет отражать предметную область, и реализовать несколько микросервисов, плохо сочетающихся с предметной областью, но хорошо — с организацией.

¹ Conway M. How Do Committees Invent? // Datamation Magazine, 1968. — April [Электронный ресурс]. — Режим доступа: <http://www.melconway.com/research/committees.html>.

Во-вторых, можно изменить структуру организации так, чтобы она отражала предметную область. Оба подхода могут вызвать проблемы. При первом возникает риск создать тесно связанные микросервисы с неподходящей областью действия. Второй требует перемещения людей и перераспределения обязанностей между разными группами. Выполнить подобные изменения довольно трудно. Вы должны сделать выбор, исходя из прагматических соображений и оценки того, какой из этих подходов доставит меньше сложностей.

Рассмотрим пример, чтобы лучше понять, что представляют собой бизнес-возможности.

Пример: POS-система

В этой главе мы рассмотрим пример POS-системы (от англ. point of sale — «точка продажи») (рис. 3.1). Я вкратце познакомлю вас с предметной областью, после чего мы выясним, как распознать в ней бизнес-возможности. Наконец, более детально рассмотрим определение области действия одного из микросервисов этой системы.

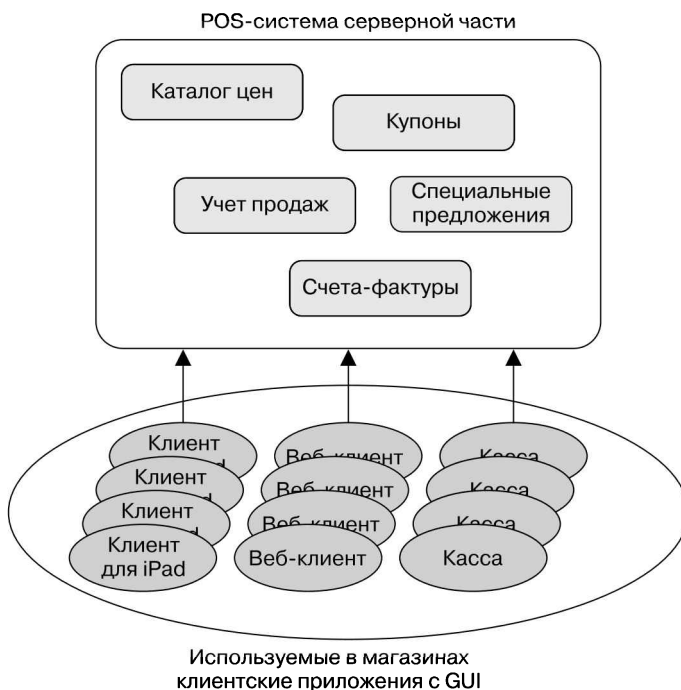


Рис. 3.1. POS-система для большой торговой сети, состоящая из серверной части, в которой реализованы все бизнес-возможности системы, и тонких клиентов с GUI, используемых кассирами в магазинах. Микросервисы в серверной части реализуют бизнес-возможности

Эта POS-система используется во всех магазинах большой торговой сети. Кассиры в магазинах взаимодействуют с системой посредством тонкого клиента с графиче-

ским интерфейсом пользователя (GUI) — им может быть приложение для планшета, веб-приложение или специализированное приложение для кассы (кассового аппарата, если угодно). Клиентское приложение с GUI представляет собой всего лишь топкий слой, располагающийся перед серверной (прикладной) частью системы. Вся бизнес-логика (бизнес-возможности) реализована в прикладной части, и именно на ней мы сосредоточимся.

Система предоставляет кассирам множество различных функций:

- ☐ сканирование товаров и добавление их в счет-фактуру;
- ☐ подготовку счета-фактуры;
- ☐ списание средств с кредитной карты посредством подключаемого к клиенту картридера;
- ☐ регистрацию платежа наличными;
- ☐ прием купонов;
- ☐ печать чека;
- ☐ отправку пользователю электронного чека;
- ☐ поиск в каталоге товаров;
- ☐ сканирование одного или нескольких товаров для отображения их цен и относящихся к ним специальных предложений.

Эти функции — то, что делает система для кассира, но они не совпадают в точности с бизнес-возможностями, составляющими движущую силу нашей POS-системы.

Распознавание бизнес-возможностей в предметной области POS-системы

Для распознавания бизнес-возможностей, составляющих движущую силу POS-системы, стоит взглянуть несколько шире на приведенный список функций. Необходимо определить, какие действия должны выполняться «за кулисами», чтобы обеспечить эту функциональность.

Начнем с функции «Поиск в каталоге товаров», где очевидная бизнес-возможность — поддержание работы каталога товаров. Это первый кандидат на роль бизнес-возможности, определяющей область действия одного из микросервисов. За предоставление доступа к текущему каталогу товаров будет отвечать микросервис Product Catalog. Необходимо будет время от времени обновлять каталог товаров, но для этой функциональности сеть магазинов использует другую систему. Микросервис Product Catalog должен будет отражать произведенные в этой другой системе изменения, так что область действия микросервиса Product Catalog должна включать получение обновлений для каталога товаров.

Следующая бизнес-возможность, которую, возможно, вы распознали, — это учет в счете-фактуре специальных предложений. Специальные предложения предоставляют покупателям скидки при покупке определенных комплектов товаров. Комплект может состоять из некоторого количества единиц одного товара по сниженной

цене (например, три единицы по цене двух) или представлять собой сочетание различных товаров (например, купите товар А и получите скидку на товар Б — 10 % от цены). В любом случае получаемый кассиром от клиента POS-терминала счет-фактура должен автоматически учитывать все подходящие специальные предложения. Эта бизнес-возможность — второй кандидат на роль области действия для микросервиса. Микросервис Special Offers будет определять, когда необходимо применить специальные предложения и какую скидку получит покупатель.

Еще раз посмотрев на перечень функциональности, видим, что система должна предоставлять кассирам возможность скапировать один или несколько товаров для отображения их цен и относящихся к ним специальных предложений. Это значит, что бизнес-возможность микросервиса Special Offers состоит не только в учете специальных предложений в счетах-фактурах — она включает также возможность поиска специальных предложений по товарам.

Продолжая поиски бизнес-возможностей в POS-системе, получаем в итоге следующий список:

- ☐ Product Catalog («Каталог товаров»);
- ☐ Price Catalog («Каталог цен»);
- ☐ Price Calculation («Расчет цены»);
- ☐ Special Offers («Специальные предложения»);
- ☐ Coupons («Купоны»);
- ☐ Sales Records («Учет продаж»);
- ☐ Invoice («Счет-фактура»);
- ☐ Payment («Платеж»).

На рис. 3.2 показано соответствие функциональности бизнес-возможностям. Речь идет о логическом соответствии в смысле демонстрации необходимых для реализации каждой из функций бизнес-возможностей без указания технических возможностей. Например, стрелка от «Подготовить счет-фактуру» к Coupons не означает обращения от кода подготовки счета-фактуры в клиенте к микросервису Coupons. Скорее эта стрелка говорит о том, что для подготовки счета-фактуры необходимо учесть купоны, так что функция «Подготовить счет-фактуру» зависит от бизнес-возможности Coupons.

Создание подобных карт соответствий весьма поучительно, поскольку заставляет задумываться о том, как достигается выполнение каждой из функций и в чем состоит задача каждой бизнес-возможности. Поиск бизнес-возможностей во встречающихся на практике предметных областях может оказаться непростой задачей и часто требует немалого числа итераций. Кроме того, перечень бизнес-возможностей не статичный список, фиксируемый в начале разработки, он расширяется и меняется с течением времени по мере расширения и углубления степени вашего понимания предметной области.

Завершив первую итерацию распознавания бизнес-возможностей, можем перейти к более детальному обсуждению одной из этих возможностей и определению области действия микросервиса.

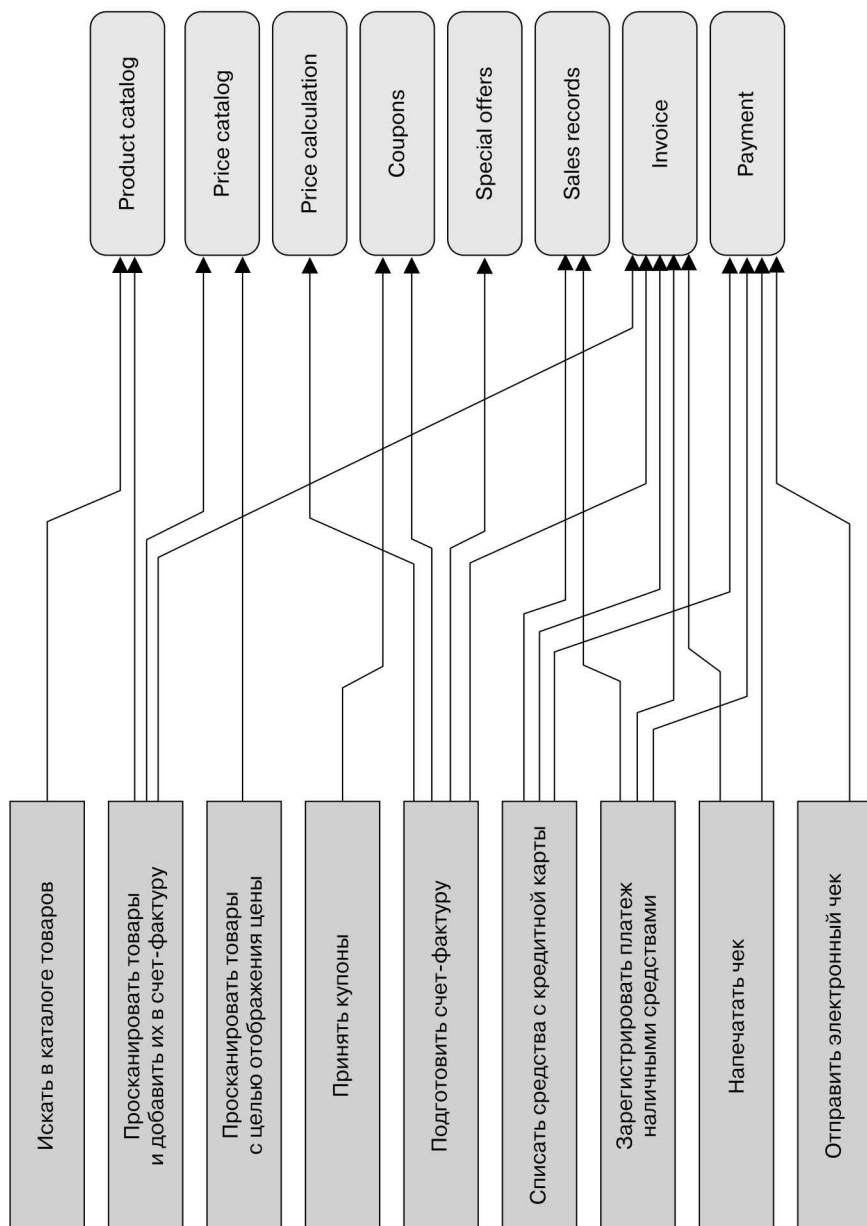


Рис. 3.2. Функции слева зависят от бизнес-возможностей справа. Каждая стрелка отражает зависимость между функцией и возможностью

Микросервис Special Offers

Микросервис Special Offers («Специальные предложения») основан на бизнес-возможности Special Offers. Чтобы сузить область действия этого микросервиса, рассмотрим соответствующую бизнес-возможность подробнее и выясним, какие процессы в ней задействованы (рис. 3.3). Каждый процесс обеспечивает реализацию одной из частей данной бизнес-возможности.



Рис. 3.3. Бизнес-возможность Special Offers включает несколько различных процессов

Бизнес-возможность Special Offers разбита на пять процессов. Четыре из них ориентированы на клиентские GUI-приложения точек продаж. Пятая — мониторинг использования специальных предложений — ориентирована на сам бизнес, которому важно, какие специальные предложения привлекают покупателей.

Реализация бизнес-возможности в виде микросервиса означает, что нужно:

- ❑ сделать доступными для использования четыре клиентоориентированных процесса в виде конечных точек API, к которым могут обращаться другие микросервисы;
- ❑ реализовать посредством ленты событий процесс мониторинга использования. Отвечающие за бизнес-аналитику части POS-системы смогут подписаться на эти события и воспользоваться ими для отслеживания использования покупателями специальных предложений.

Компоненты микросервиса Special Offers показаны на рис. 3.4.

Компоненты микросервиса Special Offers аналогичны компонентам микросервиса Shopping Cart из главы 2 (продублированы на рис. 3.5).

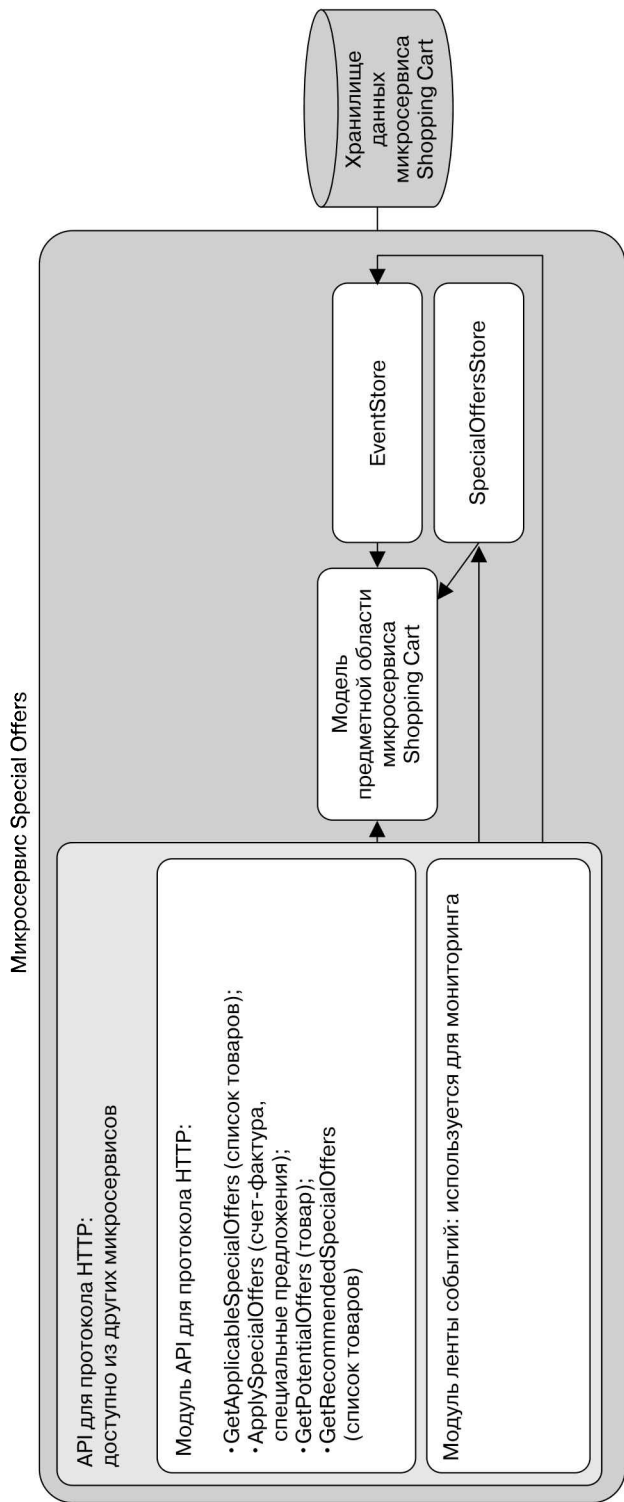


Рис. 3.4. Процессы, относящиеся к бизнес-возможности Special Offers, отражены в реализации микросервиса Special Offers. Процессы доступны для использования другими микросервисами посредством API для протокола HTTP микросервиса Special Offers

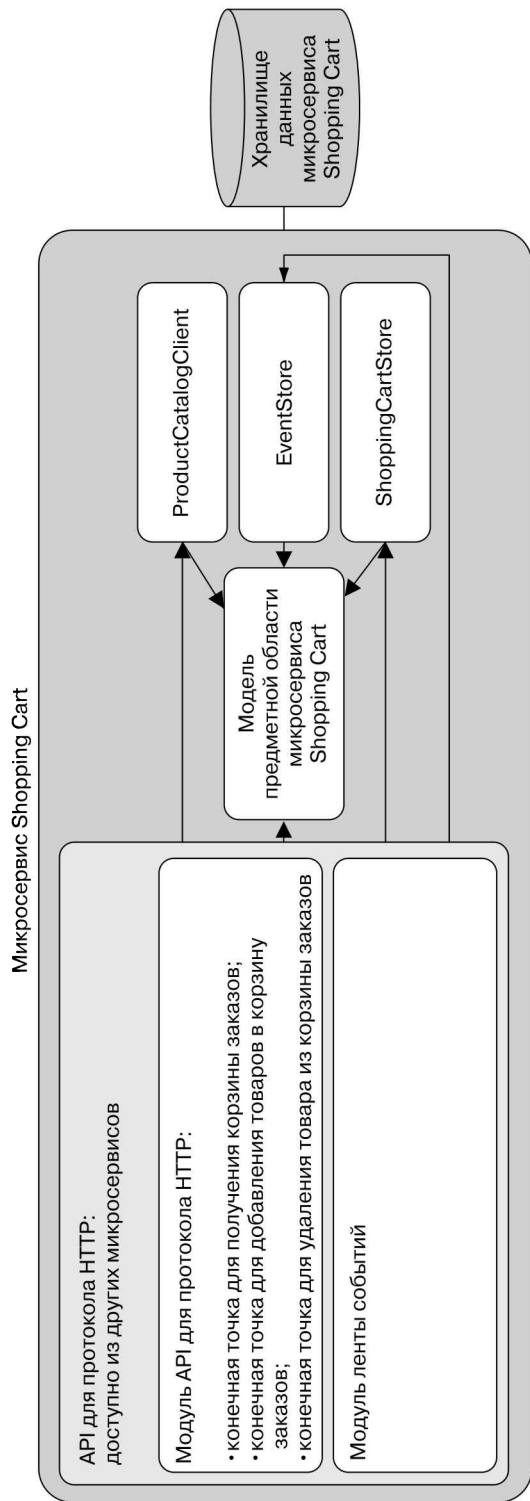


Рис. 3.5. Компоненты микросервиса Shopping Cart из главы 2 аналогичны компонентам микросервиса Special Offers

3.2. Дополнительный фактор определения области действия микросервисов: вспомогательные технические возможности

Еще один способ определения области действия микросервисов — исследование вспомогательных технических возможностей. *Вспомогательная техническая возможность* (*supporting technical capability*) — нечто, не способствующее непосредственно реализации бизнес-цели, но обеспечивающее возможность работы других микросервисов, например, интеграция с другими системами или планирование выполнения события в какой-то момент в будущем.

Что такое вспомогательная техническая возможность

Вспомогательные технические возможности — вторичный фактор определения области действия микросервисов, поскольку они не способствуют непосредственно реализации бизнес-целей системы. Цель их существования состоит в упрощении и поддержке других микросервисов, реализующих бизнес-возможности.

Как вы помните, один из отличительных признаков хорошего микросервиса — его заменяемость. Но микросервис, реализующий, помимо бизнес-возможности, еще и сложную техническую возможность, может стать слишком сложным и большим для того, чтобы быть заменяемым. В подобных случаях желательно реализовать техническую возможность в отдельном микросервисе, вспомогательном для исходного. Прежде чем перейти к обсуждению вопроса о том, как и когда распознавать вспомогательные технические возможности, не помешает рассмотреть несколько примеров.

Примеры вспомогательных технических возможностей

Чтобы вы лучше поняли, что я имею в виду под вспомогательными техническими возможностями, рассмотрим два примера: интеграцию с другой системой и возможность отправки уведомлений покупателям.

Интеграция с внешней системой — каталогом товаров

В примере POS-системы мы распознали каталог товаров в качестве бизнес-возможности. Я также упоминал, что информация о товаре хранится в отдельной системе, внешней по отношению к основанной на микросервисах POS-системе. Эта другая система — система планирования ресурсов предприятия (Enterprise Resource Planning (ERP)). Следовательно, микросервис Product Catalog должен интегрироваться с ERP-системой (рис. 3.6). Интеграцию должен взять на себя отдельный микросервис.

Допустим, у вас нет возможности менять ERP-систему, так что придется выполнять интеграцию с помощью интерфейса ERP-системы. Она может использовать работающий по протоколу SOAP веб-сервис для выборки информации или экспортировать информацию о товаре в проприетарный формат файлов. В любом случае именно она будет диктовать условия интеграции. В зависимости от предоставляемого ERP-системой интерфейса сложность этой задачи может различаться. В любом

случае эта задача в основном связана с техническими деталями интеграции с другой системой и может оказаться довольно сложной. Цель этой интеграции — поддержка работы микросервиса Product Catalog.



Рис. 3.6. Данные о товаре передаются из ERP-системы в микросервис Product Catalog. ERP-система сама определяет протокол, который будет использоваться для получения из нее информации о товаре. Она может предоставлять, например, работающий по протоколу SOAP веб-сервис для выборки информации или экспортировать информацию о товаре в проприетарный формат файлов

Мы выделим интеграционную функциональность из микросервиса Product Catalog и реализуем ее в отдельном микросервисе ERP Integration («Интеграция с ERP»), отвечающем исключительно за эту интеграцию (рис. 3.7). Для этого есть два основания.

- Благодаря вынесению чисто технических сложностей интеграции в отдельный микросервис становится возможно максимально сузить область действия микросервиса Product Catalog и сфокусироваться на основной его задаче.
- Благодаря использованию отдельного микросервиса для форматирования и организации ERP-данных становится возможно отделить представление ERP-системы о товаре от POS-системы. Не забывайте о том, что в разных частях предметной области термины могут иметь разные значения. Маловероятно, что микросервис Product Catalog и ERP-система будут едины в вопросе моделирования занизис о товаре. Необходимо вынолнить нребразование между двумя представлениями, и лучше всего сделать это с номощью нового микросервиса. В терминах нредметно-ориентированного нроектирования новый микросервис служит *предохранительным слоем (anti-corruption layer)*.

ПРИМЕЧАНИЕ

Предохранительный слой — понятие, позаимствованное из предметно-ориентированного проектирования. Он используется при взаимодействии двух систем и защищает модель предметной области одной системы от контаминации языком или понятиями модели другой.

Донолнительная выгода от интеграции в отдельном микросервисе состоит в том, что именно тут удобно нрилагать усилия но решению любых связанных с интеграцией вопросов надежности. Если ERP-система ненадежна, нринимать меры следует именно в микросервисе ERP Integration. Если ERP-система работает слишком медленно,

именно микросервис ERP Integration должен позаботиться об этом. Со временем можно будет настроить используемые в микросервисе ERP Integration стратегии, чтобы справиться с любыми возможными проблемами с надежностью ERP-системы, вообще не трогая микросервис Product Catalog. Эта интеграция с ERP-системой представляет собой пример вспомогательной технической возможности, а микросервис ERP Integration — пример реализующего эту возможность микросервиса.

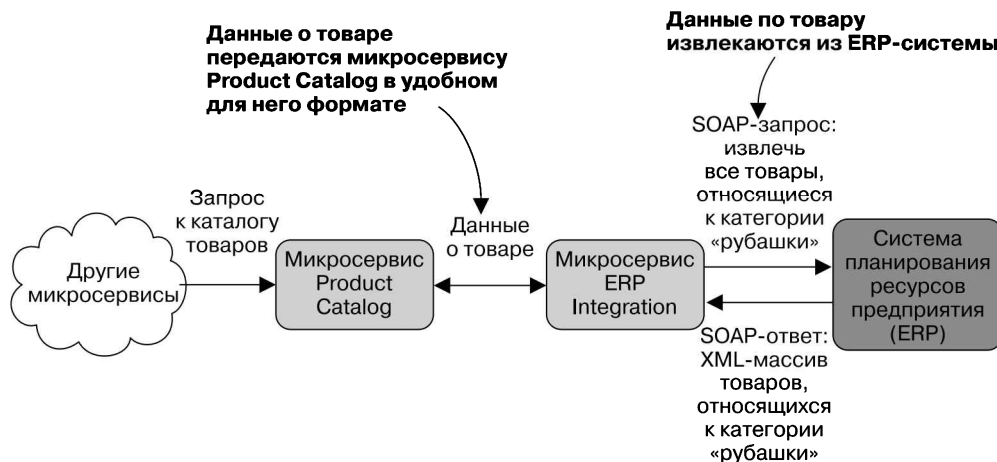


Рис. 3.7. Микросервис ERP Integration поддерживает функционирование микросервиса Product Catalog путем интеграции с ERP-системой. Он преобразует предоставляемые ERP-системой данные о товарах в понятный микросервису Product Catalog вид

Отправка уведомлений покупателям

Рассмотрим теперь расширение POS-системы путем реализации возможности отправки покупателям уведомлений о новых специальных предложениях по электронной почте, в виде СМС или push-уведомлений в мобильное приложение. Эту возможность можно встроить в один или несколько отдельных микросервисов.

Пока что наша POS-система не знает ничего о покупателях. Чтобы обеспечить повышенную вовлеченность потребителей и их лояльность, фирма решает запустить небольшую программу лояльности, в которой покупатели могут подписаться на рассылку оповещений о специальных предложениях. Программа лояльности покупателей — новая бизнес-возможность, за нее будет отвечать новый микросервис Loyalty Program («Программа лояльности»). Этот микросервис, отвечающий за оповещение зарегистрированных покупателей при каждом появлении нового специального предложения, показан на рис. 3.8.

В процессе регистрации покупатели могут выбрать способ уведомления: по электронной почте, посредством СМС или, если у них установлено фирменное мобильное приложение, push-уведомлений. Это несколько усложняет микросервис Loyalty Program, так как ему приходится не только выбирать тип уведомлений, но и учитывать особенности работы каждого из них. В качестве первого шага мы введем в систему во вспомогательному техническому микросервису для каждого типа уведомлений. Они показаны на рис. 3.9.

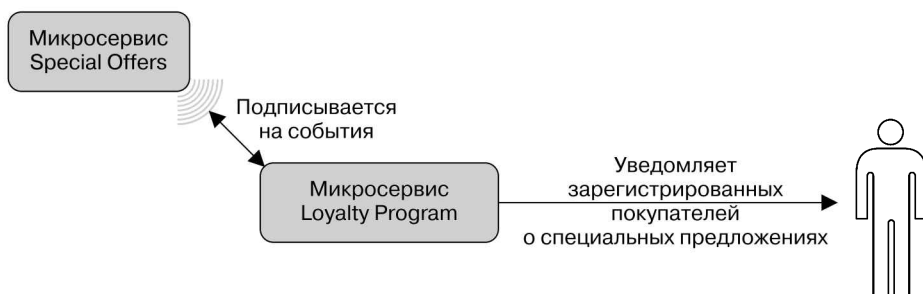


Рис. 3.8. Микросервис Loyalty Program подписывается на события, генерируемые микросервисом Special Offers, и уведомляет зарегистрированных покупателей при появлении новых специальных предложений

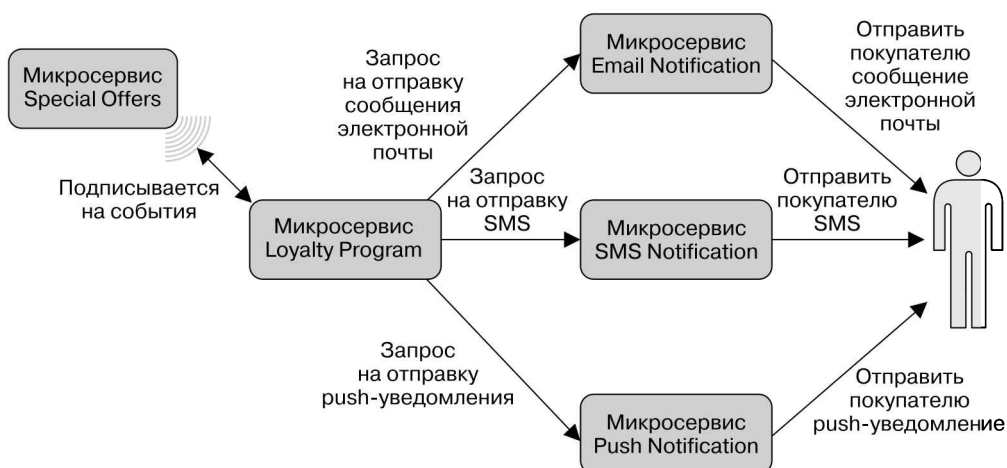


Рис. 3.9. Чтобы микросервис Loyalty Program не оказался перегружен техническими деталями по обработке всех типов уведомлений, введем в систему три вспомогательных технических микросервиса, по одному для каждого из типов уведомлений

Уже лучше. Микросервису Loyalty Program не нужно реализовывать все нюансы обработки каждого из типов уведомлений, что позволяет максимально сузить область его действия и сфокусировать его на основной задаче. Однако эта ситуация неидеальна: он все равно должен заниматься определением того, какой из вспомогательных технических микросервисов вызвать для каждого из зарегистрированных покупателей.

Поэтому приходится внедрить еще один микросервис, служащий адаптером для трех микросервисов, отвечающих за три типа уведомлений. Этот новый микросервис Notifications (рис. 3.10) отвечает за выбор типа уведомления всякий раз, когда необходимо уведомить покупателя. Это нельзя считать бизнес-возможностью, хотя она в меньшей степени техническая, чем задача отправки СМС. Я предпочитаю считать, что Notifications скорее вспомогательный технический микросервис, а не микросервис, реализующий бизнес-возможность.

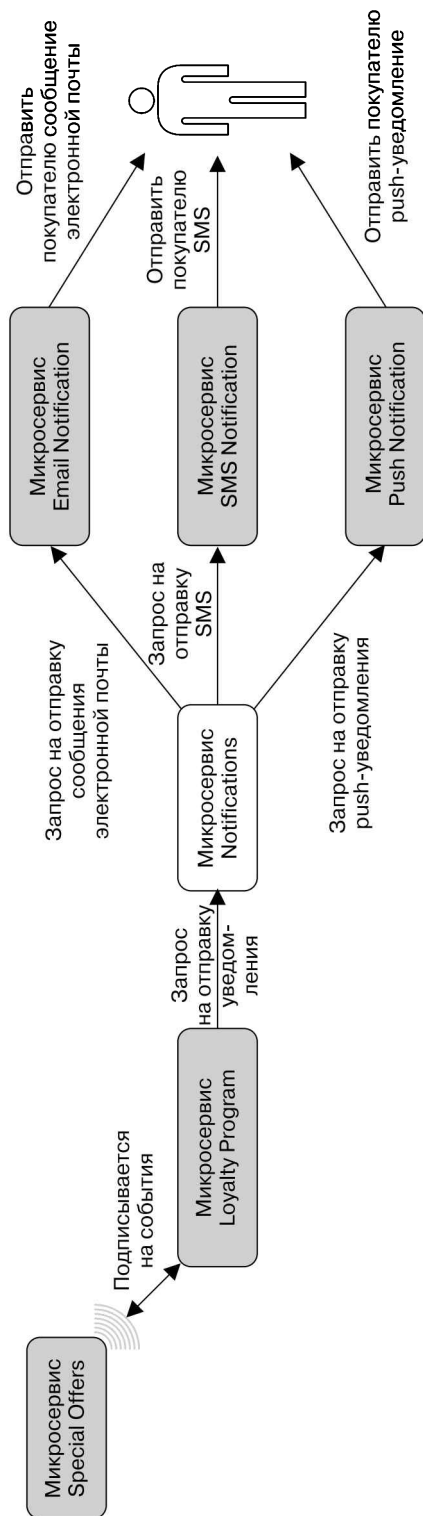


Рис. 3.10. Чтобы снизить степень сложности микросервиса Loyalty Program, введем в систему микросервис Notifications, отвечающий за выбор типа уведомления в зависимости от предпочтений пользователя. Еще одно преимущество использования этого микросервиса — облегчение для других микросервисов задачи по уведомлению покупателей

Этот пример вспомогательной технической возможности отличается от предыдущего примера ERP-интеграции тем, что другим микросервисам тоже может понадобиться отправить уведомления конкретным покупателям. Например, одна из функций POS-системы состоит в отправке покупателю электронного чека. Отвечающий за это микросервис тоже может воспользоваться микросервисом Notifications. Один из мотивов к перемещению этой вспомогательной технической возможности в отдельные микросервисы — то, что можно повторно использовать ее реализацию.

Распознавание технических возможностей

Вспомогательные технические возможности вводят в систему с целью упрощения микросервисов, которые реализуют бизнес-возможности. Иногда, как в случае отправки уведомлений, удастся распознать необходимую для нескольких микросервисов техническую возможность и сделать из нее отдельный микросервис, который смогут использовать другие микросервисы. А иногда, например в случае ERP-интеграции, удастся распознать техническую возможность, неоправданно усложняющую какой-либо микросервис, и сделать из нее отдельный микросервис. В обоих случаях у реализующих бизнес-возможности микросервисов остается на одну техническую задачу меньше.

Принимая решение о реализации технической возможности в отдельном микросервисе, следите за тем, чтобы не нарушить отличительный признак индивидуальной развертываемости. Реализовывать техническую возможность в виде отдельного микросервиса имеет смысл только в том случае, когда этот микросервис можно будет первично и повторно развертывать независимо от других микросервисов. Аналогично развертывание микросервисов, чье функционирование обеспечивается реализующим техническую возможность микросервисом, не должно требовать его повторного развертывания.

Распознавание бизнес-возможностей и основанных на бизнес-возможностях микросервисов представляет собой стратегическую задачу, а распознавание вспомогательных технических возможностей, допускающих реализацию в виде отдельного микросервиса, — вероятностную. Реализовывать ли вспомогательную техническую возможность в виде отдельного микросервиса, зависит от того, что окажется проще в целом. Задайте себе следующие вопросы.

- ☐ Если оставить техническую возможность в микросервисе, чья область действия представляет собой бизнес-возможность, то существует ли опасность утратить возможность заменить данный микросервис с приемлемыми затратами труда?
- ☐ Реализована ли данная техническая возможность в нескольких микросервисах, чьи области действия представляют собой бизнес-возможности?
- ☐ Окажется ли возможно выолнить отдельное развертывание реализующего данную техническую возможность микросервиса?
- ☐ Можно ли будет по-прежнему развертывать отдельно все микросервисы, чьи области действия представляют собой бизнес-возможности, при реализации вспомогательной технической возможности в отдельном микросервисе?

Если вы ответили «да» на последние два вопроса и по крайней мере на один из остальных, то это хороший кандидат на роль микросервиса.

3.3. Что делать, если сложно очертить область действия

Возможно, на данном этапе вам представляется, что определение области действия микросервисов — непростая задача: необходимо правильно определить бизнес-возможности, что требует глубокого понимания предметной области, а также верно оценить степень сложности вспомогательных технических возможностей. И вы правы — это действительно не просто, и вы столкнетесь с ситуациями, в которых непонятно, как правильно определить области действия микросервисов.

Причин отсутствия ясности может быть несколько, в том числе следующие.

- ❑ *Недостаточный уровень понимания предметной области.* Анализ предметной области и формирование глубоких знаний о ней представляет собой задачу непростую и требующую немалых затрат времени. Иногда придется принимать решения относительно области действия микросервиса до того, как у вас появится достаточное для уверенности в правильности этих решений понимание данного бизнеса.
- ❑ *Неразбериха в предметной области.* Относительно предметной области могут оказаться неясными не только вопросы разработки. Иногда не вполне понятно, как работать с данной предметной областью с коммерческой точки зрения. Например, может происходить переход на новые рынки сбыта, требующий освоения новой предметной области. В других случаях существующий рынок сбыта может меняться в результате действий конкурентов или самого предприятия. В любом случае предметная область все время изменяется как с точки зрения разработки, так и с коммерческих позиций, и ваше понимание ее тоже эволюционирует.
- ❑ *Незнание всех нюансов технической возможности.* У вас может не быть доступа ко всей информации обо всем необходимом для реализации данной технической возможности. Например, вам может понадобиться выполнить интеграцию с плохо документированной системой, так что полное понимание того, как это нужно было сделать, придет только по окончании интеграции.
- ❑ *Невозможность оценить степень сложности технической возможности.* Если ранее вам не приходилось реализовывать подобную техническую возможность, может оказаться не просто оценить, насколько сложна будет ее реализация.

Ни одна из этих проблем не означает, что вы потерпели неудачу. Это просто рабочие ситуации, встречающиеся время от времени. Фокус в том, чтобы знать, как в подобных ситуациях продвигаться вперед, несмотря на недостаток ясности. В этом разделе мы обсудим, что делать в сомнительных случаях.

Для начала чуть завышаем размер

Если вы сомневаетесь в том, какой должна быть область действия микросервиса, при определении ее размера лучше ошибиться в большую сторону по сравнению с идеальным. Это может показаться странным, ведь я столько говорил о создании маленьких узкоспециализированных микросервисов и преимуществах их малого размера. И это правда: из малого размера и узкой специализации микросервисов

можно извлечь существенные выгоды. Но не мешает внимательно изучить, что произойдет, если ошибиться, слишком сузив области действия.

Рассмотрим микросервис Special Offers, обсуждавшийся ранее в данной главе. Он воплощает бизнес-возможность Special Offers в POS-системе и включает пять различных бизнес-процессов, как показано на рис. 3.3 и воспроизведено вверху на рис. 3.11. Если вы не уверены в том, каковы правильные границы бизнес-возможности Special Offers, и решаете, что лучше ошибиться, чересчур сузив область действия, то может оказаться, что вы разобьете бизнес-возможность так, как показано внизу на рис. 3.11.

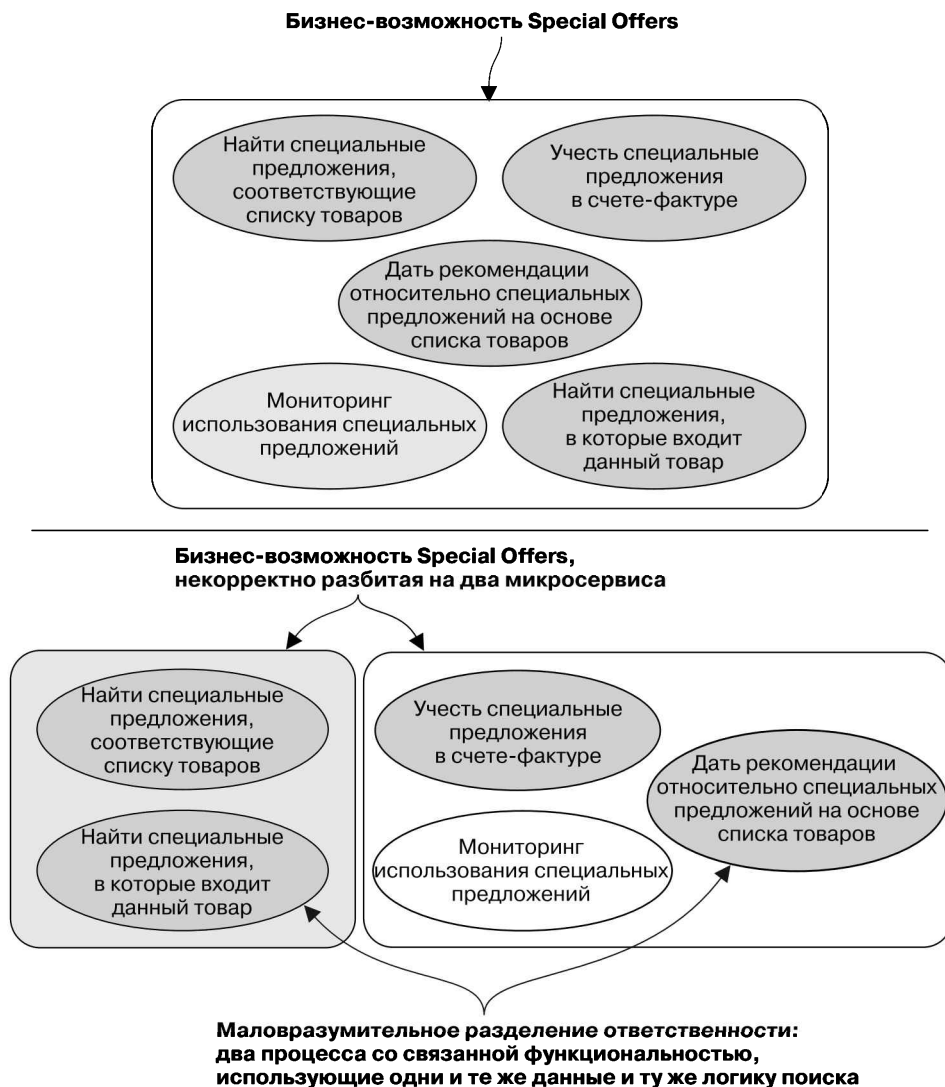


Рис. 3.11. При слишком узкой области действия микросервиса может оказаться, что одна бизнес-возможность разбита на несколько тесно связанных частей

Если область действия будет основана только на части бизнес-возможности Special Offers, это приведет к значительным издержкам.

- ❑ *Дублирование данных и модели данных в двух микросервисах.* Обоим элементам реализации придется хранить все специальные предложения в своих хранилищах данных.
- ❑ *Маловразумительное разделение ответственности.* Одна из частей разделенной бизнес-возможности знает, включен ли данный товар в какие-либо специальные предложения, а другая выдает покупателям специальные предложения на основе их прошлых покупок. Эти две функции тесно связаны, и вы быстро попадете в ситуацию, когда будет сложно понять, к какому микросервису должен относиться тот или иной фрагмент кода.
- ❑ *Сложности при рефакторинге кода бизнес-возможности.* Возникают вследствие того, что код разбросан по базам кода двух микросервисов. Подобный рефакторинг с вовлечением нескольких баз кода представляет собой непростую задачу из-за отсутствия полной картины последствий рефакторинга и слабой поддержки его со стороны IDE.
- ❑ *Невозможность полностью независимого развертывания этих двух микросервисов.* После рефакторинга или реализации какой-либо новой возможности, касающейся обоих микросервисов, может оказаться необходимым развертывать их вместе или в определенном порядке. В любом случае связанность между версиями этих двух микросервисов не соответствует отличительному признаку, состоящему в возможности отдельного развертывания.

Вам придется нести эти издержки с самого начала создания микросервисов вплоть до обретения вами знаний и опыта, достаточных для более точного распознавания бизнес-возможности и области действия микросервиса (в нашем случае равной всей бизнес-возможности Special Offers). Вдобавок сложности рефакторинга и внесения изменений в бизнес-возможности приведут к тому, что вы будете выполнять и то и другое в меньшем объеме, а значит, изучение бизнес-возможности займет больше времени. На протяжении всего этого времени вы будете нести издержки на дублирование данных и их модели, а также издержки из-за невозможности отдельного развертывания.

Мы выяснили, что чрезмерное сужение области действия приводит к созданию тесной связанности между микросервисами. Чтобы понять, предпочтительнее ли ошибиться, определив слишком широкую область действия, выясним, каковы издержки этого подхода.

Допустим, вы задали слишком широкую область действия и теперь область действия микросервиса Special Offers включает обработку купонов (рис. 3.12).

Здесь тоже возникают издержки, связанные со слишком большой областью действия микросервиса.

- ❑ База кода увеличивается и усложняется, что приводит к большим затратам на выполнение изменений.
- ❑ Замена микросервиса становится более сложной задачей.

Эти издержки несомненны, но при довольно малой области действия микросервиса не так уж непомерны. Однако обратите внимание на то, что они быстро растут при увеличении области действия каждого микросервиса и становятся непомерными при близком к монолитной архитектуре размере области действия.



Рис. 3.12. При слишком широкой области действия микросервиса можно прийти к решению включить обработку купонов в бизнес-возможность Special Offers

Тем не менее рефакторинг одной базы кода намного проще рефакторинга сразу двух. При этом намного удобнее экспериментировать и изучать бизнес-возможность на собственном опыте. Если воспользоваться этой возможностью, можно добиться отличного понимания обеих бизнес-возможностей — Special Offers и Coupons — быстрее, чем если бы вы слишком сильно сузили их области действия.

Это утверждение справедливо, когда микросервисы немного великоваты, но перестает быть таковым, если они намного больше, чем должны быть, так что не надо лениться и сваливать в одну кучу несколько бизнес-возможностей в одном микросервисе. Так вы быстро получите огромную плохо контролируемую базу кода, имеющую множество недостатков полноценного монолита.

В общем, чуть большие по размеру, чем должны быть в идеале, микросервисы требуют меньших затрат и более адантивны, чем чуть меньшие. Так что следуйте эмпирическому правилу: лучше ошибиться, слегка нрвысив оптимальный размер области действия.

Смирившись с тем, что вы иногда — а может, и довольно часто — будете не уверены, какую область действия для микросервиса лучше всего выбрать, и что в подобных случаях лучше выбрать чуть большую область действия, вы сможете смириться и с тем, что микросервисы в вашей системе иногда — а может, и довольно часто — будут несколько большего размера, чем должны быть в идеальном случае. Это значит, что вам придется время от времени выделять новые микросервисы из существующих.

Выделение новых микросервисов из существующих

Когда вы обнаруживаете, что один из микросервисов стал слишком большим, но является желание выделить из него новый микросервис. Прежде всего необходимо определить подходящие области действия как для существующего микросервиса, так и для нового. Для этого можно воспользоваться описанными ранее в данной главе факторами.

После определения областей действия необходимо изучить код, чтобы выяснить, согласуется ли его внутреннее устройство с новыми областями действия. Если нет, придется произвести определенный рефакторинг, чтобы достичь этой согласованности. На рис. 3.13 иллюстрирован в укрупненном виде рефакторинг, необходимый для выделения нового микросервиса из уже существующего.

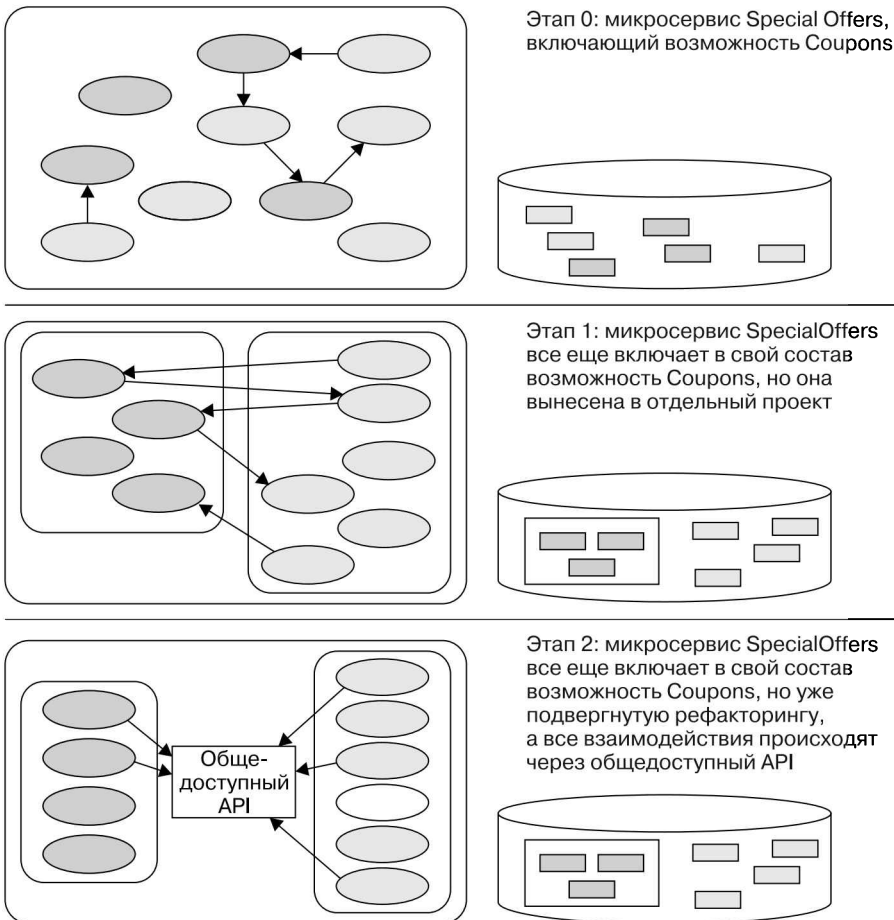


Рис. 3.13. Подготовка к выделению нового микросервиса путем рефакторинга: сначала перемещаем в отдельный проект все относящееся к новому микросервису, после чего направляем все взаимодействия через общедоступный API, аналогичный тому, который будет в конце концов у нового микросервиса

Прежде всего все, что в итоге должно стать частью нового микросервиса, перемещается в отдельную библиотеку классов. Далее все взаимодействия между оставшимся в исходном микросервисе кодом и кодом, который переносится в новый, должны происходить через интерфейс. Этот интерфейс после разделения микросервисов станет частью общедоступного HTTP-интерфейса.

После достижения этапа 2 новый микросервис можно будет отделить от старого, приложив незначительные усилия. Достаточно создать новый микросервис, перенести код, который требует извлечения из существующего микросервиса, в новый и направить все взаимодействия между этими двумя частями через протокол HTTP.

Заранее планируем выделение нового микросервиса

Поскольку вы осознанно приняли решение, что лучше ошибиться в сторону небольшого увеличения размера микросервисов в случае сомнения относительно того, какова должна быть область их действия, у вас появляется возможность заранее предвидеть, какие из микросервисов необходимо будет разделить в дальнейшем. Если непонятно, что какой-то микросервис, вероятно, позже необходимо будет разделить, не мешает заранее спланировать это разделение так, чтобы избавиться от одного или двух из показанных на рис. 3.13 этапов рефакторинга. Такое планирование нередко оказывается вполне возможным.

Часто бывает неясно, является ли конкретная функция отдельной бизнес-возможностью, так что приходится придерживаться описанного ранее эмпирического правила и включать ее в большую бизнес-возможность, реализованную в микросервисе с соответствующим образом увеличенной областью действия. Но при этом нужно отдавать себе отчет, что этот фрагмент *может оказаться* отдельной бизнес-возможностью.

Вернемся к описанию бизнес-возможности Special Offers, включающей процессы для работы с купонами. Сразу было неясно, является ли обработка купонов отдельной бизнес-возможностью, так что бизнес-возможность Special Offers смоделирована так, чтобы включать все показанные на рис. 3.12 процессы.

При первоначальной реализации микросервиса Special Offers с областью действия, соответствующей показанному на рис. 3.12 ее пониманию, еще неизвестно, нужно ли будет в дальнейшем перемещать относящуюся к купонам функциональность в микросервис Coupons. Но сразу понятно, что относящаяся к купонам функциональность не столь близко связана с остальной частью микросервиса, как некоторые из других его частей. А следовательно, отделение кода обработки купонов четкой границей в виде общедоступного API и помещение его в отдельную библиотеку классов, будет неплохой идеей. Это правильный подход к проектированию программного обеспечения, и он вполне окупится потом, когда вам придется выделять код обработки купонов для создания нового микросервиса Coupons.

3.4. Микросервисы с корректно заданной областью действия обладают типичным набором свойств

Мы обсудили определение области действия микросервисов путем распознавания, во-первых, бизнес-возможностей, а во-вторых, вспомогательных технических возможностей. В этом разделе обсудим соответствие этого подхода к определению областей действия следующим четырем отличительным признакам микросервисов, упомянутым в начале главы.

- ❑ Микросервис отвечает за одну возможность.
- ❑ Микросервисы можно развертывать по отдельности.
- ❑ Небольшая команда разработчиков может сопровождать несколько микросервисов.
- ❑ Микросервис можно легко заменить.

ПРИМЕЧАНИЕ

Важно отметить, что взаимосвязь между факторами, определяющими область действия микросервисов, и отличительными признаками микросервисов двусторонняя. Основной и вторичный факторы определения области действия обуславливают то, что микросервисы склонны соответствовать отличительным признакам, но отличительные признаки также помогают понять, правильно ли была определена область действия микросервиса или необходимо поработать с этими факторами еще, чтобы найти более подходящую область действия.

Сведение основных областей действия к бизнес-возможностям повышает качество микросервисов

Основной фактор для определения областей действия микросервисов — распознавание бизнес-возможностей. Посмотрим, как он способствует созданию микросервисов, хорошо соответствующих отличительным признакам микросервисов.

Ответственность за одну возможность

Микросервис, чья область действия определяется отдельной бизнес-возможностью, по определению соответствует первому отличительному признаку микросервисов: он отвечает за отдельную функциональную возможность. Как вы видели в примерах по распознаванию вспомогательных технических возможностей, следует соблюдать осторожность: нет ничего проще, чем возложить слишком много обязанностей на микросервис, чья область действия определяется отдельной бизнес-возможностью. Нужно убедиться, что микросервис реализует только одну бизнес-возможность, а не смесь двух или трех. Лучше также быть аккуратными при распределении вспомогательных технических возможностей по отдельным микросервисам. Но если вы

будете аккуратны, микросервисы, чья область действия определяется отдельной бизнес-возможностью, будут соответствовать первому отличительному признаку микросервисов.

Развертываемость по отдельности

Бизнес-возможности могут осуществляться на предприятии довольно независимыми группами, так что бизнес-возможности сами по себе должны быть достаточно независимыми. В результате микросервисы с соответствующей бизнес-возможностью областью действия тоже довольно независимы. Это не означает отсутствия взаимодействия между ними — объем взаимодействия как посредством прямых обращений между сервисами, так и посредством событий может быть значительным. Суть в том, что взаимодействие происходит через четко описанный общедоступный интерфейс с обратной совместимостью. При качественной реализации характер взаимодействий таков, что другие микросервисы продолжают работать в случае кратковременного сбоя в работе какого-либо микросервиса. Это значит, что качественно реализованные микросервисы, чья область действия определяется отдельной бизнес-возможностью, допускают развертывание по отдельности.

Возможность развертывания и поддержки небольшой командой разработчиков

Бизнес-возможность — нечто такое, с чем может справиться небольшая группа сотрудников предприятия. Это ограничивает ее область действия, а значит, и область действия соответствующего микросервиса. Опять же, если тщательно проверять, реализует ли микросервис только одну бизнес-возможность, а всевозможные технические возможности реализованы в отдельных микросервисах, область действия микросервиса окажется достаточно мала для того, чтобы небольшая команда разработчиков смогла сопровождать как минимум пять микросервисов и быстро заменять микросервисы при необходимости.

Сведение дополнительных областей действия к вспомогательным техническим возможностям повышает качество микросервисов

Дополнительный фактор для определения областей действия микросервисов — распознавание вспомогательных технических возможностей. Посмотрим, как он способствует созданию микросервисов, обладающих отличительными признаками микросервисов.

Ответственность за одну возможность

Аналогично ситуации с микросервисами, чья область действия определяется отдельной бизнес-возможностью, ограничение области действия микросервиса вспомогательной технической возможностью по определению означает соответствие первому отличительному признаку микросервисов: микросервис отвечает за отдельную функциональную возможность.

Развертываемость по отдельности

Прежде чем реализовывать техническую возможность в виде отдельной всномогательной технической возможности в отдельном микросервисе, необходимо спросить себя: можно ли будет выолнить развертывание этого нового микросервиса отдельно? Если ответ на этот вопрос — «нет», реализовывать ее в виде отдельного микросервиса не стоит. Она же микросервис, чья область действия онределяется всномогательной технической возможностью, соответствует второму отличительному признаку микросервисов.

Возможность развертывания и поддержки небольшой командой разработчиков

Онределяемая всномогательной технической возможностью область действия микросервиса обычно довольно узка и четко определена. В то же время смысл реализации подобных возможностей в отдельных микросервисах частично и состоит в том, что они могут быть довольно сложны. Другими словами, микросервисы, чьи области действия онределяются всномогательной технической возможностью, чаще всего небольшого размера, что хорошо согласуется с отличительными признаками микросервисов, говорящими о заменяемости и удобстве сопровождения. Однако их внутренний код может оказаться довольно сложным, что затрудняет их поддержку и замену.

В этой области требуется онределенный компромисс между использованием всномогательных технических возможностей как фактора определения области действия микросервисов, с одной стороны, и отличительными признаками микросервисов — с другой. Если всномогательная техническая возможность оказывается настолько сложной, что микросервис становится трудно заменять, это верный признак того, что пора носмотреть на эту возможность внимательнее и поискать способ дальнейшего ее разбиения. Как было показано в примере с уведомлением покунателей (см. подраздел «Примеры вспомогательных технических возможностей» раздела 3.2), вполне допустимо, чтобы один всномогательный технический микросервис использовал другие «за кулисами».

3.5. Резюме

- ❑ Основной фактор определения области действия микросервисов — распознавание бизнес-возможностей. Бизнес-возможности — то, что организация выолняет для достижения своих бизнес-целей.
- ❑ Для распознавания бизнес-возможностей можно использовать методы предметно-ориентированного проектирования. Предметно-ориентированное проектирование — мощный инструмент для более глубокого и полного понимания предметной области. Подобное понимание позволяет распознавать бизнес-возможности.
- ❑ Дополнительный фактор онределения области действия микросервисов — распознавание вспомогательных технических возможностей. Всномогательная техническая возможность — техническая функция, требующаяся одному или нескольким микросервисам, чьи области действия онределяются бизнес-возможностями.

- ❑ Желательно перемещать вспомогательные технические возможности в отдельные микросервисы только тогда, когда они настолько сложны, что в противном случае будут вызывать проблемы в содержащих их микросервисах, и если их можно развертывать отдельно.
- ❑ Распознавание вспомогательных технических возможностей — вероятностный вид проектирования. Выделять вспомогательные технические возможности в отдельные микросервисы следует только тогда, когда это приводит к упрощению системы в целом.
- ❑ Если вы не уверены в том, какова должна быть область действия микросервиса, лучше сделать ее чуть больше идеальной, а не чуть меньше.
- ❑ Определение области действия микросервисов — непростая задача, и вы, вероятно, иногда будете испытывать сомнения при этом. Вполне возможно, что при первой итерации некоторые из определенных вами областей действия окажутся неправильными.
- ❑ Будьте готовы к тому, что время от времени вам придется выделять новые микросервисы из существующих.
- ❑ Если вы сомневаетесь в том, что правильно определили области действия микросервисов, организуйте их код таким образом, чтобы было удобно потом выделять из них новые микросервисы.

4

Взаимодействие микросервисов

В этой главе:

- ❑ способы взаимодействия микросервисов посредством команд, запросов и событий;
- ❑ сравнение взаимодействия на основе событий с взаимодействием с помощью команд и запросов;
- ❑ реализация ленты событий;
- ❑ реализация взаимодействия на основе команд, запросов и событий.

Каждый микросервис реализует одну возможность, но они должны взаимодействовать, чтобы предоставлять полную функциональность конечному пользователю. Для совместной работы микросервисы могут использовать три основных стиля взаимодействия: *команды* (*commands*), *запросы* (*queries*) и *события* (*events*). У каждого из этих стилей есть свои достоинства и недостатки, и для выбора подходящего для каждого из микросервисов необходимо понимать, каково соотношение выгод и потерь. При правильном выборе стиля взаимодействия появляется возможность реализовать слабо связанные микросервисы с четко определенными границами. В этой главе я приведу примеры кода для реализации всех трех стилей взаимодействия.

4.1. Типы взаимодействия: команды, запросы и события

Микросервисы представляют собой мелко структурированные элементы с узкой областью действия. Для доставки функциональности конечному пользователю микросервисы должны взаимодействовать.

В качестве примера рассмотрим микросервис *Loyalty Program* из POS-системы, описанной в главе 3. Микросервис *Loyalty Program* отвечает за бизнес-возможность *Loyalty Program*. Программа проста: покупатели могут регистрироваться в качестве пользователей программы лояльности, после регистрации они начинают получать уведомления о новых специальных предложениях и зарабатывать бонусные баллы при покупках. Тем не менее бизнес-возможность *Loyalty Program* по-прежнему зависит от других бизнес-возможностей, а другие бизнес-возможности — от нее.

Микросервис Loyalty Program должен взаимодействовать с несколькими другими микросервисами (рис. 4.1).

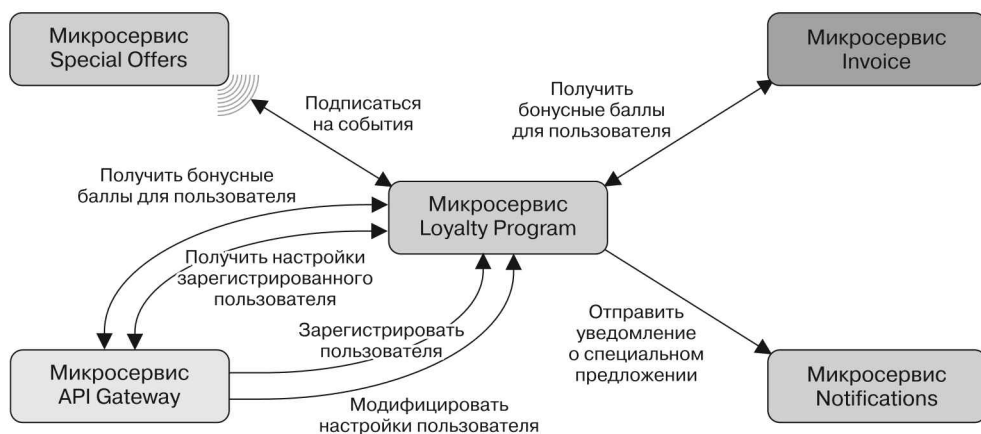


Рис. 4.1. Микросервис Loyalty Program взаимодействует с несколькими другими микросервисами. В одних случаях микросервис Loyalty Program получает запросы от других микросервисов, в других — отправляет им запросы

Как отмечалось в главе 1 при знакомстве со списком отличительных признаков микросервисов, микросервис отвечает за одну возможность, а как говорилось в главе 3, обычно это бизнес-возможность. Предназначенные для конечного пользователя функциональные возможности — сценарии использования — часто включают несколько бизнес-возможностей, так что реализующие эти возможности микросервисы должны взаимодействовать для предоставления конечному пользователю нужной функциональности.

Существует три основных стиля взаимодействия двух микросервисов.

- ❑ **Команды.** Используются, когда одному микросервису необходимо, чтобы другой выполнил какое-либо действие. Например, микросервис Loyalty Program отправляет команду микросервису Notifications, если нужно послать зарегистрированному пользователю уведомление.
- ❑ **Запросы.** Применяются, если одному микросервису необходима информация от другого. Поскольку покупатели, набравшие много бонусных баллов, получают право на скидку, микросервис Invoice запрашивает микросервис Loyalty Program о количестве имеющихся у пользователя бонусных баллов.
- ❑ **События.** Используются, когда микросервису необходимо реагировать на что-либо происходящее в другом микросервисе. Микросервис Loyalty Program подписывается на события микросервиса Special Offers, так что при появлении нового специального предложения он может обеспечить отправку зарегистрированным пользователям уведомлений.

Взаимодействие между двумя микросервисами может задействовать один, два или все три упомянутых стиля взаимодействия. Каждый раз, когда требуется

взаимодействие двух микросервисов, необходимо решить, какой стиль использовать. На рис. 4.2 вновь показаны взаимодействия микросервиса Loyalty Program, но на этот раз с указанием выбранного мпой для каждого из пих стиля взаимодействия.

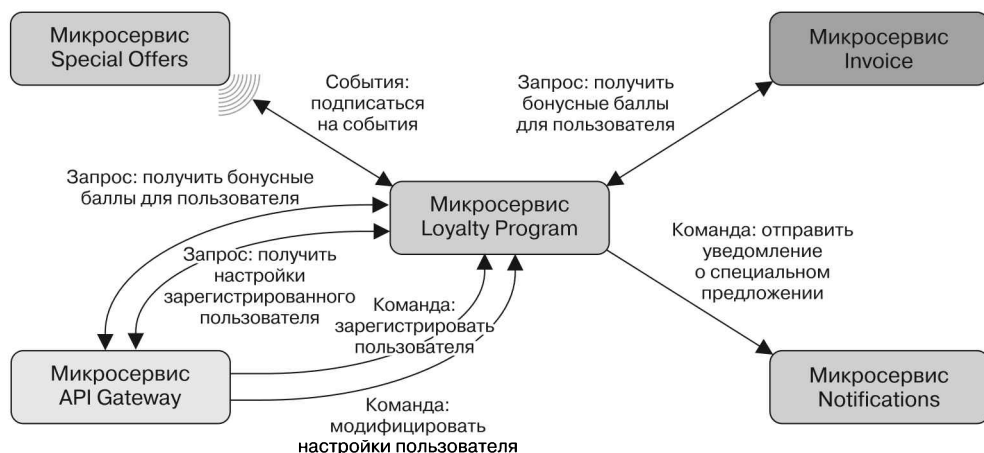


Рис. 4.2. Микросервис Loyalty Program использует все три стиля взаимодействия: команды, запросы и события

При основанпом на командах и запросах взаимодействии используются относительно высокоуровневые команды/запросы. Выполняемые между микросервисами вызовы — удаленные в том смысле, что они как минимум пересекают границу процесса, а обычно и проходят по сети. Соответственно, вызовы между микросервисами выполняются относительно медленно. Несмотря на мелкомодульность микросервисов, не падейтесь, что обращения от одного из пих к другому будут выполняться столь же быстро, как вызовы фупкций в микросервисе.

Более того, основанное на событиях взаимодействие предпочтительнее основанного на командах или запросах. Основанное на событиях взаимодействие связывает микросервисы слабее, чем две остальные формы взаимодействия, поскольку события обрабатываются асинхронно. Это означает отсутствие временной связанности взаимодействующих через события микросервисов: событие не должно обрабатываться сразу же после генерации. Лучше, чтобы обработка события происходила тогда, когда подписчик готов ее выполнить. Напротив, команды и запросы синхронны, а значит, должны обрабатываться сразу же после отправки.

Команды и запросы: синхронное взаимодействие

Как команды, так и запросы представляют собой синхронные формы взаимодействия. И те и другие реализуются в виде HTTP-запросов от одного микросервиса к другому. Запросы реализуются посредством HTTP-запросов типа GET, а команды — посредством HTTP-запросов типа POST или PUT.

Микросервис Loyalty Program может отвечать па запросы о зарегистрировапных пользователях и выполнять комапды по созданию пользователей или модификации дапных о зарегистрировапных пользователях. На рис. 4.3 показаны основанные на комапдах и запросах взаимодействия, в которых участвует микросервис Loyalty Program.

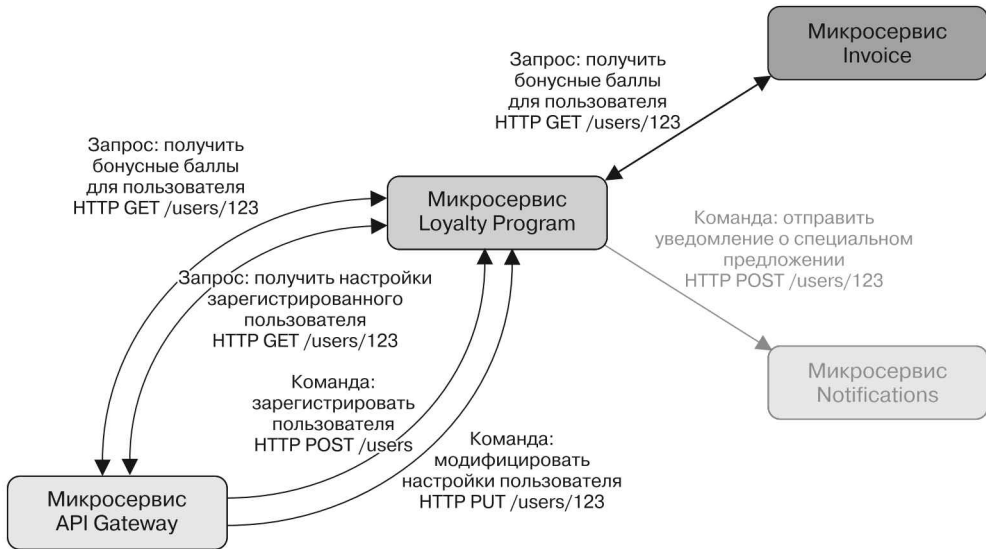


Рис. 4.3. Микросервис Loyalty Program взаимодействует с тремя другими микросервисами с помощью команд и запросов. Запросы реализованы как HTTP-запросы типа GET, а команды реализованы как HTTP-запросы типа POST или PUT. Взаимодействие с помощью команд с микросервисом Notifications закрашено серым цветом, поскольку я не собираюсь демонстрировать его реализацию — оно выполняется точно так же, как и остальные взаимодействия

На рис. 4.3 показаны два запроса: «Получить бонусные баллы для пользователя» и «Получить пастройки зарегистрировапного пользователя». Мы будем использовать для обработки обоих одну и ту же конечную точку, возвращающую представление зарегистрировапного пользователя. Это представление включает как количество бонусных баллов, так и пастройки. Поступим мы так по двум причинам: во-первых, это проще, чем заводить две конечные точки, а во-вторых, выглядит аккуратнее, поскольку микросервису Loyalty Program приходится при этом выдавать только одно представление зарегистрировапного пользователя вместо возни со специализированными форматами для специализированных запросов.

На рис. 4.3 микросервису Loyalty Program отправляются две комапды: одна для реализации нового пользователя, вторая — для обновления данных уже существующего зарегистрировапного пользователя. Первую мы реализуем с помощью HTTP-запроса GET, а вторую — с помощью HTTP-запроса PUT. Это стандартное применение HTTP-методов POST и PUT. Метод POST часто используется для создания нового ресурса, а PUT описан в спецификации протокола HTTP как предназначенный для обновления ресурсов.

В целом микросервис *Loyalty Program* должен предоставлять три конечные точки.

- ❑ Конечная точка HTTP типа GET, располагающаяся по URL вида `/users/{userId}`, возвращает представление пользователя. Эта конечная точка реализует оба запроса, показанных на рис. 4.3.
- ❑ Конечная точка HTTP типа POST, ожидающая получения в теле запроса представления пользователя, регистрирует этого пользователя в программе лояльности.
- ❑ Конечная точка HTTP типа PUT, располагающаяся по URL вида `/users/{userId}`, ожидает получения в теле запроса представления пользователя и обновляет настройки этого уже зарегистрированного пользователя.

Микросервис *Loyalty Program* состоит из таких же стандартных компонентов, какие вы встречали ранее (рис. 4.4). Конечные точки реализованы в компоненте HTTP API.

Другие участники этих взаимодействий — микросервисы, которые почти наверняка имеют ту же стандартную структуру, с добавлением компонента *LoyaltyProgramClient*. Например, на рис. 4.5 показано, как мог бы быть структурирован микросервис *Invoice*.

Ожидаемое микросервисом в командах *Loyalty Program* представление зарегистрированного пользователя, с помощью которого он будет отвечать на запросы, представляет собой сериализацию следующего класса *LoyaltyProgramUser* (листинг 4.1).

Листинг 4.1. Представление пользователя микросервиса *Loyalty Program*

```
public class LoyaltyProgramUser
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int LoyaltyPoints { get; set; }
    public LoyaltyProgramSettings Settings { get; set; }
}

public class LoyaltyProgramSettings
{
    public string[] Interests { get; set; }
}
```

Приведенные в этом коде описания конечных точек и двух классов фактически формируют публикуемый микросервисом *Loyalty Program* контракт. Компонент *LoyaltyProgramClient* микросервиса *Invoice* соблюдает этот контракт при выполнении обращений к микросервису *Loyalty Program* (рис. 4.6).

Команды и запросы — формы взаимодействия, обладающие большими возможностями, но у них есть общий недостаток — синхронная природа. Как уже упоминалось, это приводит к связности между микросервисами, предоставляющими конечные точки, и микросервисами, обращающимися к ним. Далее мы перейдем к обсуждению асинхронного взаимодействия посредством событий.

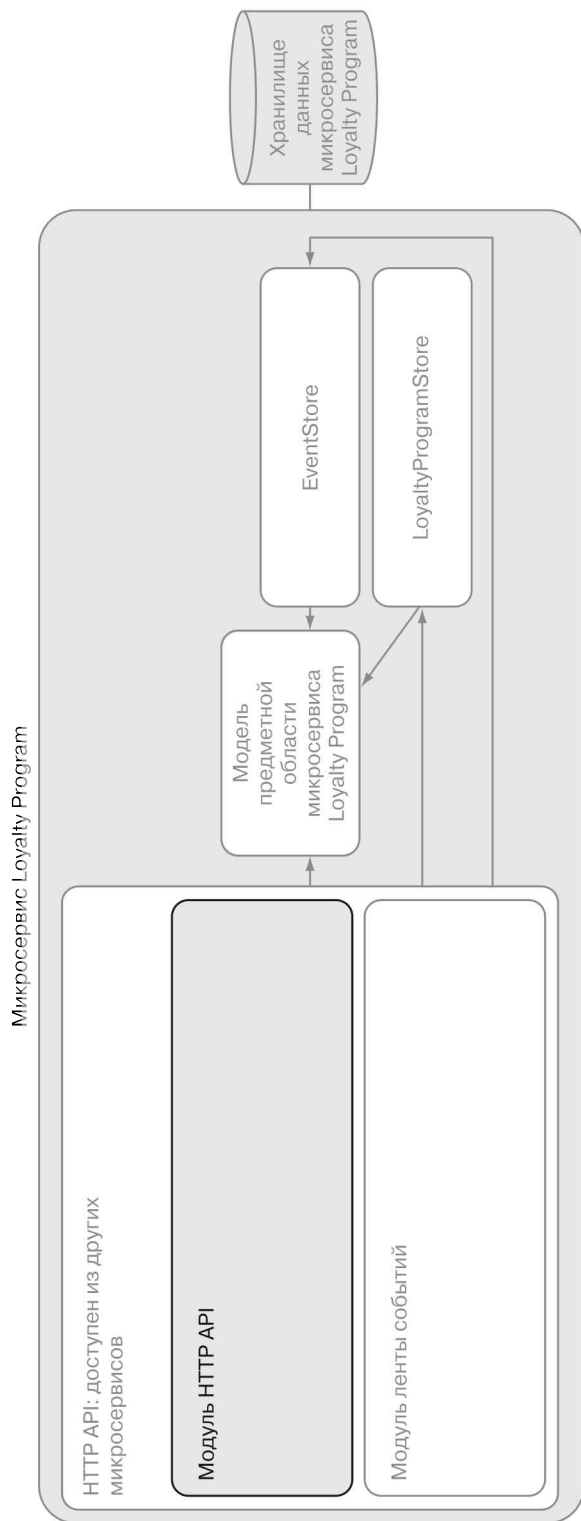


Рис. 4.4. Предоставляемые микросервисом Loyalty Program конечные точки реализованы в компоненте HTTP API

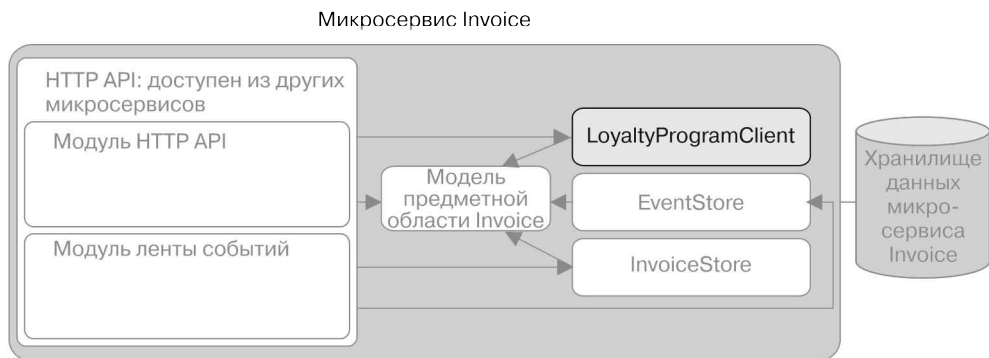


Рис. 4.5. В микросервисе Invoice имеется компонент LoyaltyProgramClient, отвечающий за обращение к микросервису Loyalty Program

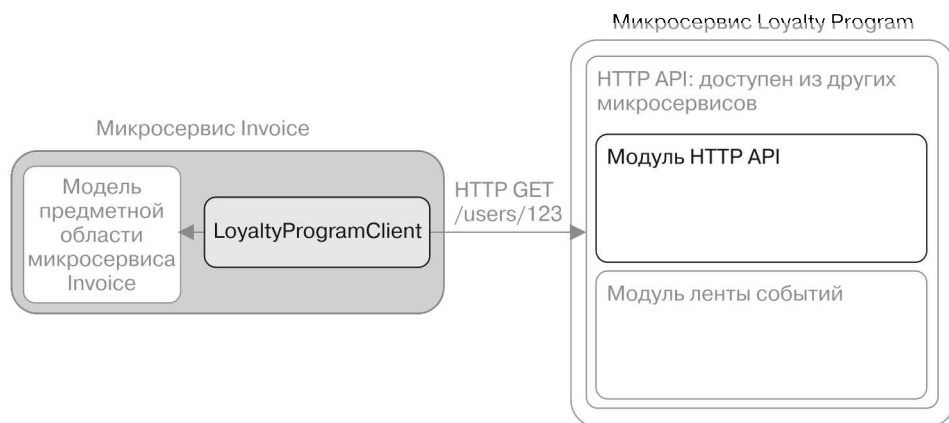


Рис. 4.6. Компонент LoyaltyProgramClient микросервиса Invoice отвечает за выполнение вызовов к микросервису Loyalty Program. Он служит адаптером между публикуемым микросервисом Loyalty Program контрактом и моделью предметной области микросервиса Invoice

События: асинхронное взаимодействие

Взаимодействие на основе событий является асинхронным. Это значит, что публикующий события микросервис не выполняет обращения к подписавшимся на эти события микросервисам. Вместо этого микросервисы-подписчики по мере своей готовности к обработке опрашивают публикующий события микросервис на предмет новых событий. Именно эти опросы я и называю *подпиской* на ленту событий. Хотя сами опросы состоят в выполнении синхронных запросов, взаимодействие остается асинхронным, поскольку публикация событий не зависит от выполнения подписчиками опросов на предмет событий.

На рис. 4.7 можно видеть микросервис Loyalty Program, подписанный на события микросервиса Special Offers. Микросервис Special Offers может публиковать события, когда в его предметной области происходит что-то важное, например, при каждой активизации нового специального предложения. Публикация события в этом контексте означает сохранение этого события в Special Offers. Микросервис Loyalty

Program не увидит этого события до тех пор, пока не выполнит обращение к ленте событий микросервиса Special Offers. Когда это сделать, целиком зависит от микросервиса Loyalty Program. Это может произойти сразу после публикации события или в любой момент времени в будущем.

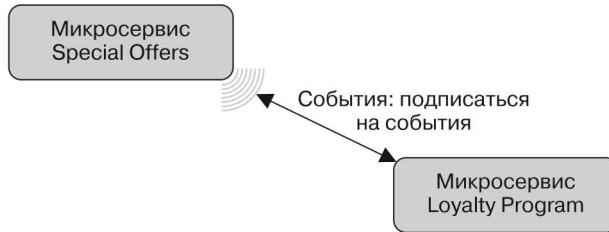


Рис. 4.7. Микросервис Loyalty Program обрабатывает события микросервиса Special Offers тогда, когда это ему удобно

Как и в случае с двумя другими типами взаимодействия, в основанном на событиях взаимодействии участвуют две стороны. Одна сторона — микросервис, публикующий события посредством ленты событий, а вторая — подписывающийся на эти события микросервис.

Предоставление ленты событий

Микросервисы могут публиковать события для других микросервисов с помощью *ленты событий (event feed)*, представляющей собой просто конечную точку HTTP — по адресу `/events`, например, — к которой могут выполнять запросы эти другие микросервисы и из которой они могут получать данные о событиях. На рис. 4.8 продемонстрированы компоненты микросервиса Special Offers. Он содержит стандартный набор компонентов, который вы уже не раз видели. Компоненты, участвующие в реализации ленты событий, на рис. 4.8 выделены.



Рис. 4.8. Лента событий микросервиса Special Offers доступна другим микросервисам по протоколу HTTP и использует хранилище событий

Публикуемые микросервисом Special Offers события хранятся в его базе данных. В компоненте EventStore имеется код для чтения событий из базы данных и записи

их в нее. Код модели предметной области может использовать компонент EventStore для сохранения событий, которые нужно опубликовать. Компонент Event Feed представляет собой реализацию конечной точки HTTP, обеспечивающей доступность события для других микросервисов, а именно конечной точки /events.

Компонент Event Feed использует компонент EventStore для чтения событий из базы данных, после чего возвращает события в теле HTTP-ответа. Подписчики имеют возможность применять параметры строки запроса для управления тем, сколько событий и какие именно будут возвращены.

Подписка на события

По существу, подписка на ленту событий означает опрос конечной точки микросервиса, на который вы подписаны. Время от времени вы отправляете HTTP-запрос типа GET на конечную точку /events, чтобы проверить, не появилось ли каких-то еще не обработанных вами событий.

Рисунок 4.9 представляет собой обзор микросервиса Loyalty Program и демонстрирует, что тот состоит из двух процессов. Мы уже обсуждали веб-процесс, по процесс — подписчик на события для нас пов.

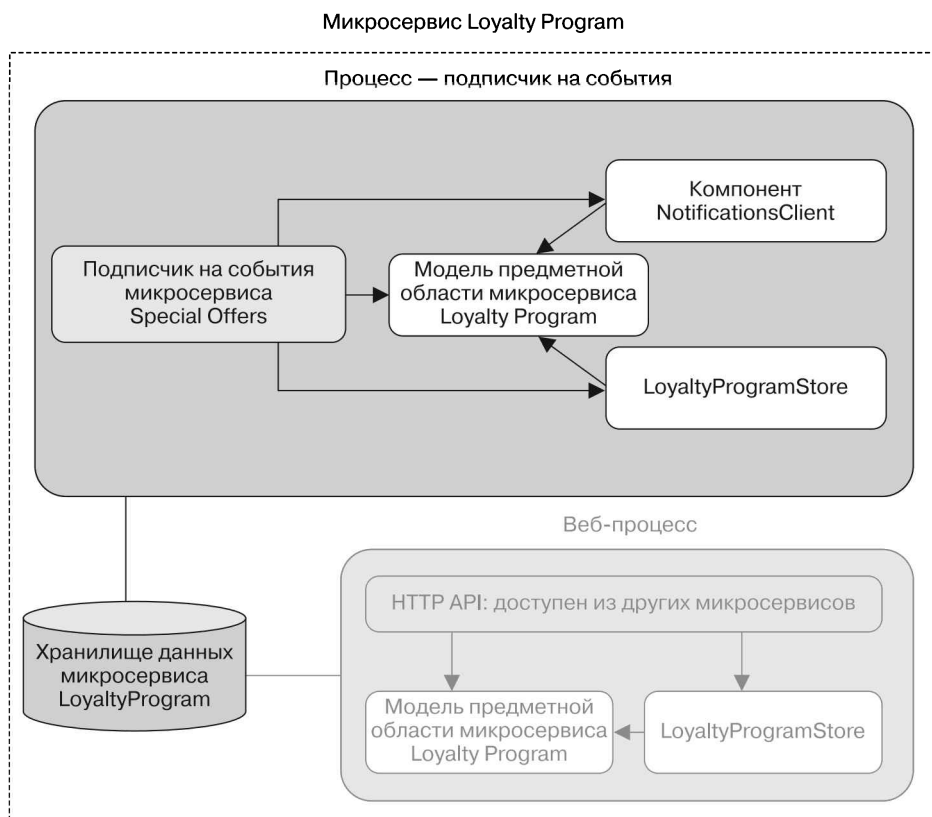


Рис. 4.9. Подпиской на события в микросервисе Loyalty Program занимается процесс — подписчик на события

Процесс — подписчик па события представляет собой фоновый процесс, периодически выполняющий запросы к ленте событий микросервиса Special Offers для получения новых событий. После получения этих событий он обрабатывает их путем отправки команд микросервису Notifications для уведомления зарегистрированных пользователей о новых специальных предложениях. Опрос ленты событий реализован в компоненте SpecialOffersSubscriber, а компонент NotificationsClient отвечает за отправку команд микросервису Notifications.

Именно таким образом реализуются подписки па события: у тех микросервисов, которым требуется подписка па события, имеется процесс-подписчик, опрашивающий ленту событий. При возвращении из ленты новых событий процесс-подписчик обрабатывает события в соответствии с бизнес-правилами.

Очереди событий

Альтернатива публикации событий в ленте событий — использование технологии очереди, например платформы RabbitMQ или шины обслуживания (Service Bus) для Windows Server. При этом подходе публикующие события микросервисы помещают их в очередь, а подписчики читают их из очереди. События необходимо маршрутизировать от их издателя к подписчикам, и конкретная реализация этого зависит от выбранной технологии очереди. Как и в случае с подходом, при котором используется лента событий, у подписывающегося на события микросервиса имеется процесс-подписчик, читающий события из ленты и обрабатывающий их.

Это вполне жизнеспособный подход к реализации взаимодействия между микросервисами на основе событий. Но в данной книге мы будем использовать для взаимодействия на основе событий ленту событий на основе протокола HTTP, поскольку это простое, но вместе с тем надежное и хорошо масштабируемое решение.

Форматы данных

До сих пор мы обмечивались данными в формате JSON. Я упоминал мимоходом, что все конечные точки, которые мы реализовали с помощью фреймворка Nancy, в равной степени поддерживают формат XML (Nancy сразу поставляется с поддержкой сериализации и десериализации как JSON, так и XML). Этих двух вариантов достаточно для большинства случаев, но есть причины, по которым может понадобиться что-либо иное.

- ❑ Если необходимо обмениваться большими объемами данных, может потребоваться более сжатый формат. Текстовые форматы, например JSON или XML, занимают значительно больше места, чем двоичные, такие как Protocol Buffers.
- ❑ Если нужен доступный для чтения человеком, но более структурированный, чем JSON, формат, можно попытаться использовать YAML.
- ❑ Если ваша фирма использует проприетарный формат данных, может потребоваться его поддержка.

Во всех этих случаях вам нужны будут конечные точки, способные принимать данные в отличных от JSON или XML форматах и отвечать в этих же форматах. В качестве примера: запрос для регистрации пользователя с помощью микросер-

веса Loyalty Program с использованием формата YAML в теле запроса выглядит следующим образом:

```
POST /users HTTP/1.1
Host: localhost:5000
Accept: application/yaml
Content-Type: application/vyaml

Name: Christian
Settings:
  Interests:
    - whisky
    - cycling
    - software design
```

Ответ на этот запрос также использует формат YAML:

```
HTTP/1.1 201 Created
Content-Type: application/yaml
Location: http://localhost:5000/users/1
Id: 1

Name: Christian
Settings:
  Interests:
    - whisky
    - cycling
```

Тела предшествующих запроса и ответа имеют формат YAML, и в обоих случаях это указано в заголовке `Content-Type`. Чтобы запросить ответ тоже в формате YAML, в запросе используется заголовок `Accept`. Этот пример демонстрирует взаимодействие микросервисов с помощью различных форматов данных, а также применение ими заголовков HTTP для описания того, какие форматы используются.

4.2. Реализация взаимодействия

В этом разделе вы научитесь писать код для взаимодействий, приведенных на рис. 4.2. В качестве отправной точки мы возьмем микросервис Loyalty Program, рассмотрим и некоторые из работающих совместно с ним микросервисов — API Gateway, Invoice и Special Offers, — чтобы показать обе взаимодействующие стороны.

Реализация взаимодействия включает в себя три этапа.

1. Создание проекта для микросервиса Loyalty Program. Точно так же, как и раньше, нужно создать пустое приложение ASP.NET 5 и добавить в него Nancy. Единственное отличие заключается в том, что придется добавить немного кода Nancy.
2. Реализация основного набора команд и запросов взаимодействия показана на рис. 4.2. Мы реализуем все команды и запросы, которые может обрабатывать микросервис Loyalty Program, а также использующий их код во взаимодействующих с ним микросервисах.

3. Реализация основанного на событиях взаимодействия показана на рис. 4.2. Мы начнем с лепты событий в микросервисе Special Offers, после чего перейдем к реализации подписки в микросервисе Loyalty Program. В процессе этого добавим дополнительный проект — и дополнительный процесс — в микросервис Loyalty Program. После этого все взаимодействия микросервиса Loyalty Program окажутся реализованы.

Микросервис Loyalty Program состоит из веб-процесса со структурой, аналогичной той, которую вы уже видели (рис. 4.10, *внизу*). Далее, реализовав взаимодействие на основе событий, добавим еще один процесс, который мы называем *процессом — подписчиком на события* (*event-subscriber process*) (см. рис. 4.10, *вверху*).

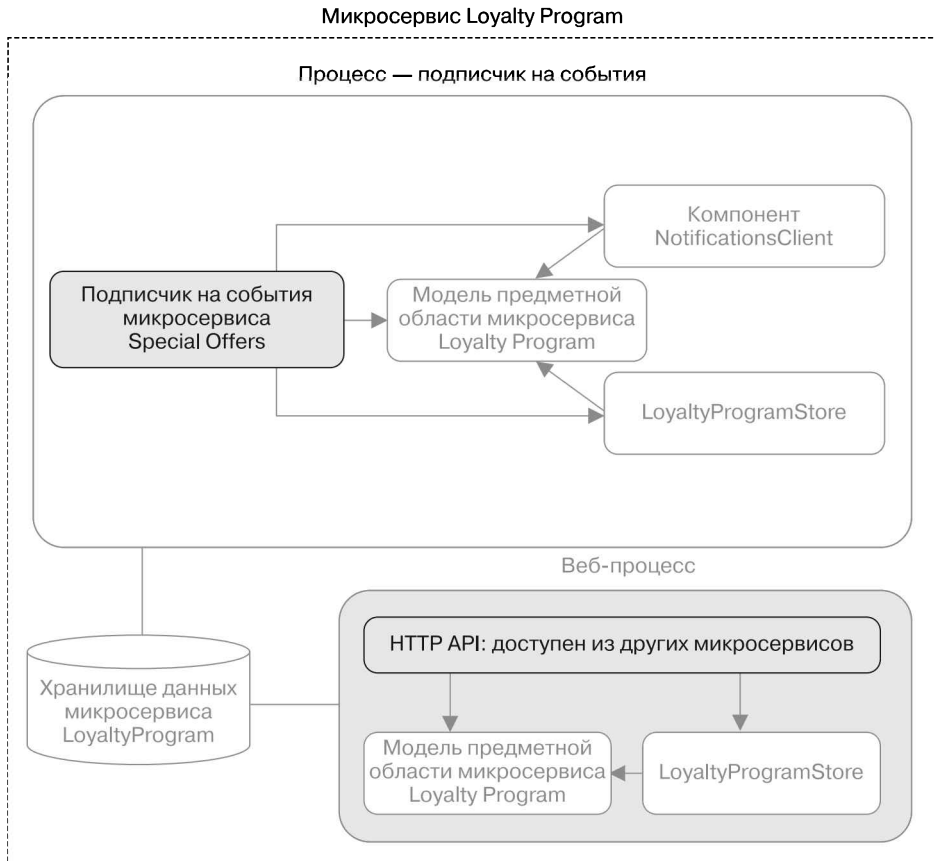


Рис. 4.10. У микросервиса Loyalty Program имеется веб-процесс, соответствующий приведенной ранее структуре, и процесс — подписчик на события, обрабатывающий подписки на события микросервиса Special Offers. В этой главе будет приведен только код для выделенных на этом рисунке компонентов

Чтобы не отвлекаться от рассмотрения взаимодействия, не будем демонстрировать весь код микросервиса Loyalty Program. Вместо этого я включу код HTTP API

в веб-процесс, а код подписчика па события о специальных предложениях — в процесс — подписчик па события.

Создание проекта для микросервиса Loyalty Program

Для реализации микросервиса Loyalty Program прежде всего необходимо создать пустое приложение ASP.NET 5 и добавить в него фреймворк Nancy в виде пакета NuGet. Вы уже несколько раз это делали в главах 1 и 2, так что здесь я не буду останавливаться па подробностях этого процесса.

Но па этот раз нужно сделать еще одно: переопределить метод обработки фреймворком Nancy ответов с кодом состояния 404 Not Found (404 Не найдено). По умолчанию Nancy помещает HTML-код для страницы ошибки в тело ответа 404 Not Found (404 Не найдено). Но поскольку клиентские приложения микросервиса Loyalty Program — не браузеры, а другие микросервисы, страница ошибки не пужпа, лучше отправить ответ с кодом состояния 404 Not Found (404 Не пайдено) и пустым телом ответа. С этой целью добавьте в проект файл с пазванием **Bootstrapper.cs** (листинг 4.2). В нем необходимо разместить следующий класс, последующий класс **DefaultNancyBootstrapper**.

Листинг 4.2. Загрузчик Nancy

```
namespace LoyaltyProgram
{
    using System;
    using Nancy;
    using Nancy.Bootstrapper;

    public class Bootstrapper : DefaultNancyBootstrapper
    {
        protected override
            Func<ITypeCatalog, NancyInternalConfiguration> InternalConfiguration =>
                NancyInternalConfiguration
                    .WithOverrides(builder => builder.StatusCodeHandlers.Clear());
    }
}
```

Убрать все заданные по умолчанию обработчики кода состояния,
чтобы они не вносили изменений в ответы

Фреймворк Nancy автоматически пайдет этот класс при загрузке, вызовет геттер **InternalConfiguration** и воспользуется возвращенной им конфигурацией. Мы используем конфигурацию по умолчанию, не считая исключения всех **StatusCodeHandlers**, а зпачит, убираем все, что может внести изменения в ответ из-за его кода состояния.

Загрузчик фреймворка Nancy

Фреймворк Nancy использует загрузчик (bootstrapper) во время запуска приложения для конфигурации как самого фреймворка, так и приложения. Nancy разрешает приложениям менять настройки всего фреймворка и заменять в своем загрузчике любую часть Nancy своей собственной реализацией. В этом отношении Nancy — открытый и гибкий фреймворк. Во многих случаях необходимости в настройке фреймворка нет — значения

по умолчанию фреймворка Nancy достаточно разумны, и даже когда такая необходимость есть, заменять целые части Nancy обычно не требуется.

Все, что необходимо сделать для создания загрузчика, — создать класс, реализующий интерфейс `INancyBootstrapper`, и Nancy его обнаружит и задействует. Обычно этот интерфейс не реализуется непосредственно, поскольку, хотя он сам по себе довольно прост, полнофункциональная его реализация отнюдь не проста. Вместо непосредственной реализации интерфейса `INancyBootstrapper` можно воспользоваться поставляемым с фреймворком загрузчиком по умолчанию (`DefaultNancyBootstrapper`) и расширить его. В этом классе имеется множество виртуальных методов, которые можно переопределить и подключиться к различным частям Nancy. Среди них, например, методы для настройки специализированной сериализации и десериализации, методы для добавления обработчиков ошибок и многое другое.

Нам неоднократно придется использовать загрузчик Nancy в этой книге, но в большинстве случаев можно будет спокойно применять значения по умолчанию фреймворка Nancy. Если в приложении отсутствует загрузчик Nancy, то Nancy использует загрузчик, заданный по умолчанию, — `DefaultNancyBootstrapper`.

Реализация команд и запросов

Наш веб-проект готов к внедрению в него реализаций предоставляемых микросервисом `Loyalty Program` конечных точек. Как уже упоминалось, речь идет о следующих конечных точках:

- ❑ о конечной точке HTTP типа `GET`, располагающейся по URL вида `/users/{userId}` и возвращающей представление пользователя. Эта конечная точка реализует оба запроса па рис. 4.3;
- ❑ конечной точке HTTP типа `POST`, располагающейся по URL `/users/`, ожидающей получения в теле запроса представления пользователя и регистрирующей этого пользователя в программе лояльности;
- ❑ конечной точке HTTP типа `PUT`, располагающейся по URL вида `/users/{userId}`, ожидающей получения в теле запроса представления пользователя и обновляющей данные этого уже зарегистрированного пользователя.

Мы реализуем сначала конечные точки для команд, а затем конечную точку для запроса.

Реализация команд с помощью методов POST и PUT протокола HTTP

Код, необходимый для реализации обработки в микросервисе `Loyalty Program` двух команд — `POST` для регистрации нового пользователя и `PUT` для обновления существующего, — аналогичен коду, который вы видели в главе 2. Мы начнем с реализации обработчика для команды регистрации пользователя. Запрос, который необходимо выполнить к микросервису `Loyalty Program` для регистрации пользователя, показан в листинге 4.3.

Листинг 4.3. Запрос для регистрации пользователя по имени Кристиан (Christian)

```

POST /users HTTP/1.1
Host: localhost:5000
Content-Type: application/json
Accept: application/json

{
  "id":0,
  "name":"Christian",
  "loyaltyPoints":0,
  "settings":{"interests" : ["whisky", "cycling"] }
}

```

← JSON-представление регистрируемого пользователя

Чтобы обработать эту команду для регистрации нового пользователя, необходимо добавить в микросервис Loyalty Program модуль фреймворка Nancy посредством добавления файла UserModule.cs со следующим кодом (листинг 4.4).

Листинг 4.4. Конечная точка типа POST для регистрации пользователей

```

using System.Collections.Generic;
using Nancy;
using Nancy.ModelBinding;

public class UsersModule : NancyModule
{
    public UsersModule() : base("/users")
    {
        Post("/", _ =>
        {
            var newUser = this.Bind<LoyaltyProgramUser>();
            this.AddRegisteredUser(newUser);
            return this.CreatedResponse(newUser);
        });
    }

    private dynamic CreatedResponse(LoyaltyProgramUser newUser)
    {
        return
            this.Negotiate
                .WithStatusCode(HttpStatusCode.Created)
                .WithHeader(
                    "Location",
                    this.Request.Url.SiteBase + "/users/" + newUser.Id)
                .WithModel(newUser);
    }

    private void AddRegisteredUser(LoyaltyProgramUser newUser)
    {
        // store the newUser to a data store
    }
}

```

Тело запроса должно включать экземпляр класса LoyaltyProgramUser. В противном случае формат запроса некорректен

←

Negotiate — точка входа текущего API создания ответов фреймворка Nancy

Используем в ответе код состояния 201 Created (201 Создано)

←

Добавляем в ответ заголовок Location (Местоположение), поскольку так предполагается по протоколу HTTP для ответов с кодом состояния 201 Created (201 Создано)

Возвращаем для удобства пользователя в ответе

Ответ на приведенный запрос выглядит следующим образом:

```

Используем в ответе код состояния
201 Created ( 201 Создано)
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Location: http://localhost:5000/users/4

{
  "id": 4,
  "name": "Christian",
  "loyaltyPoints": 0,
  "settings": { "interests": ["whisky", "cycling"]
}
  
```

Механизм согласования содержимого фреймворка Nancy задает заголовок Content-Type

Заголовок Location указывает на только что созданный ресурс

Главное, что нового для нас в листинге 4.4, — использование `Negotiate` для создания ответа на команду. `Negotiate` — свойство класса `NancyModule`, применяемого нами в качестве базового класса для `UserModule`. В данном случае оно представляет собой в основном точку входа для текущего API фреймворка Nancy но созданию ответов. В обработчике мы используем этот API для задания кода состояния, добавления заголовка `Location` и добавления в ответ объекта пользователя. Этот API предоставляет и другие возможности для обработки ответов, например возможность задания других заголовков и указания представления, которое будет использоваться при ответе на запросы, требующие HTML в заголовке `Accept`.

Свойство `Negotiate` также вызывает срабатывание функций фреймворка Nancy, предназначенных для согласования содержимого. Согласование содержимого — то, как в HTTP задается механизм определения формата данных в ответах. По существу, оно означает чтение заголовка `Accept` запроса и сериализацию ответа в указанный там формат. В листинге 4.3 этот заголовок имеет следующий вид: `Accept: application/json`, то есть в ответе данные будут сериализованы в формат JSON.

Закончив с обработчиком для команды регистрации пользователей, обратим внимание на реализацию обработчика для команды обновления данных пользователя. Этот обработчик тоже добавляется в класс `UserModule` (листинг 4.5).

Листинг 4.5. Конечная точка типа PUT для регистрации пользователей

```

public class UsersModule : NancyModule
{
    public UsersModule() : base("/users")
    {
        Post("/", _ => ...);

        Put("/{userId:int}", parameters =>
        {
            int userId = parameters.userId;
            var updatedUser = this.Bind<LoyaltyProgramUser>();
            // store the updatedUser to a data store
            return updatedUser;
        });
    }
    ...
}
  
```

Сохраняем updatedUser в хранилище данных

Ничего нового для себя в этом коде вы не увидите.

Обработчики команд — лишь одна сторона взаимодействия. Другая сторона — отправляющий команды код. На рис. 4.2 можно видеть, что микросервис API Gateway отправляет команды микросервису Loyalty Program. Здесь мы не будем приводить микросервис API Gateway полностью, но в доступных для скачивания примерах кода для данной главы вы можете найти консольное приложение, которое ведет себя так, как вел бы себя микросервис API Gateway применительно к взаимодействию с микросервисом Loyalty Program. Сосредоточимся лишь на отправляющем команды коде.

В микросервисе API Gateway создадим класс `LoyaltyProgramClient`, который будет отвечать за взаимодействие с микросервисом Loyalty Program. Этот класс инкапсулирует всю функциональность, участвующую в построении HTTP-запросов, сериализации данных для запросов, разборе HTTP-ответов и десериализации данных ответов.

Используемый для отправки команды регистрации пользователя код принимает на входе экземпляр класса `LoyaltyProgramUser` и создает HTTP-запрос типа POST с объектом `LoyaltyProgramUser` в теле, который отправляет микросервису Loyalty Program. Проверив код состояния ответа и убедившись, что он равен 201 Created (Создано), он десериализует тело ответа в тип `LoyaltyProgramUser` и возвращает его. Если код состояния не такой, метод возвращает `null`. Реализация показана в листинге 4.6.

Листинг 4.6. Регистрация новых пользователей микросервисом API Gateway

```
using System;
using System.Text;
using System.Threading.Tasks;
using System.Net;
using System.Net.Http;
using Newtonsoft.Json;

public class LoyaltyProgramClient
{
    public async Task<LoyaltyProgramUser>
        RegisterUser(LoyaltyProgramUser newUser)
    {
        using(var httpClient = new HttpClient())
        {
            httpClient.BaseAddress = new Uri($"http://{this.hostName}");
            var response = await
                httpClient.PostAsync(
                    ← Отправляет команду микросервису
                    "/users/",                               LoyaltyProgram
                    new StringContent(
                        JsonConvert.SerializeObject(newUser), ← Сериализует newUser
                        Encoding.UTF8,                          в формат JSON
                        "application/json"));
            ThrowOnTransientFailure(response);
            return JsonConvert.DeserializeObject<LoyaltyProgramUser>(
                ← Задает заголовок Content-Type
                await response.Content.ReadAsStringAsync());
        }
    }
    ← Десериализует ответ, если команда была обработана успешно
}
```

Аналогично в классе `LoyaltyProgramClient` имеется метод для отправки команды обновления пользователя. Этот метод тоже инкапсулирует HTTP-взаимодействия, необходимые для отправки команды (листинг 4.7).

Листинг 4.7. Обновление данных пользователей микросервисом API Gateway

```
public async Task<LoyaltyProgramUser> UpdateUser(LoyaltyProgramUser user)
{
    using(var httpClient = new HttpClient())
    {
        httpClient.BaseAddress = new Uri($"http://{this.hostName}");
        var response = await
            httpClient.PutAsync(
                $"/users/{user.Id}",
                new StringContent(
                    JsonConvert.SerializeObject(user),
                    Encoding.UTF8,
                    "application/json"));
        ThrowOnTransientFailure(response);
        return JsonConvert.DeserializeObject<LoyaltyProgramUser>(
            await response.Content.ReadAsStringAsync());
    }
}
```

← Отправляет команду обновления пользователя в виде запроса типа PUT

Этот код аналогичен коду команды регистрации пользователя, за исключением того, что HTTP-запрос использует метод PUT. Реализовав в микросервисе Loyalty Program обработчики команд, а в микросервисе API Gateway — компонент `LoyaltyProgramClient`, мы реализовали взаимодействие на основе команд. Микросервис API Gateway уже умеет регистрировать пользователей и обновлять данные о них, но пока не может выполнять запросы.

Реализация запросов с помощью метода GET протокола HTTP

Микросервис Loyalty Program уже может обрабатывать нужные команды, но еще не умеет отвечать на запросы относительно зарегистрированных пользователей. Напомню, что для обработки запросов ему требуется только одна конечная точка. Как уже упоминалось, конечная точка для обработки запросов — это конечная точка типа GET, располагающаяся по URL вида `/users/{userId}` и возвращающая представление пользователя (листинг 4.8). Эта конечная точка реализует оба запроса, приведенные на рис. 4.3.

Листинг 4.8. Конечная точка типа GET для запроса данных пользователя по ID

```
public class UsersModule : NancyModule
{
    private static IDictionary<int, LoyaltyProgramUser> registeredUsers =
        new Dictionary<int, LoyaltyProgramUser>();

    public UsersModule() : base("/users")
    {

```

```

Post("/", _ => ...);

Put("/{userId:int}", parameters => ...);

Get("/{userId:int}", parameters =>
{
    int userId = parameters.userId;
    if (registerUsers.ContainsKey(userId))
        return registerUsers[userId];
    else
        return HttpStatusCode.NotFound;
});
}
...
}

```

В этом коде нет ничего такого, чего бы вы не видели ранее. Код, который необходимо добавить в микросервис API Gateway для выполнения запросов к этой конечной точке, тоже не должен стать для вас неожиданностью:

```

public class LoyaltyProgramClient
{
    ...
    public async Task<LoyaltyProgramUser> QueryUser(int userId)
    {
        var userResource = $"/users/{userId}";
        using (var httpClient = new HttpClient())
        {
            httpClient.BaseAddress = new Uri($"http://{this.hostName}");
            var response = await httpClient.GetAsync(userResource);
            ThrowOnTransientFailure(response);
            return JsonConvert.DeserializeObject<LoyaltyProgramUser>(
                await response.Content.ReadAsStringAsync());
        }
    }
}

```

Вот и все, что требуется для взаимодействия на основе запросов. На этом реализация взаимодействия на основе команд и запросов микросервиса Loyalty Program завершена.

Форматы данных

Допустим, необходимо, чтобы только что реализованные нами конечные точки поддерживали формат YAML. Реализовывать поддержку еще одного формата данных в обработчиках конечных точек не рекомендуется — это техническая функция, а не часть логики приложения.

Фреймворк Nancy позволяет организовать поддержку десериализации в другие форматы путем реализации интерфейса `IBodyDeserializer`. Как обычно в фреймворке Nancy, все реализации этого интерфейса подгружаются при запуске приложения и подключаются в привязку модели фреймворка Nancy. Аналогично для поддержки сериализации тела ответа в какой-либо третий формат можно реализовать интерфейс

`IResponseProcessor`, который Nancy также автоматически обнаруживает и подключает к своему процессу согласования содержимого.

Для реализации поддержки формата YAML в микросервисе `Loyalty Program` необходимо сначала установить в проект пакет `YamlDotNet` системы управления пакетами NuGet. Далее вам понадобится добавить файл `YamlSerializerDeserializer.cs`. Мы будем использовать этот файл для реализации как сериализации, так и десериализации. Десериализация выглядит следующим образом (листинг 4.9).

Листинг 4.9. Десериализация из формата YAML

```
namespace LoyaltyProgram
{
    using System.IO;
    using Nancy.ModelBinding;
    using Nancy.Responses.Negotiation;
    using YamlDotNet.Serialization;

    public class YamlBodyDeserializer : IBodyDeserializer
    {
        public bool CanDeserialize(
            MediaRange mediaRange, BindingContext context)
            => mediaRange.Subtype.ToString().EndsWith("yaml");

        public object Deserialize(
            MediaRange mediaRange, Stream bodyStream, BindingContext context)
        {
            var yamlDeserializer = new Deserializer();
            var reader = new StreamReader(bodyStream);
            return yamlDeserializer.Deserialize(
                reader, context.DestinationType);
        }
    }
}
```

Указываем фреймворку Nancy, какие типы содержимого может обрабатывать десериализатор

Пытаемся десериализовать тело запроса в нужный для кода приложения тип данных

В этом коде для десериализации данных из тела запроса в основном используется библиотека `YamlDotNet`.

Реализация поддержки сериализации не так проста, но это все равно лишь вопрос реализации двух методов и одного свойства (листинг 4.10).

Листинг 4.10. Сериализация в формат YAML

```
namespace LoyaltyProgram
{
    using System;
    using System.Collections.Generic;
    using System.IO;
    using Nancy;
    using Nancy.Responses.Negotiation;
    using YamlDotNet.Serialization;
    ...

    public class YamlBodySerializer : IResponseProcessor
    {
        public IEnumerable<Tuple<string, MediaRange>> ExtensionMappings
```

```

{
    get
    {
        yield return new Tuple<string, MediaRange>(
            "yaml", new MediaRange("application/yaml"));
    }
}

public ProcessorMatch CanProcess(
    MediaRange requestedMediaRange, dynamic model, NancyContext context)
=>
    requestedMediaRange.Subtype.ToString().EndsWith("yaml")
    ? new ProcessorMatch
        {
            ModelResult = MatchResult.DontCare,
            RequestedContentTypeResult = MatchResult.NonExactMatch
        }
    : ProcessorMatch.None;

public Response Process(
    MediaRange requestedMediaRange, dynamic model, NancyContext context)
=>
    new Response
    {
        Contents = stream =>
        {
            var yamlSerializer = new Serializer();
            var streamWriter = new StreamWriter(stream);
            yamlSerializer.Serialize(streamWriter, model);
            streamWriter.Flush();
        },
        ContentType = "application/yaml"
    };
}

```

Указываем фреймворку Nancy, какие расширения файлов может обрабатывать этот процессор обработки ответов. Мы не используем эту возможность

Указываем фреймворку Nancy, что данный процессор может обрабатывать значения заголовка Ассерпт, оканчивающиеся на yaml

Создаем новый объект ответа, который будет использоваться в остальных частях конвейера Nancy

Описываем функцию для записи тела ответа в поток данных

Записываем сериализованный в формат YAML объект в используемый фреймворком Nancy для тела ответа поток данных

Библиотека YamlDotNet тоже выполняет сериализацию. Методы `ExtensionMappings` и `CanProcess` класса `YamlBodySerializer` сообщают Nancy, для каких ответов следует ее выполнять. В коде метода `Process` создается ответ с сериализованным в формат YAML телом. Для дальнейшей обработки этого ответа можно дополнительно адаптировать его с помощью кода в обработчике. Например, ответ на команду регистрации пользователя можно создать следующим образом:

```

return
    this.Negotiate
        .WithStatusCode(HttpStatusCode.Created)
        .WithHeader(
            "Location",
            this.Request.Url.SiteBase + "/users/" + newUser.Id)
        .WithModel(newUser);

```

Этот код адаптирует ответ с помощью методов расширения `.With*`. После создания ответа, включая тело в формате YAML, классом `YamlBodySerializer` его можно дополнительно адаптировать под конкретную задачу с помощью методов

withStatusCode и withHeader. Как видите, для поддержки дополнительного формата данных микросервисами на основе фреймворка Nancy достаточно реализовать интерфейсы `IBodyDeserializer` и `IResponseProcessor`.

Реализация взаимодействия на основе событий

Разобравшись с реализацией взаимодействия микросервисов на основе команд и запросов, мы можем заняться взаимодействием на основе событий. На рис. 4.11 снова приведены взаимодействия, в которых участвует микросервис `Loyalty Program`. Микросервис `Loyalty Program` подписывается на события от микросервиса `Special Offers` и использует события, чтобы определить, когда нужно уведомлять зарегистрированных пользователей о новых специальных предложениях.

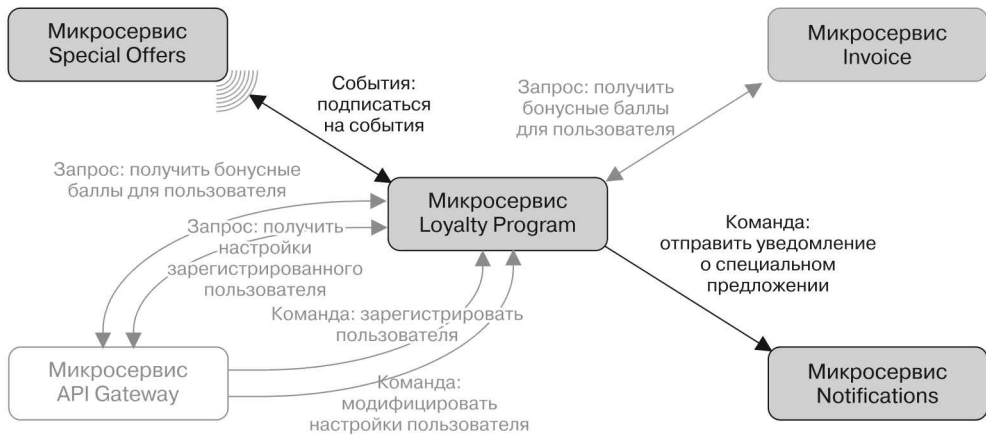


Рис. 4.11. Основанное на событиях взаимодействие микросервиса `Loyalty Program` — подписка на ленту событий микросервиса `Special Offers`

Рассмотрим сначала, каким образом микросервис `Special Offers` выдает события в ленте. После этого вернемся к микросервису `Loyalty Program` и добавим в него второй процесс, который будет отвечать за подписку на события и их обработку.

Реализация ленты событий

Мы уже видели простую ленту событий в главе 2. Микросервис `Special Offers` реализует свою ленту событий аналогичным образом: предоставляет конечную точку `/events`, возвращающую список последовательно пронумерованных событий. Конечная точка может принимать на входе два параметра строки запроса, `start` и `end`, задающие диапазон событий. Например, запрос к ленте событий может выглядеть следующим образом:

```
GET /events?start=10&end=110 HTTP/1.1
```

```
Host: specialoffers.mycompany.com
Accept: application/json
```

Ответ на этот запрос может быть следующим (не считая того, что я обрезал остаток ответа после второго события):

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

```
[
  {
    "sequenceNumber": 10,
    "occuredAt": "2015-10-02T18:37:00.7070659+00:00",
    "name": "NewSpecialOffer",
    "content": {
      "offerId": 123,
      "offer": {
        "productCatalogueId": 1,
        "productName": "Классическая футболка",
        "description": "Потрясающая футболка за полцены!",
      }
    }
  },
  {
    "sequenceNumber": 11,
    "occuredAt": "2015-10-02T20:01:00.3050629+00:00",
    "name": "UpdatedSpecialOffer",
    "content": {
      "offerId": 124,
      "offer": {
        "productCatalogueId": 10,
        "productName": "Чашка горячего чая",
        "description": "Чашка чая. Ведь вы этого хотите.",
        "update": "Теперь на 10 % больше"
      }
    }
  }
]
```

Обратите внимание, что названия событий различаются (**NewSpecialOffer** и **UpdatedSpecialOffer**) и список полей данных двух разновидностей событий неодинаков. Это нормально: различные события несут различную информацию. Однако это нужно учесть при реализации процесса подписчика в микросервисе Loyalty Program. Нельзя ожидать, что конфигурация всех событий будет одинакова.

Реализация конечной точки `/events` в микросервисе **Special Offers** представляет собой простой модуль **Nancy**, точно такой же, как в главе 2 (листинг 4.11).

Листинг 4.11. Конечная точка, которая читает события и возвращает их

```
namespace SpecialOffers.EventFeed
{
    using Nancy;

    public class EventsFeedModule : NancyModule
    {
        public EventsFeedModule(IEventStore eventStore) : base("/events")
        {
```

```

    Get("/", _ =>
    {
        long firstEventSequenceNumber, lastEventSequenceNumber;
        if (!long.TryParse(this.Request.Query.start.Value,
            out firstEventSequenceNumber))
            firstEventSequenceNumber = 0;
        if (!long.TryParse(this.Request.Query.end.Value,
            out lastEventSequenceNumber))
            lastEventSequenceNumber = long.MaxValue;

        return
            eventStore.GetEvents(
                firstEventSequenceNumber,
                lastEventSequenceNumber);
    });
}
}
}

```

Этот модуль использует только уже обсуждавшиеся нами возможности фреймворка Nancy. Однако вы могли обратить внимание на то, что он возвращает неносредственно результат выполнения метода `eventStore.GetEvents`, представляющий собой `IEnumerable<Event>`, Nancy сериализует его как массив. `Event` — структура, содержащая небольшое количество метаданных и поле `Content`, в котором находятся данные нашего события (листинг 4.12).

Листинг 4.12. Класс `Event`, служащий для представления событий

```

public struct Event
{
    public long SequenceNumber { get; }
    public DateTimeOffset OccuredAt { get; }
    public string Name { get; }
    public object Content { get; }

    public Event(
        long sequenceNumber,
        DateTimeOffset occurredAt,
        string name,
        object content)
    {
        this.SequenceNumber = sequenceNumber;
        this.OccuredAt = occurredAt;
        this.Name = name;
        this.Content = content;
    }
}

```

Свойство `Content` используется для данных конкретного события, именно в нем проявляется различие между событиями `NewSpecialOffer` и `UpdatedSpecialOffer`. У первого из них один тип объекта в свойстве `Content`, а у второго — другой.

Вот и все, что требуется для создания ленты событий. Подобная простота — огромное преимущество основанной на протоколе HTML ленты, применяемой для публикации событий. Взаимодействие на основе событий можно реализовать и посредством

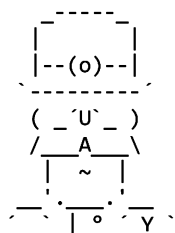
системы очереди, но это требует еще одной сложной технологии, которую необходимо научиться использовать и администрировать при эксплуатации в производственной среде. Эти сложности оправданы в некоторых случаях, но явно не в нашем.

Создание и запуск процесса — подписчика на события

Первый шаг при реализации процесса — подписчика на события — создание консольного приложения. Мы используем основанную на .NET Core платформу ASP.NET Core для веб-процессов в примерах микросервисов, так что создадим основанное на .NET Core консольное приложение и назовем его `LoyaltyProgramEventConsumer`. Создать консольное приложение на базе .NET Core в Visual Studio 2015 можно, выбрав тип проекта `Console Application (Package)` (Консольное приложение (Пакет)) в диалоговом окне `New Project` (Новый проект). Или можно воспользоваться командной строкой PowerShell, запустить генератор Yeoman для платформы ASP.NET¹ и выбрать опцию генерации `Console Application` (Консольное приложение) (листинг 4.13).

Листинг 4.13. Генерация консольного приложения с помощью генератора Yeoman для платформы ASP.NET

```
PS> yo aspnet
```



```
? What type of application do you want to create?
  Empty Web Application
> Console Application
  Web Application
  Web Application Basic [without Membership and Authorization]
  Web API Application
  Nancy ASP.NET Application
  Class Library
  Unit test project (xUnit.net)
```

Переместите курсор на эту строку и нажмите Enter для генерации консольного приложения

Создавали ли вы `LoyaltyProgramEventConsumer` с помощью Visual Studio или Yeoman, запустить его можно, перейдя в корневой каталог проекта — каталог, где находится файл `project.json` (в PowerShell), и воспользовавшись утилитой `dotnet`:

```
PS> dotnet run
```

Приложение пока еще пустое, так что ничего интересного в нем не происходит. Подобный запуск приложения `LoyaltyProgramEventConsumer` из PowerShell

¹ См. инструкции по установке Yeoman и генератора Yeoman для платформы ASP.NET в приложении А.

нужен только для проверки. При эксплуатации на производстве приложение `LoyaltyProgramEventConsumer` может запускаться как сервис операционной системы Windows. Если производственная среда основана на серверах Windows, которые администрируете вы или ваша фирма, подобный выбор вполне может оказаться оптимальным, но если ваша производственная среда находится в облаке, то не обязательно.

ПРЕДУПРЕЖДЕНИЕ

Мы реализуем приложение `LoyaltyProgramEventConsumer` в виде Windows-сервиса, который будет работать только в операционной системе Windows. Если вы хотели бы работать под управлением операционной системы Linux, можно создать аналогичное приложение `LoyaltyProgramEventConsumer` в виде Linux-демона.

Создание Windows-сервиса — задача несложная и в случае консольного приложения на основе платформы .NET Core ничем не отличающаяся от того, что было до появления .NET Core. В проекте уже имеется файл `Program.cs`, содержащий класс `Program`. В классе `Program` имеется метод `Main` — входная точка приложения. Чтобы превратиться в Windows-сервис, класс `Program` просто должен наследовать класс `ServiceBase` и переопределять методы `OnStart` и `OnStop`, как показано в листинге 4.14.

Листинг 4.14. Запуск `Program` в виде Windows-сервиса

```
using System.ServiceProcess;
```

```
public class Program : ServiceBase
{
```

```
    private EventSubscriber subscriber;
```

```
    public void Main(string[] args)
```

```
    {
        // В дальнейшем тут будет еще код
        Run(this);
    }
```

Запускаем как
Windows-сервис

Вызывается
при запуске
Windows-сервиса

```
    protected override void OnStart(string[] args)
```

```
    {
        // В дальнейшем тут будет еще код
    }
```

Вызывается, когда
Windows-сервис
останавливается

```
    protected override void OnStop()
```

```
    {
        // В дальнейшем тут будет еще код
    }
```

```
}
```

Если вы попытаетесь скомпилировать предыдущий код, то вам будут возвращены ошибки: тип `ServiceBase` не определен. Чтобы загрузить содержащую класс `ServiceBase` сборку, необходимо добавить строку в раздел `dependencies` файла `project.json` и отредактировать раздел `frameworks`, чтобы указать компилятору, что

это приложение использует полный фреймворк .NET. Раздел `frameworks` должен выглядеть следующим образом:

```
"dependencies": {
  "Newtonsoft.Json": "8.0.3",
  "System.ServiceProcess.ServiceController": "4.1.0",
  "System.Net.Http": "4.1.0"
},

"frameworks": {
  "net461": { }
},
```

После этого приложение снова должно компилироваться без ошибок. Для запуска необходимо установить его как Windows-сервис. С этой целью нам понадобится бинарная версия, так что необходимо явным образом скомпилировать проект. Это можно сделать с помощью утилиты командной строки `dotnet`:

```
PS> dotnet build
```

Эта команда компилирует проект и помещает результат в каталог `bin` внутри каталога проекта. Запустить результат можно путем вызова скомпилированного исполняемого файла:

```
PS> .\bin\Debug\net452\LoyaltyProgramEventConsumer
```

Теперь у нас имеется бинарная версия, и мы можем установить ее в качестве Windows-сервиса с помощью утилиты `sc.exe` операционной системы Windows. Необходимо указать утилите `sc.exe` имя Windows-сервиса и команду, которую Windows-сервис должен выполнять. В данном случае команда — это просто имя исполняемого файла `LoyaltyProgramEventConsumer`. В итоге получаем следующую команду:

```
PS> sc.exe create loyalty-program-event-consumer binPath="<path-to-project>\bin\Debug\net452\LoyaltyProgramEventConsumer"
```

После установки приложения `LoyaltyProgramEventConsumer` в качестве Windows-сервиса можно запускать и останавливать его точно так же, как и любой другой Windows-сервис.

Подписываемся на ленту событий

Теперь у нас имеется консольное приложение `LoyaltyProgramEventConsumer`, которое мы можем запускать как Windows-сервис. Его задача заключается в подписке на события микросервиса `Special Offers` и использовании микросервиса `Notifications` для уведомления зарегистрированных пользователей о специальных предложениях. На рис. 4.12 показаны взаимодействия микросервиса `Loyalty Program`, причем уже реализованные затенены.

Подписка на ленту событий, по существу, означает опрос конечной точки с событиями микросервиса, на который мы подписываемся. Время от времени будем отправлять HTTP-запрос типа `GET` к конечной точке `/events`, чтобы проверить, нет ли еще не обработанных нами событий.

Начнем реализацию сверху вниз. Прежде всего необходимо создать класс `EventSubscriber` и установить в нем таймер на 10 секунд (листинг 4.15).



Рис. 4.12. Основанное на событиях взаимодействие микросервиса Loyalty Program — подписка на ленту событий микросервиса Special Offers

Листинг 4.15. Запускаем таймер и задаем функцию обратного вызова

```

public class EventSubscriber
{
    private readonly string loyaltyProgramHost;
    private long start = 0;
    private int chunkSize = 100;
    private readonly Timer timer;

    public EventSubscriber(string loyaltyProgramHost)
    {
        this.loyaltyProgramHost = loyaltyProgramHost;
        this.timer = new Timer(10 * 1000);
        this.timer.AutoReset = false;
        this.timer.Elapsed += (_, __) => SubscriptionCycleCallback().Wait();
    }
}
  
```

Устанавливаем таймер на 10 секунд

Вызываем каждый раз, когда проходит заданное в таймере время

По прошествии 10 секунд мы проверяем, нет ли новых событий, если есть, то обрабатываем их, после чего снова ждем 10 секунд до очередной проверки наличия новых событий. Каждый раз при срабатывании таймера код из листинга 4.15 вызывает метод `SubscriptionCycleCallback`, который предпринимает попытку чтения новых событий из ленты и обрабатывает их, если обнаруживает. Обе эти задачи передаются на выполнение другим методам, которые мы рассмотрим совсем скоро. А пока вот код метода `SubscriptionCycleCallback` (листинг 4.16).

Листинг 4.16. Чтение и обработка событий

```

private async Task SubscriptionCycleCallback()
{
    var response = await ReadEvents().ConfigureAwait(false);
    if (response.StatusCode == HttpStatusCode.OK)
        HandleEvents(response.Content);
    this.timer.Start();
}
  
```

Ожидает выполнения HTTP-запроса типа GET к ленте событий

Метод `ReadEvents` выполняет HTTP-запрос типа GET к ленте событий (листинг 4.17). Он использует экземпляр класса `HttpClient`, который вы уже неоднократно видели.

Листинг 4.17. Чтение следующей группы событий

```
private async Task<HttpResponseMessage> ReadEvents()
{
    using (var httpClient = new HttpClient())
    {
        httpClient.BaseAddress =
            new Uri($"http://{this.loyaltyProgramHost}");
        var response = await httpClient.GetAsync(
            $"/events/?start={this.start}&end={this.start + this.chunkSize}")
            .ConfigureAwait(false);
        return response;
    }
}
```

Ждем получения новых событий

Используем параметры строки запроса для ограничения количества читаемых событий

Этот метод читает события из ленты и возвращает их методу `SubscriptionCycleCallback`. Если запрос оказался выполнен успешно, вызывается метод `HandleEvents`. События сначала десериализуются, после чего обрабатываются по очереди (листинг 4.18).

Листинг 4.18. Десериализация событий с последующей обработкой

```
private void HandleEvents(string content)
{
    var events = JsonConvert
        .DeserializeObject<IEnumerable<SpecialOfferEvent>>(content);
    foreach (var ev in events)
    {
        dynamic eventData = ev.Content;
        // Обрабатываем события 'ev' посредством eventData
        this.start = Math.Max(this.start, ev.SequenceNumber + 1);
    }
}
```

Работаем со свойством `Content` как с динамическим объектом ❶

Отслеживаем максимальный номер обработанного события ❷

В этом фрагменте кода стоит обратить внимание на несколько вещей.

Метод отслеживает, какие события уже были обработаны ❷. Это гарантирует, что из ленты не будут запрашиваться уже обработанные события.

Свойство `Content` событий обрабатывается как динамическое ❶. Как вы уже видели, не все события содержат одинаковые данные в свойстве `Content`, так что динамическая их обработка дает возможность обращаться к нужным свойствам `.Content`, не обращая внимания на остальные. Это разумный подход, поскольку желательно избегать строгостей при приеме входных данных — добавление нового поля в JSON-события микросервисом `Special Offers` не должно вызывать каких-либо проблем. Если *нужные* вам данные на месте, все остальное можно проигнорировать.

События десериализуются к типу `SpecialOfferEvent`. Он отличается от типа `Events`, который используется для сериализации событий в микросервисе `Special Offers`. Сделано это намеренно, поскольку представления данных двух различных

микросервисов не обязательно должны совпадать. До тех пор, пока микросервис Loyalty Program не требует отсутствующих здесь данных, никаких проблем нет.

Тип `SpecialOfferEvent` прост и содержит только используемые в микросервисе Loyalty Program поля:

```
public struct SpecialOfferEvent
{
    public long SequenceNumber { get; set; }
    public string Name { get; set; }
    public object Content { get; set; }
}
```

Чтобы связать код класса `EventSubscriber` с созданным в начале реализации процесса — подписчика на события (см. листинг 4.14) Windows-сервисом, мы добавим в класс `EventSubscriber` еще два метода: один будет запускать таймер, а второй — останавливать его. По существу, эти два метода запускают и останавливают процесс подписки на события:

```
public void Start()
{
    this.timer.Start();
}

public void Stop()
{
    this.timer.Stop();
}
```

Теперь наш Windows-сервис может создавать объект `EventSubscriber` при загрузке и вызывать методы `Start` и `Stop` при запуске и остановке сервиса. После заполнения недостающих частей листинга 4.14 Windows-сервис приобретает следующий вид (листинг 4.19).

Листинг 4.19. Windows-сервис, запускающий и прекращающий подписку

```
public class Program : ServiceBase
{
    private EventSubscriber subscriber;

    public void Main(string[] args)
    {
        this.subscriber = new EventSubscriber("localhost:5000");
        Run(this);
    }

    protected override void OnStart(string[] args)
    {
        this.subscriber.Start();
    }

    protected override void OnStop()
    {
        this.subscriber.Stop();
    }
}
```

Этот фрагмент кода завершает реализацию подписки на события. Как вы уже видели, подписка на ленту событий означает нериодический опрос ее на предмет появления новых событий с последующей их обработкой.

4.3. Резюме

- ❑ Существует три типа взаимодействия микросервисов:
 - на основе команд, при котором один микросервис использует HTTP-запросы типа POST или PUT, чтобы заставить другой микросервис вынолнить какое-то действие;
 - на основе запросов, при котором один микросервис вынолняет HTTP-запрос типа GET, чтобы получить данные о состоянии другого микросервиса;
 - на основе событий, при котором один микросервис предоставляет ленту событий, на которую другие микросервисы могут подписываться путем опроса этой ленты на предмет новых событий.
- ❑ Связанность при взаимодействии на основе событий более слабая, чем при взаимодействии на основе команд и запросов.
- ❑ Существует возможность подключиться к привязке модели и согласованию поддерживаемого фреймворка Nancy для обеспечения поддержки форматов данных, отличных от JSON и XML.
- ❑ Загрузчик фреймворка Nancy используется для настройки самого Nancy и приложений Nancy.
- ❑ Для отправки команд другим микросервисам и выполнения запросов к ним можно использовать класс `HttpClient`.
- ❑ Фреймворк Nancy можно применять для предоставления конечных точек для получения и обработки команд и запросов.
- ❑ Фреймворк Nancy может обеспечивать работу простой ленты событий.
- ❑ Создать процесс для подписки на события можно выполнением следующих действий:
 - созданием консольного приложения платформы .NET Core;
 - реализацией и установкой консольного приложения в качестве Windows-сервиса;
 - применением таймера для организации опросов ленты событий консольным приложением;
 - использованием класса `HttpClient` для чтения событий из ленты.

5

Хранение данных и их принадлежность

В этой главе:

- ❑ микросервисы и их хранилища данных;
- ❑ как бизнес-возможности обуславливают принадлежность данных;
- ❑ дублируем данные для ускорения работы системы и новышения ее надежности;
- ❑ создаем модели чтения из лент событий с помощью подписчиков на события;
- ❑ как микросервисы хранят данные.

Программные системы создают, используют и преобразуют данные. Без данных большинство программных систем ничего бы не стоили, и для систем микросервисов это тоже справедливо. Из этой главы вы узнаете, где должны храниться данные и какие микросервисы должны отвечать за поддержание их актуальности. Более того, научитесь использовать дублирование данных, чтобы ускорить работу системы и повысить ее надежность.

5.1. У любого микросервиса имеется хранилище данных

Один из приведенных в главе 1 отличительных признаков микросервисов гласит, что у каждого микросервиса должно быть свое хранилище данных. Микросервис полностью контролирует данные в своем хранилище — именно те, которые ему нужны. По большей части это данные, относящиеся к реализуемой им бизнес-возможности, а также вспомогательные, например заэкшированные данные и созданные на основе лент событий модели чтения.

Наличие у каждого микросервиса хранилища данных означает, что не обязательно использовать одну технологию баз данных для всех микросервисов. Можно подобрать для каждого микросервиса ту технологию, которая лучше всего подходит для хранимых им данных.

Микросервисам обычно требуется хранить три типа данных:

- ❑ данные, относящиеся к реализуемой конкретным микросервисом бизнес-возможности. Это данные, за которые этот микросервис отвечает, которые он должен хранить в целости и сохранности и актуальность которых поддерживать;
- ❑ порождаемые микросервисом события. Во время обработки команд микросервису может понадобиться породить какие-либо события, чтобы оновестить остальную систему об обновлениях данных, за которые он отвечает;

- ❑ модели чтения, основанные на запросах к другим микросервисам или порождаемых ими событиях.

Эти три типа данных можно хранить в различных базах данных или даже в базах данных разного типа.

5.2. Распределение данных по микросервисам

При определении того, где хранить данные в системе микросервисов, вступают в действие конкурирующие силы, основные из которых — принадлежность данных и их локальность.

- ❑ *Владеть данными* означает нести ответственность за поддержание их актуальности.
- ❑ *Локальность данных* относится к тому, где должны храниться требующиеся микросервису данные. Обычно их следует хранить неподалеку — желательно в самом микросервисе.

Эти две силы могут конфликтовать, и, чтобы удовлетворить обе, зачастую приходится хранить данные в нескольких местах. Это нормально, но важно, чтобы только одно из этих мест считалось эталонным источником информации. На рис. 5.1 показано, что при хранении эталонной копии информации в одном из микросервисов другие микросервисы могут создавать зеркальные копии этой информации в собственных хранилищах данных.

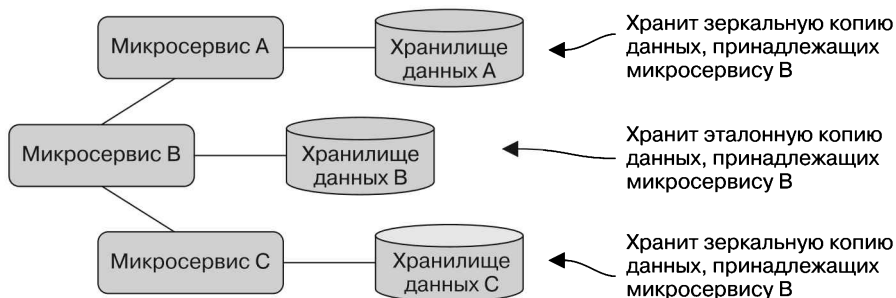


Рис. 5.1. Микросервисы А и С взаимодействуют с микросервисом В. В микросервисах А и С могут храниться зеркальные копии данных, принадлежащих микросервису В, но эталонная копия находится в хранилище данных микросервиса В

Правило 1: бизнес-возможности обуславливают принадлежность данных

Первое правило при выяснении того, кому принадлежит элемент данных в системе микросервисов: принадлежность данных определяется бизнес-возможностями. Как говорилось в главе 3, основной фактор определения области ответственности микросервиса — то, что он должен реализовывать бизнес-возможность. Бизнес-возможность определяет границы микросервиса — в микросервисе должно быть реализовано все, что к ней относится. Сюда входит и хранение относящихся к бизнес-возможности данных.

Предметно-ориентированное проектирование учит нас, что некоторые понятия могут встречаться в нескольких бизнес-возможностях, причем их значение может слегка различаться. Например, в нескольких микросервисах может существовать понятие «нокупатель», и они будут обрабатывать и хранить сущности нокупателей. Данные, хранимые в различных микросервисах, могут пересекаться, но важно четко понимать, какой микросервис за какие данные отвечает.

Например, домашний адрес нокупателя должен принадлежать только одному микросервису. Другому микросервису может принадлежать история покупок нокупателя, третьему — настройки уведомлений нокупателя. Чтобы определиться с тем, какой микросервис отвечает за конкретный элемент данных, например домашний адрес покупателя, необходимо выяснить, какой бизнес-процесс отвечает за поддержание актуальности этого элемента. Отвечающий за эту бизнес-возможность микросервис должен отвечать и за хранение/актуальность этих данных.

Вновь рассмотрим сайт интернет-магазина из глав 1 и 2. На рис. 5.2 приведен общий обзор того, как эта система обрабатывает пользовательские запросы по добавлению товаров в корзину заказов. Большая часть микросервисов на рис. 5.2 затенена серым цветом, чтобы привлечь ваше внимание к трем микросервисам: Shopping Cart, Product Catalog и Recommendations.

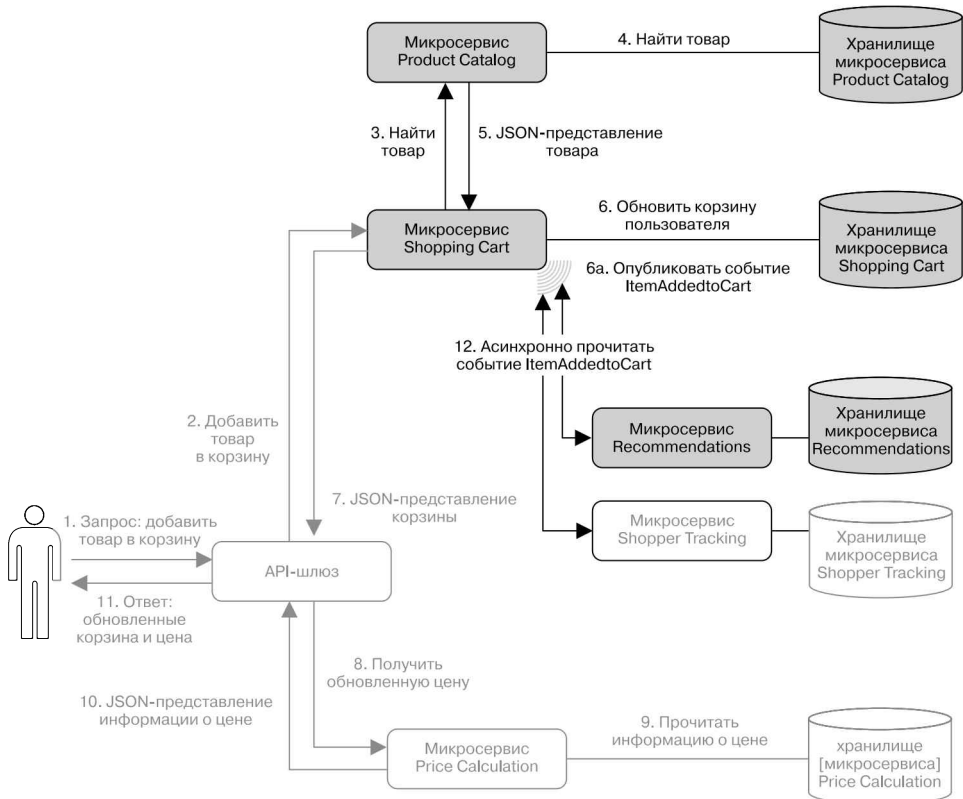


Рис. 5.2. В этом примере интернет-магазина сосредоточимся на разделении данных между микросервисами Shopping Cart, Product Catalog и Recommendations

Каждый из выделенных микросервисов отвечает за обработку бизнес-возможности: Shopping Cart — за отслеживание корзины заказов пользователей, Product Catalog — предоставление остальной системе доступа к информации из каталога товаров, а Recommendations — за оценку и выдачу пользователям интернет-магазина рекомендаций по товарам. С каждой из этих бизнес-возможностей связаны определенные данные, и каждый из неречисленных микросервисов владеет связанными с его бизнес-возможностью данными и несет за них ответственность. На рис. 5.3 показаны принадлежащие каждому из этих микросервисов данные. Фраза «данные принадлежат микросервису» (или «микросервис владеет данными») означает, что микросервис должен обеспечивать хранение этих данных и служить их эталонным источником.

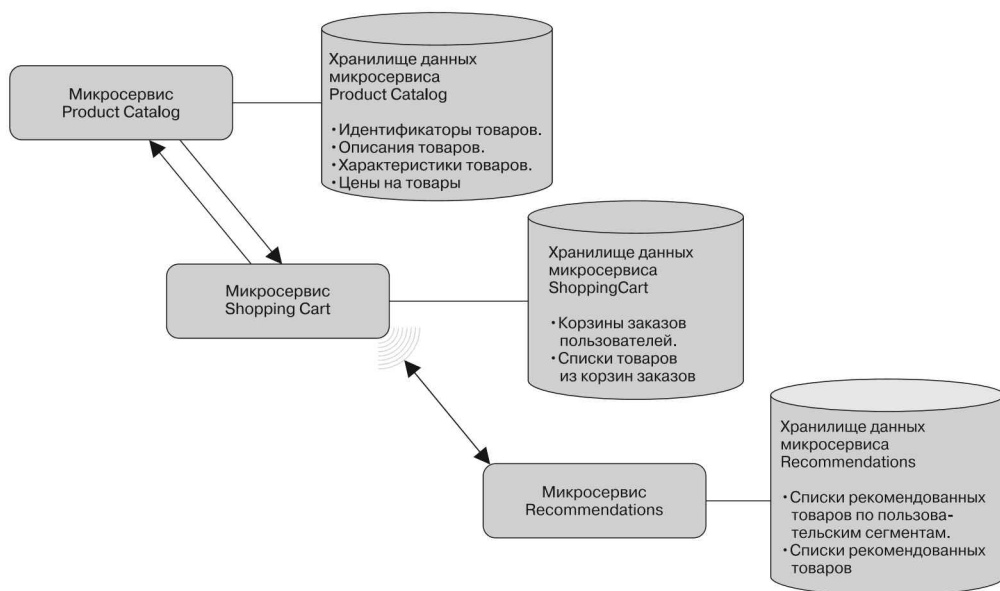


Рис. 5.3. Каждому из микросервисов принадлежат данные, относящиеся к реализуемой им бизнес-возможности

Правило 2: репликация данных для ускорения работы системы и повышения ее надежности

Второй из факторов, играющих роль в определении места хранения данных в системе микросервисов, — локальность. Есть значительная разница между запросом микросервиса к его собственной базе данных и запросом к другому микросервису. Запрос к собственной базе данных в целом и быстрее, и устойчивее к ошибкам, чем запрос к другому микросервису.

Онределившись с принадлежностью данных, вы, вероятно, обнаружите, что микросервисам требуется запрашивать данные друг у друга. Подобный вид взаимодействия создает определенную связанность: запрос одного микросервиса к другому

означает, что первый из них связан со вторым. Если второй микросервис в данный момент не работает или работает медленно, это повлияет на работу первого.

Для ослабления этой связанности можно кэшировать ответы на запросы. Иногда мы будем кэшировать ответы в первоначальном виде, а иногда — сохранять модель чтения, основанную на ответах на запросы. В обоих случаях необходимо заранее решить, когда и как кэшированный элемент данных перестает быть действительным. Именно микросервис — владелец данных должен принимать решение о том, до каких пор элемент данных остается действительным, а когда он становится недействительным. Таким образом, конечные точки, отвечающие на запросы относительно принадлежащих микросервису данных, должны включать в ответ заголовки кэширования, указывающие вызывающей стороне, сколько времени она может кэшировать данные ответа.

Использование HTTP-заголовков для управления кэшированием

Протокол HTTP определяет несколько заголовков, которые можно применять для управления кэшированием HTTP-ответов. Механизм кэширования HTTP во многих случаях избавляет от необходимости:

- ❑ запрашивать уже имеющуюся у запрашивающей стороны информацию;
- ❑ отправлять полные HTTP-ответы.

Для устранения необходимости запрашивать уже имеющуюся у запрашивающей стороны информацию сервер может добавлять к ответам заголовок `cache-control`. В спецификации протокола HTTP определен ряд опций, которые можно указывать в заголовке `cache-control`. Наиболее распространенные из них — директивы `private|public` и `max-age`. Первая определяет, кто может кэшировать ответы: только запрашивающая сторона или также промежуточные узлы — прокси-серверы, например. Директива `max-age` задает промежуток времени (в секундах), на который может быть кэширован ответ. Например, следующий заголовок `cache-control` означает, что запрашивающая сторона и только она может кэшировать ответ на 3600 секунд:

```
cache-control: private, max-age:3600
```

То есть запрашивающая сторона может в любой момент при необходимости выполнить HTTP-запрос на тот же URL с тем же методом — `GET`, `POST`, `PUT`, `DELETE` — и тем же телом запроса и повторно воспользоваться полученным ответом в течение 3600 секунд. Стоит отметить, что строка запроса является частью URL, так что она при кэшировании учитывается.

Для исключения необходимости отправки полного ответа в случаях, когда в кэше у запрашивающей стороны содержится ответ, возможно потерявший актуальность, сервер может добавлять в ответы заголовок `etag`. Это идентификатор ответа. Запрашивающая сторона может включать `etag` в заголовок запроса при выполнении в дальнейшем запроса на тот же URL с тем же методом и тем же телом. Сервер, прочитав `etag`, будет знать, какой ответ уже кэширован у запрашивающей стороны. Если сервер считает, что ответ все еще актуален, он может вернуть его с кодом со-

стояния 304 Not Modified (Не изменялось), сообщая клиенту, что можно использовать уже кэшированный ответ. Более того, сервер может добавить в ответ 304 заголовок `cache-control`, чтобы продлить период времени кэширования ответа. Обратите внимание на то, что заголовок `etag` устанавливается сервером и позднее читается тем же сервером.

Вернемся к микросервисам с рис. 5.3. Микросервис Shopping Cart использует информацию о товаре, которую он получает, выполняя запрос к микросервису Product Catalog. Срок актуальности информации из каталога товаров для каждого конкретного товара должен определять микросервис Product Catalog — владелец этих данных. Следовательно, микросервис Product Catalog должен добавлять относящиеся к кэшированию заголовки в свои ответы, а Shopping Cart — использовать их для определения того, на протяжении какого срока он может кэшировать ответ. На рис. 5.4 приведена последовательность запросов к микросервису Product Catalog, которые должен выполнять микросервис Shopping Cart.

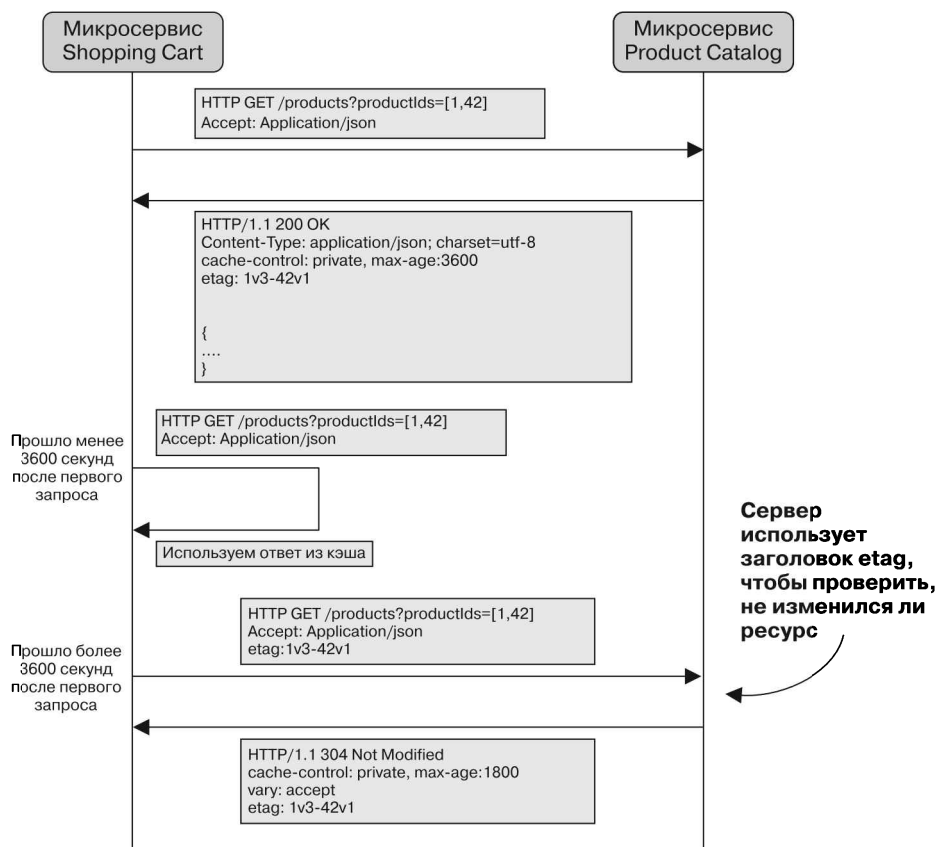


Рис. 5.4. Микросервис Product Catalog с помощью заголовков кэширования в своих HTTP-ответах может предоставлять взаимодействующим с ним микросервисам возможность кэширования ответов. Он устанавливает заголовок `max-age`, указывающий длительность кэширования ответов, а также состоящий из идентификатора и версии товара заголовок `etag`

На рис. 5.4 заголовки кэширования в ответе на первый запрос указывают микросервису Shopping Cart, что он может кэшировать ответ на 3600 секунд. При второй попытке выполнения того же запроса микросервисом Shopping Cart используется тот же ответ, поскольку прошло менее 3600 секунд. В третий раз выполняется запрос к микросервису Product Catalog, поскольку прошло более 3600 секунд. Этот запрос включает заголовок `etag` из первого ответа. Микросервис Product Catalog на основе `etag` решает, что ответ остался неизменным, так что он отправляет более короткий ответ `304 Not Modified` (Не изменялось) вместо полного ответа. Ответ `304` включает новый набор заголовков кэширования, разрешающий микросервису Shopping Cart кэшировать уже находящийся в кэше ответ еще на 1800 секунд.

В дальнейшем мы обсудим включение в ответы заголовков кэширования в обработчиках маршрутов фреймворка Napsy. Мы также рассмотрим их чтение из ответа на стороне клиентского приложения.

Использование моделей чтения для зеркального копирования данных других микросервисов

Для микросервиса запрашивать принадлежащие ему данные из своей базы данных — вполне обычное дело, но не столь естественным представляется запрос из своей базы чужих данных. Естественный способ получения принадлежащих другому микросервису данных — запрос к этому другому микросервису. Но зачастую оказывается возможно заменить запрос к другому микросервису запросом к собственной базе данных путем создания *модели чтения* (*read model*) — модели данных, запросы к которой можно выполнять легко и эффективно. Они отличаются от моделей, используемых для хранения принадлежащих микросервису данных, задача которых состоит в хранении эталонной копии данных и обеспечении при необходимости возможности удобного ее обновления.

Конечно, в модели чтения записываются данные — иначе они были бы пустыми, — но эти данные записываются в виде последовательности выполняемых в других местах изменений. При этом за счет некоторого усложнения записи достигается большая простота чтения.

Модели чтения часто основываются на порождаемых другими микросервисами событиях. Один микросервис подписывается на события другого и обновляет свою модель данных событий по мере их поступления.

Модели чтения можно также создавать на основе ответов на запросы к другим микросервисам. В этом случае время жизни данных в модели чтения определяется заголовками кэширования в этих ответах, как и при обычном кэшировании ответов. Разница между прямым кэшированием и моделью чтения состоит в том, что для построения модели чтения данные из ответов преобразуются и, возможно, дополняются с целью упрощения и ускорения дальнейшего чтения. Это значит, что форма данных определяется сценариями чтения вместо сценариев записи.

Рассмотрим пример. Микросервис Shopping Cart публикует события при каждом добавлении товара в корзину заказов или удалении товара из нее. Рисунок 5.5 демонстрирует микросервис Shopper Tracking, подписывающийся на эти события и обновляющий на их основе модель чтения. Микросервис Shopper Tracking предо-

ставляет бизнес-пользователям возможность выяснения того, сколько раз конкретные товары добавлялись в корзины заказов или удалялись из них.

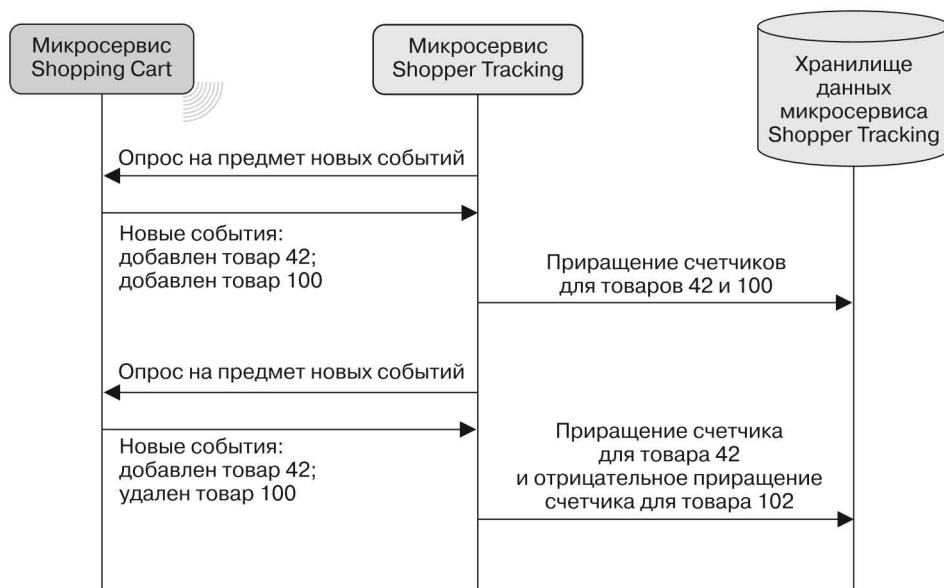


Рис. 5.5. Микросервис Shopper Tracking подписывается на события микросервиса Shopping Cart и отслеживает, сколько раз товары добавлялись в корзины заказов или удалялись из них

Публикуемые микросервисом Shopping Cart события сами по себе не являются эффективной моделью, которую можно опрашивать для выяснения частоты добавления товаров в корзины заказов или их удаления из них. Но это хороший источник информации для создания такой модели. У микросервиса Shopper Tracking имеется два счетчика для каждого товара: один для количества добавлений товара в корзину заказов, а второй — для количества удалений. При каждом поступлении события от микросервиса Shopping Cart обновляется один из этих счетчиков, и оба этих счетчика считываются при каждом запросе о товаре.

Где микросервисы хранят свои данные

Микросервис может использовать одну, две базы данных или более. Для части хранимых микросервисов данных может хорошо подходить один тип базы данных, а для другой части — другой. Сегодня доступно множество вполне жизнеспособных технологий баз данных, и я не буду заниматься их сравнением. Существуют, однако, более широкие категории баз данных, которые не мешает учитывать при выборе, включая реляционные базы данных (relational database), хранилища типа «ключ — значение» (key/value store), документоориентированные базы данных (document database), хранилища на базе столбцов (column store) и графовые базы данных (graph database).

На выбор технологии базы данных (или баз данных) для микросервиса могут влиять многие факторы, в том числе следующие.

- ❑ Какова конфигурация данных? Хорошо ли для них подходит реляционная модель, документоориентированная модель, модель «ключ — значение» или графовая модель?
- ❑ Каковы сценарии записи? Какой объем данных записывается? Выполняются ли записи пакетами, или они равномерно распределены по времени?
- ❑ Каковы сценарии чтения? Какой объем данных читается за один раз? А в целом? Производится ли чтение пакетами?
- ❑ Каков объем записываемых данных по сравнению с читаемыми?
- ❑ Для каких баз данных команда уже имеет опыт разработки и эксплуатации в производственной среде?

Эти вопросы — и поиск ответов на них — помогут вам не только определиться с подходящей базой данных, но и углубить понимание нефункциональных качеств, которыми должен обладать микросервис. Благодаря им вы поймете, насколько надежным должен быть микросервис, какую нагрузку должен выдерживать, как эта нагрузка выглядит, какая латентность допустима и т. д.

Эти углубленные познания важны, но отметьте, что я не рекомендую выполнять полномасштабный анализ всех «за» и «против» использования различных баз данных при каждом создании нового микросервиса. Необходимо, чтобы микросервис можно было быстро создать и развернуть для эксплуатации в производственной среде. Цель должна состоять не в том, чтобы найти идеальную базу данных для конкретной задачи — нужно просто найти подходящую с точки зрения ответов на приведенные ранее вопросы. Вы можете столкнуться с ситуацией, когда документоориентированная база данных представляется хорошим вариантом, причем вы уверены, что как Couchbase, так и MongoDB вполне подойдут. В таком случае просто выберите любую. Лучше быстро ввести микросервис в эксплуатацию на производстве с одной из них, а когда-нибудь потом, возможно, заменить его реализацией, использующей другую, чем задерживать развертывание первой версии микросервиса из-за подробного сравнения Couchbase и MongoDB.

Сколько баз данных может быть в системе

Решение о том, какую базу данных использовать, зависит не только от того, какая база данных подходит для конкретного микросервиса. Желательно принимать во внимание более широкие соображения. В системе микросервисов присутствует множество микросервисов и множество хранилищ данных. Стоит задуматься, сколько технологий баз данных должно быть в системе. Должен соблюдаться компромисс: или остановиться на нескольких технологиях баз данных, или получить в системе свалку технологий.

Преимущества нескольких технологий таковы.

- Надежная работа баз данных при эксплуатации в производственной среде, особенно в долгосрочной перспективе.
- Разработчики могут включиться в работу и эффективно трудиться над ранее неизвестным им микросервисом.

Преимущества свалки следующие:

- Возможность выбора оптимальной технологии базы данных для каждого микросервиса в смысле удобства сопровождения, производительности, безопасности, надежности и т. д.
- Обеспечение заменяемости микросервисов. Если микросервисом начинают заниматься новые разработчики, не согласные с выбором базы данных, у них должна быть возможность заменить базу данных или даже микросервис в целом.

Степень важности этих целей различается от фирмы к фирме, главное — осознавать необходимость компромиссов.

5.3. Реализация хранилища данных микросервиса

Мы обсудили, что должно происходить с данными в системе микросервисов, включая то, какими данными микросервис должен владеть, а какие — зеркально конировать. Пришло время переключиться на код, необходимый для хранения данных.

Сосредоточимся на хранении микросервисом принадлежащих ему данных, включая хранение порождаемых им событий. Сначала я покажу, как сделать это с помощью SQL Server и облегченной библиотеки доступа к данным Dapper. А затем — как хранить события в специально предназначенной для этой цели базе данных, носящей идеально подходящее название Event Store («Хранилище событий»).

Используемые в этой главе новые технологии

В этой главе мы начнем использовать несколько пока еще не встречавшихся в данной книге технологий.

- *SQL Server* — SQL база данных компании «Майкрософт». Информацию об установке SQL Server можно найти в приложении А.
- *Dapper* (<https://github.com/StackExchange/dapper-dot-net>) — облегченное средство объектно-реляционного отображения (object-relational mapper (ORM)). Я познакомлю вас с Dapper чуть позже.
- *Event Store* (<https://geteventstore.com/>) — база данных, специально предназначенная для хранения событий. Я познакомлю вас с Event Store буквально через несколько строк.

Dapper: облегченное O/RM

Dapper — простая библиотека для работы с данными, находящимися в SQL-базах данных, из приложений на языке C#. Она представляет собой часть семейства библиотек, которые иногда называют *микро-ORM*, включающего также Simple.Data и Massive. Эти библиотеки используют SQL, и основные их преимущества — удобство использования и быстрота.

В то время как более традиционные ORM формируют весь код SQL, необходимый для чтения данных из базы данных и записи их туда, Dapper предполагает, что вы сами будете писать свой код SQL. Мне кажется, что это развязывает руки разработчикам при работе с базой данных с простой схемой.

Полностью в духе использования облегченных технологий для микросервисов я предпочел Dapper полноценным ORM, таким как Entity Framework или NHibernate. Часто база данных микросервиса проста, так что проще всего добавить к ней тонкий слой, например Dapper, чтобы упростить решение в целом. Можно было выбрать любой другой микро-ORM, но мне нравится именно Dapper. В этой главе будем использовать Dapper для взаимодействия с SQL Server, но Dapper работает и с другими SQL базами данных, такими как PostgreSQL и MySQL.

Event Store: специализированная база данных

Event Store — сервер баз данных с открытым исходным кодом, предназначенный специально для хранения событий. Event Store хранит события в виде JSON-документов, но отличается от документоориентированных баз данных тем, что рассматривает JSON-документы как часть потока событий. Хотя Event Store — нишевой продукт в силу своей специализации на хранении событий, он широко используется и доказал свою работоспособность при эксплуатации в производственной среде со значительной нагрузкой.

Помимо хранения событий, Event Store может читать их и обеспечивать подписку на них. Например, Event Store предоставляет собственные ленты событий (в виде лент ATOM), на которые могут подписываться клиентские приложения. Если вы готовы положиться на Event Store, использование его ленты событий ATOM для обнаружения событий для других микросервисов может оказаться вполне реальной альтернативой тому способу, с помощью которого мы будем реализовывать ленты событий в этой книге.

Event Store работает, предоставляя HTTP API для хранения, чтения и подписки на события. Существует множество клиентских библиотек для Event Store на различных языках программирования, включая C#, F#, Java, Scala, Erlang, Haskell и JavaScript, что значительно облегчает работу с этой базой данных.

Хранение принадлежащих микросервису данных

После того как решен вопрос о том, какие именно данные принадлежат микросервису, задача их хранения становится вопросом техники. Нюансы реализации зависят от выбранной базы данных. Единственная связанная с микросервисами особенность — то, что данные принадлежат исключительно самому микросервису и к ним может обращаться только он.

В качестве примера вернемся к микросервису Shopping Cart. Он владеет данными о корзинах заказов пользователей и, следовательно, хранит их. Будем хранить корзины заказов в SQL Server с помощью библиотеки Dapper.

Мы реализовали большую часть микросервиса Shopping Cart в главе 2. В этом разделе дополним реализацию частями, относящимися к хранению данных.

ПРИМЕЧАНИЕ

Для работы с кодом этого примера вам понадобится SQL Server. Необходимую для установки SQL Server информацию можно найти в приложении А. Или же можно использовать PostgreSQL, но это потребует внесения в код различных мелких изменений.

Если вы уже сталкивались с хранением данных в SQL Server, то реализация должна быть вам привычна, в этом-то и смысл. Ничего хитрого в хранении принад-

лежащих микросервису данных быть не должно. Вот этапы реализации хранения корзины заказов.

1. Создать базу данных.
2. Воспользоваться библиотекой Dapper для реализации кода чтения, записи и обновления корзины заказов.

Создадим простую базу данных для хранения корзины заказов. Она будет состоять из двух таблиц (рис. 5.6).

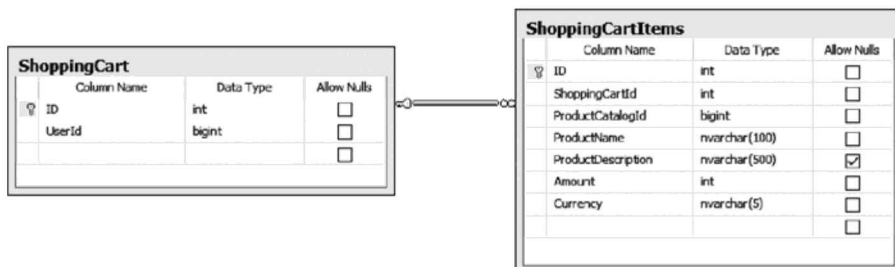


Рис. 5.6. База данных ShoppingCart состоит всего из двух таблиц: в одной содержится по строке для каждой корзины заказов, а в другой — по строке для каждого товара в корзине

В загружаемом коде можно найти SQL-сценарий для создания базы данных ShoppingCart. Его файл называется `create-shopping-cart-db.sql` и находится в каталоге `\code\Chapter05\ShoppingCart\src\database-scripts\`. Для создания базы данных ShoppingCart достаточно выполнить этот сценарий в SQL Server Management Studio.

После создания базы данных необходимо реализовать в микросервисе код для чтения, записи и обновления базы данных. Установите в микросервис пакет Dapper системы управления пакетами NuGet. Напомним: сделать это можно, добавив библиотеку Dapper в файл `project.json` и выполнив команду `dotnet restore` из PowerShell. Раздел `dependencies` файла `project.json` должен выглядеть следующим образом:

```
"dependencies": {
  "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
  "Microsoft.AspNetCore.Owin": "1.0.0",
  "Nancy": "2.0.0-barneyrubble",
  "Polly": "4.2.1",
  "Dapper": "1.50.0-rc2a"
},
```

Добавляем библиотеку Dapper

В главе 2 для микросервиса Shopping Cart требовалась реализация интерфейса `IShoppingCart`. Мы немного изменим этот интерфейс, чтобы его реализация могла выполнять асинхронные обращения к базе данных. Вот модифицированный интерфейс:

```
public interface IShoppingCartStore
{
  Task<ShoppingCart> Get(int userId);
  Task Save(ShoppingCart shoppingCart);
}
```

Теперь пришло время посмотреть на реализацию этого интерфейса. Во-первых, рассмотрим код для чтения корзины заказов из базы данных (листинг 5.1).

Листинг 5.1. Чтение корзины заказов с помощью библиотеки Dapper

```
namespace ShoppingCart.ShoppingCart
{
    using System.Threading.Tasks;
    using System.Data.SqlClient;
    using Dapper;

    public class ShoppingCartStore : IShoppingCartStore
    {
        private string connectionString =
@"Data Source=.\SQLEXPRESS;Initial Catalog=ShoppingCart;
Integrated Security=True";

        private const string readItemsSql =
@"select * from ShoppingCart, ShoppingCartItems
where ShoppingCartItems.ShoppingCartId = ID
and ShoppingCart.UserId=@UserId";

        public async Task<ShoppingCart> Get(int userId)
        {
            using (var conn = new SqlConnection(connectionString))
            {
                var items = await
                    conn.QueryAsync<ShoppingCartItem>(
                        readItemsSql,
                        new { UserId = userId });
                return new ShoppingCart(userId, items);
            }
        }
    }
}
```

Строка подключения к базе данных ShoppingCart

Библиотека Dapper имеет дело с SQL, так что у нас будет код SQL в коде на языке C#

Открывает соединение с базой данных ShoppingCart

Используем метод расширения библиотеки Dapper для выполнения SQL-запроса и преобразования результатов обратно в IEnumerable объектов типа ShoppingCartItem

Библиотека Dapper — простой инструмент с некоторыми удобными методами расширения интерфейса `IDbConnection`, упрощающими работу с SQL на языке программирования C#. Она также предоставляет некоторые простейшие возможности отображения. Например, в листинге 5.1 Dapper отображает возвращаемые SQL-запросом строки в `IEnumerable<ShoppingCartItem>`, поскольку названия столбцов в базе данных такие же, как и названия свойств в `ShoppingCartItem`.

Библиотека Dapper не скрывает факта работы с SQL, так что вы можете видеть строки SQL-запросов в коде. Это может показаться атавизмом нервных дней существования .NET, но я обнаружил, что при работе с простой схемой базы данных, как это обычно и происходит в микросервисах, строки кода SQL-запросов в коде на языке C# не являются проблемой.

Запись корзины заказов в базу данных тоже выполняется через библиотеку Dapper. Следующий метод из класса `ShoppingCartStore` реализует эту возможность (листинг 5.2).

Листинг 5.2. Запись корзин заказов в базу данных

```
private const string deleteAllForShoppingCartSql=
@"delete item from ShoppingCartItems item
inner join ShoppingCart cart on item.ShoppingCartId = cart.ID
and cart.UserId=@UserId";

private const string addAllForShoppingCartSql=
@"insert into ShoppingCartItems
(ShoppingCartId, ProductCatalogId, ProductName,
ProductDescription, Amount, Currency)
values
(@ShoppingCartId, @ProductCatalogId, @ProductName,v
@ProductDescription, @Amount, @Currency)";

public async Task Save(ShoppingCart shoppingCart)
{
    using (var conn = new SqlConnection(connectionString))
    using (var tx = conn.BeginTransaction())
    {
        await conn.ExecuteNonQuery( ←
            deleteAllForShoppingCartSql,
            new { UserId = shoppingCart.UserId },
            tx).ConfigureAwait(false);
        await conn.ExecuteNonQuery( ←
            addAllForShoppingCartSql,
            shoppingCart.Items,
            tx).ConfigureAwait(false);
    }
}
```

Удаляет все уже существующие элементы корзины заказов

Добавляет текущие элементы корзины заказов

На этом реализацию хранения информации о корзине заказов в микросервисе Shopping Cart можно считать завершенной. Она напоминает хранение данных в более традиционной архитектуре, например монолите или традиционном SOA-сервисе, если не считать того, что узкая область действия микросервиса зачастую означает настолько простую модель, что требуются минимальные усилия по отображению между кодом на языке C# и схемой базы данных.

Хранение порождаемых микросервисом событий

Этот раздел посвящен хранению порождаемых микросервисом событий. При обработке команды микросервис может решить, что необходимо сгенерировать событие. На рис. 5.7 показан стандартный набор компонентов микросервиса: модель предметной области порождает события. Обычно это происходит при изменении (нескольких изменениях) состояния данных, за которые отвечает этот микросервис.

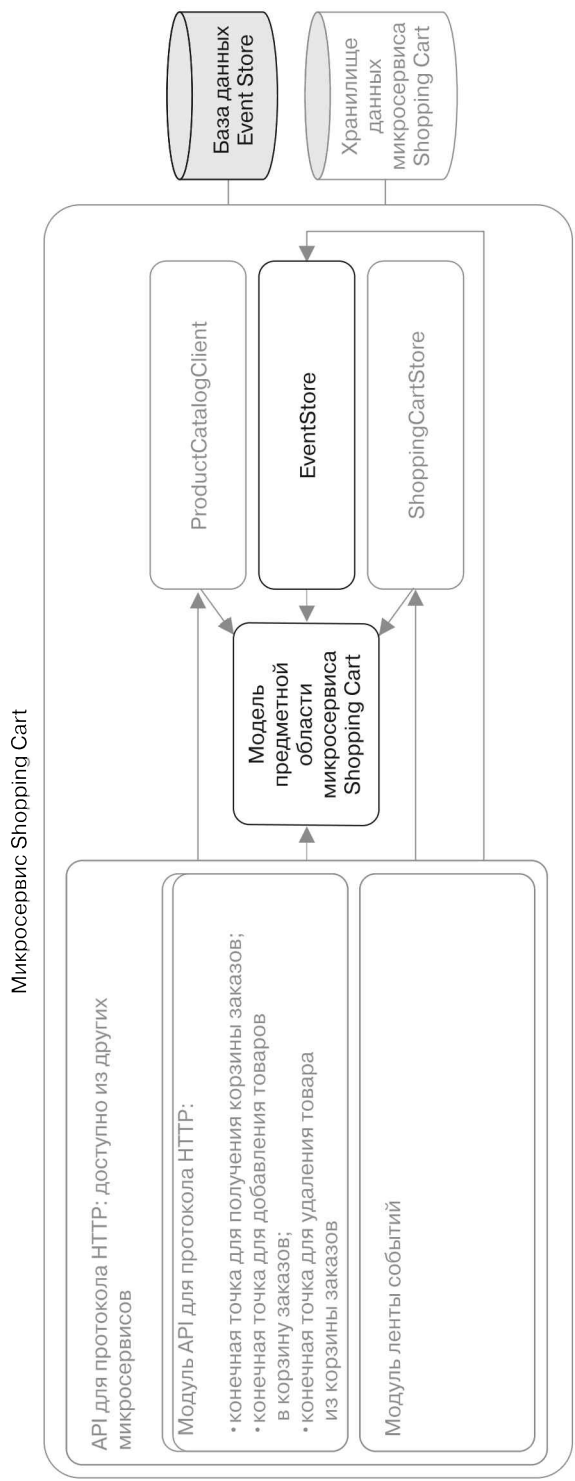


Рис. 5.7. Участвующие в порождении и сохранении событий компоненты микросервиса Shopping Cart: модель предметной области микросервиса Shopping Cart, компонент EventStore и база данных Event Store

События должны отражать изменения состояния принадлежащих микросервису данных. Они также должны иметь смысл для реализуемой микросервисом возможности. Например, в микросервисе Shopping Cart при добавлении пользователем товара в корзину заказов порождается событие `ItemAddedToShoppingCart`¹, а не `RowAddedToShoppingCartTable`². Различие в следующем: первое отражает важное для системы событие — пользователь сделал что-то интересное в коммерческом смысле, второе сообщает о какой-то технической детали — элемент программного обеспечения выполнил какое-либо действие, потому что программист решил реализовать это именно так, а не иначе. События должны быть значимыми на данном уровне абстракции реализуемой микросервисом возможности, и они зачастую будут охватывать несколько изменений в расположенной ниже базе данных. События должны соответствовать транзакциям в коммерческом смысле, а не транзакциям в смысле базы данных.

Событие, сгенерированное предметной логикой микросервиса, сохраняется в хранилище событий микросервиса. На рис. 5.8 это происходит с помощью компонента `EventStore`, отвечающего за взаимодействие с базой данных, в которой хранятся события.



Рис. 5.8. При порождении моделью предметной области события код компонента `EventStore` должен записать его в базу данных `Event Store`

В следующих двух разделах показаны две реализации компонента `EventStore`. Первая сохраняет события вручную в таблице в SQL-базе данных, а вторая использует базу данных `Event Store` с открытым исходным кодом.

Сохранение событий вручную

В этом разделе мы создадим реализацию компонента `EventStore` микросервиса Shopping Cart, который сохраняет события в таблице в СУБД SQL Server. Компонент `EventStore` отвечает как за запись событий в базу данных, так и за чтение их из нее.

ПРИМЕЧАНИЕ

Ручная реализация демонстрирует все нюансы сохранения событий. Она должна прояснить для вас понятие «хранилище событий», но для эксплуатации в производственной среде такая реализация не подходит.

¹ То есть товар добавлен в корзину заказов. — *Примеч. пер.*

² Строка добавлена в таблицу для корзины заказов. — *Примеч. пер.*

Реализация компонента `EventStore` включает в себя следующие этапы.

1. Добавить таблицу `EventStore` в базу данных `ShoppingCart`. В ней будет содержаться по строке для каждого порождаемого моделью предметной области события.
2. Воспользоваться библиотекой `Dapper` для реализации ответственной за запись событий части компонента `EventStore`.
3. Воспользоваться библиотекой `Dapper` для реализации ответственной за чтение событий части компонента `EventStore`.

Прежде чем заняться реализацией компонента `EventStore`, напомним, что представляет собой тип `Event` в микросервисе `Shopping Cart`:

```
public struct Event
{
    public long SequenceNumber { get; }
    public DateTimeOffset OccurredAt { get; }
    public string Name { get; }
    public object Content { get; }
    public Event(
        long sequenceNumber,
        DateTimeOffset occurredAt,
        string name,
        object content)
    {
        this.SequenceNumber = sequenceNumber;
        this.OccurredAt = occurredAt;
        this.Name = name;
        this.Content = content;
    }
}
```

События именно этого типа мы будем хранить в базе данных `Event Store`. Первый шаг — добавить показанную на рис. 5.9 таблицу в базу данных `ShoppingCart`. Находящийся в файле `Chapter05\ShoppingCart\src\database-scripts\create-shoppingcart-db.sql` в загружаемом коде сценарий базы данных создает в базе данных `ShoppingCart` эту, а также две другие таблицы.

EventStore			
	Column Name	Data Type	Allow Nulls
PK	ID	int	☐
	Name	nvarchar(100)	☐
	OccurredAt	datetimeoffset(7)	☐
	[Content]	nvarchar(MAX)	☐
			☐

Рис. 5.9. Таблица `EventStore` имеет четыре столбца для соответствующих категорий: идентификатора события, названия события, времени, когда произошло событие, и содержимого события

Далее добавим в микросервис `Shopping Cart` файл `EventStore.cs` и вставим в него следующий код для записи событий (листинг 5.3).

Листинг 5.3. Порождаем событие, что соответствует его сохранению

```

namespace ShoppingCart.EventFeed
{
    using System;
    using System.Threading.Tasks;
    using System.Collections.Generic;
    using System.Data.SqlClient;
    using System.Linq;
    using Dapper;
    using Newtonsoft.Json;
    public interface IEventStore
    {
        Task<IEnumerable<Event>> GetEvents(long firstEventSequenceNumber,
            long lastEventSequenceNumber);
        Task Raise(string eventName, object content);
    }

    public class EventStore : IEventStore
    {
        private string connectionString =
            @"Data Source=.\SQLEXPRESS;Initial Catalog=ShoppingCart;Integrated
            Security=True";

        private const string writeEventSql =
            @"insert into EventStore(Name, OccurredAt, Content) values
            (@Name, @OccurredAt, @Content)";
        public Task Raise(string eventName, object content)
        {
            var jsonContent = JsonConvert.SerializeObject(content);
            using (var conn = new SqlConnection(connectionString))
            {
                return
                    conn.ExecuteAsync(
                        writeEventSql,
                        new
                        {
                            Name = eventName,
                            OccurredAt = DateTimeOffset.Now,
                            Content = jsonContent
                        }
                    );
            }
        }
    }
}

```

Используем библиотеку Dapper для выполнения простого оператора вставки языка SQL

Данный код пока не компилируется, поскольку в интерфейсе `IEventStore` есть еще один метод, предназначенный для чтения событий. Эта реализация показана далее.

ПРИМЕЧАНИЕ

Сохранение событий, по существу, сводится к сохранению в строке таблицы `EventTable` JSON-сериализации содержимого события вместе с идентификатором события, его названием и моментом времени, в который событие было порождено. Возможно, концепция хранения событий и публикации их посредством ленты событий для вас нова, но реализация очень проста.

Вот и все, что нужно для реализации простейшего хранилища событий. Микросервис Shopping Cart теперь может порождать события в модели предметной области и рассчитывать на то, что компонент EventStore запишет их в таблицу EventStore базы данных ShoppingCart. Кроме того, у микросервиса Shopping Cart имеется лента событий, реализованная нами в главе 2 и использующая теперь компонент EventStore для чтения событий из базы данных, которые в дальнейшем он отправит подписчикам при опросе ленты (листинг 5.4).

Листинг 5.4. Метод компонента EventStore, предназначенный для чтения событий

```
private const string readEventsSql =
    @"select * from EventStore where ID >= @Start and ID <= @End";
public async Task<IEnumerable<Event>> GetEvents(
    long firstEventSequenceNumber,
    long lastEventSequenceNumber)
{
    using (var conn = new SqlConnection(connectionString))
    {
        return (await conn.QueryAsync<dynamic>(
            readEventsSql,
            new
            {
                Start = firstEventSequenceNumber,
                End = lastEventSequenceNumber
            }).ConfigureAwait(false))
            .Select(row =>
            {
                var content = JsonConvert.DeserializeObject(row.Content);
                return new Event(row.ID, row.OccurredAt, row.Name, content);
            });
    }
}
```

Читаем строки таблицы
EventStore от Start до End

Преобразуем строки таблицы
EventStore в объекты типа Event

Как упоминалось ранее, эта реализация хранилища событий не подходит для эксплуатации в производственной среде. Например, она столкнется с проблемами блокировок/конкуренции сразу же, как только микросервис начнет порождать события из нескольких параллельных потоков выполнения. Эти проблемы еще более усугубятся при масштабировании микросервиса на несколько серверов с потенциально несколькими одновременно порождающими события потоками выполнения каждый. Однако этот пример хорошо иллюстрирует концепцию хранения событий.

Хранение событий с помощью СУБД Event Store

Теперь мы реализуем другую версию компонента EventStore микросервиса Shopping Cart, на этот раз с помощью базы данных с открытым исходным кодом Event Store. Преимущество использования Event Store по сравнению с хранением событий в СУБД SQL Server состоит в том, что ее API приспособлено именно для хранения событий, их чтения и подписки на новые события. Event Store — тщательно продуманная проверенная реализация хранилища событий с открытым исходным кодом, хорошо масштабирующаяся и стабильно работающая под нагрузкой. Более

того, она ноставляется с набором удобных дополнительных возможностей нрямо «из коробки», таких как веб-интерфейс для нросмотра событий и лента событий редактора АТОМ. СУБД SQL Server, конечно, тоже тщательно продуманная, нроверенная, хорошо масштабируемая и стабильная, но она не нредназначена именно для хранения событий.

ПРИМЕЧАНИЕ

Эта реализация использует уже существующую, проверенную, хорошо масштабируемую, надежную реализацию хранилища событий. Она подходит для эксплуатации в производственной среде.

ПРЕДУПРЕЖДЕНИЕ

На момент написания данной книги клиентская библиотека для языка C# для взаимодействия с Event Store еще не адаптирована к .NET Core. Поэтому код, приводимый в данном разделе, необходимо выполнять с помощью полной версии фреймворка .NET, а следовательно, в операционной системе Windows.

Реализация этой версии будет состоять из следующих этапов.

1. Установить базу данных Event Store.
2. Организовать запись событий в Event Store с помощью компонента EventStore.
3. Организовать чтение событий из Event Store с помощью компонента EventStore.

По окончании получим полнофункциональную действующую реализацию компонента EventStore микросервиса Shopping Cart, основанного на использовании базы данных Event Store.

Скачать СУБД Event Store для различных платформ можно по адресу <http://geteventstore.com/downloads>. Скачиваемые файлы представляют собой ZIP-архивы, содержащие базу данных Event Store. Скачайте и разархивируйте соответствующий вашей платформе файл, и база данных Event Store будет установлена на вашей машине.

Теперь можно открыть командную оболочку и перейти в каталог, в который вы разархивировали скачиваемый файл с Event Store. Выполните следующую команду¹, чтобы запустить Event Store:

```
PS> ./EventStore.ClusterNode --db ./db --log ./logs
```

Проверить, работает ли она, можно, перейдя в браузере по адресу <http://127.0.0.1:2113/>. При этом вы должны увидеть приглашение на вход в систему, предоставляющее возможность входа с использованием имени пользователя `admin` и пароля `changeit`.

Чтобы использовать базу данных Event Store из микросервиса Shopping Cart, прежде всего необходимо добавить в проект пакет `EventStore.Client` системы управления пакетами NuGet. После этого можно реализовывать компонент EventStore для работы с базой данных Event Store. В листинге 5.5 приведен код для записи событий.

¹ Для операционной системы Windows можно не указывать в начале команды обозначение текущего каталога: `EventStore.ClusterNode --db ./db --log ./logs`. — *Примеч. пер.*

Листинг 5.5. Сохранение событий в базе данных Event Store

```

namespace ShoppingCart.EventFeed
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;
    using global::EventStore.ClientAPI;
    using Newtonsoft.Json;

    public class EventStore : IEventStore
    {
        private const string connectionString =
            "discover://http://127.0.0.1:2113/";
        private IEventStoreConnection connection =
            EventStoreConnection.Create(connectionString);

        public async Task Raise(string eventName, object content)
        {
            await connection.ConnectAsync().ConfigureAwait(false);
            var contentJson = JsonConvert.SerializeObject(content);
            var metaDataJson =
                JsonConvert.SerializeObject(new EventMetadata
                {
                    OccurredAt = DateTimeOffset.Now,
                    EventName = eventName
                });

            var eventData = new EventData(
                Guid.NewGuid(),
                "ShoppingCartEvent",
                isJson: true,
                data: Encoding.UTF8.GetBytes(contentJson),
                metadata: Encoding.UTF8.GetBytes(metaDataJson)
            );

            await
                connection.AppendToStreamAsync(
                    "ShoppingCart",
                    ExpectedVersion.Any,
                    eventData);
        }

        private class EventMetadata
        {
            public DateTimeOffset OccurredAt { get; set; }
            public string EventName { get; set; }
        }
    }
}

```

Создаем
соединение
с базой данных
EventStore

Открываем
соединение
с базой
данных
EventStore

Задаем соответствие
OccurredAt и EventName
метаданным,
которые нужно
сохранить вместе
с событием

EventData —
представление
события в базе
данных Event Store

Записываем событие
в базу данных Event Store

Этот код преобразует объект типа `Event` микросервиса `Shopping Cart` к типу `EventData` базы данных `Event Store`, после чего сохраняет результат в базе дан-

ных Event Store. В листинге 5.6 показана реализация чтения событий из базы данных Event Store.

Листинг 5.6. Чтение событий из базы данных Event Store

```
public async Task<IEnumerable<Event>>
    GetEvents(long firstEventSequenceNumber, long lastEventSequenceNumber)
{
    await connection.ConnectAsync().ConfigureAwait(false); // Читаем события из базы данных Event Store

    var result = await connection.ReadStreamEventsForwardAsync(
        "ShoppingCart",
        start: (int) firstEventSequenceNumber,
        count: (int) (lastEventSequenceNumber - firstEventSequenceNumber),
        resolveLinkTos: false).ConfigureAwait(false);

    return
        result.Events // Доступ к событиям в полученных из Event Store данных
            .Select(ev =>
                new
                {
                    Content = JsonConvert.DeserializeObject(
                        Encoding.UTF8.GetString(ev.Event.Data)),
                    Metadata = JsonConvert.DeserializeObject<EventMetadata>(
                        Encoding.UTF8.GetString(ev.Event.Data))
                })
            .Select((ev, i) =>
                new Event(
                    i + firstEventSequenceNumber,
                    ev.Metadata.OccurredAt,
                    ev.Metadata.EventName,
                    ev.Content));
}
```

Десериализация относящейся к содержимому части каждого из событий

Десериализация относящейся к метаданным части каждого из событий

Преобразование из объектов типа EventData базы данных Event Store в события [типа Event]

Этот код выполняет чтение событий из базы данных Event Store, десериализует относящиеся к содержимому и метаданным части событий, после чего преобразует их обратно к типу Event микросервиса Shopping Cart.

На этом реализация компонента EventStore на основе базы данных Event Store завершена. Рассмотрим теперь применение кэширования.

Установка заголовков кэширования в ответах Nancy

Вернемся к микросервисам, изображенным на рис. 5.2. Микросервис Shopping Cart использует информацию о товаре, получаемую путем запроса к микросервису Product Catalog, мы реализовали относящуюся к микросервису Shopping Cart часть этого взаимодействия в главе 2. А здесь сначала зададим заголовки кэширования в коде, реализующем конечную точку в микросервисе Product Catalog. После этого перенишем обращающийся к Product Catalog код из микросервиса Shopping Cart так, чтобы читать и использовать заголовок кэширования.

Пусть конечная точка `/products` микросервиса `Product Catalog` реализована в модуле `Nancy` под названием `ProductsModule`. Мы будем принимать в качестве параметра запроса список разделенных занятыми идентификаторами товаров. Эта конечная точка возвращает информацию о каждом из товаров, чей идентификатор внесен в этот список. Реализация аналогична модулям `Nancy`, которые вы встречали ранее в данной книге. Новое здесь только добавление в ответ заголовка `cache-control` в качестве разрешения клиентским приложениям кэшировать ответ на 24 часа (листинг 5.7).

Листинг 5.7. Добавление заголовков кэширования в список товаров

```
namespace ProductCatalog
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using Nancy;

    public class ProductsModule : NancyModule
    {
        public ProductsModule(ProductStore productStore) : base("/products")
        {
            Get("", _ =>
            {
                string productIdsString = this.Request.Query.productIds;
                var productIds = ParseProductIdsFromQueryString(productIdsString);
                var products = productStore.GetProductsByIds(productIds);

                return
                    this
                        .Negotiate
                        .WithModel(products)
                        .WithHeader("cache-control", "max-age:86400");
            });
        }

        private IEnumerable<int>
            ParseProductIdsFromQueryString(string productIdsString)
        {
            ...
        }
    }
}
```

Добавляем заголовок `cache-control`, указывая значение директивы `max-age` в секундах

Эта реализация добавляет в ответ заголовок `cache-control`, который выглядит следующим образом:

```
cache-control: max-age:86400
```

Этот заголовок указывает вызывающей стороне, что ответ может быть закэширован на срок, задаваемый максимальным значением, указываемым в секундах. В данном случае вызывающая сторона может кэшировать ответ на срок до 86 400 секунд (24 часа).

Чтение и использование заголовков кэширования

В главе 2 вы видели код, который выполнял обращения к конечной точке `/products` из микросервиса Shopping Cart. Это следующий код, он представляет собой часть класса `ProductCatalogClient` (листинг 5.8).

Листинг 5.8. Обращение к микросервису Product Catalog

```
private static async Task<HttpResponseBody>
    RequestProductFromProductCatalogue(int[] productCatalogueIds)
{
    var productsResource = string.Format(
        getProductPathTemplate, string.Join(",", productCatalogueIds));
    using (var httpClient = new HttpClient())
    {
        httpClient.BaseAddress = new Uri(productCatalogueBaseUrl);
        return await
            httpClient.GetAsync(productsResource).ConfigureAwait(false);
    }
}
```

При таком коде каждый раз, когда микросервису Shopping Cart понадобится информация о товаре, будет выполняться HTTP-запрос независимо от каких-либо заголовков кэширования. Такое поведение неэффективно в случаях, когда микросервису Shopping Cart несколько раз на протяжении 24 часов требуется информация об одних и тех же товарах, поскольку таковое заданное в ответах от микросервиса Product Catalog значение директивы `max-age`. Такие ситуации будут возникать всякий раз, когда пользователь добавляет в свою корзину заказов товар, который другой пользователь добавлял в свою корзину заказов на протяжении предшествующих 24 часов. Вполне вероятно, что такие ситуации будут происходить часто.

Расширим в микросервисе Product Catalog код, обращающийся к конечной точке `/products`, так, чтобы он учитывал заголовки кэширования. Добавим в `ProductCatalogClient` зависимость для кэша, реализующую интерфейс `ICache`:

```
private ICache cache;

public ProductCatalogClient(ICache cache)
{
    this.cache = cache;
}
```

Как вы помните, фреймворк Nancy внедряет зависимости вместо нас, так что при наличии в решении реализации интерфейса `ICache` Nancy внедрит ее. Тут я только показываю вам интерфейс, но в прилагаемом к книге коде вы найдете простую статическую реализацию этого интерфейса для кэша. Интерфейс прост и содержит два метода:

```
public interface ICache
{
    void Add(string key, object value, TimeSpan ttl);
    object Get(string key);
}
```

ttl означает время жизни (time to live)

Переменную `cache` класса `ProductCatalogClient` мы будем использовать, чтобы проверить перед выполнением HTTP-запроса, существует ли в кэше действительный объект (листинг 5.9).

Листинг 5.9. Выполнение запросов при отсутствии в кэше действительных ответов

```
private async Task<HttpResponseMessage>
➔ RequestProductFromProductCatalogue(int[] productCatalogueIds)
{
    var productsResource = string.Format(
        getProductPathTemplate, string.Join(",", productCatalogueIds));
    var response = this.cache.Get(productsResource) as HttpResponseMessage;
    if (response == null)
    {
        using (var httpClient = new HttpClient())
        {
            httpClient.BaseAddress = new Uri(productCatalogueBaseUrl);
            response = await
                httpClient.GetAsync(productsResource).ConfigureAwait(false);
            AddToCache(productsResource, response);
        }
    }
    return response;
}

private void AddToCache(string resource, HttpResponseMessage response)
{
    var cacheHeader = response
        .Headers
        .FirstOrDefault(h => h.Key == "cache-control");
    if (string.IsNullOrEmpty(cacheHeader.Key))
        return;
    var maxAge =
➔ CacheControlHeaderValue.Parse(cacheHeader.Value.ToString())
        .MaxAge;
    if (maxAge.HasValue)
        this.cache.Add(key: resource, value: response, ttl: maxAge.Value);
}

Выполняем
синтаксический
разбор значения
заголовка cache-control
и извлекаем из него
значение директивы
max-age
```

Пытаемся извлечь
действительный
ответ из кэша

Выполняем HTTP-запрос
только в том случае,
если в кэше ответа нет

Читаем из ответа
заголовок
cache-control

Добавляем ответ в кэш,
если в нем указано
значение директивы
max-age

После включения этого кода в микросервис `Shopping Cart` ответы от микросервиса `Product Catalog` будут использоваться столько времени, сколько допускает значение `max-age` в заголовке `cache-control` (в этом примере 24 часа).

5.4. Резюме

- ❑ Микросервис владеет всеми данными, относящимися к реализуемой им возможности, и хранит их.
- ❑ Микросервис — эталонный источник принадлежащей ему информации.
- ❑ Микросервис хранит свои данные в собственной специализированной базе данных.
- ❑ Микросервисы часто кэшируют принадлежащие другим микросервисам данные:
 - для снижения степени связанности с другими микросервисами. Это повышает стабильность системы в целом;
 - для ускорения обработки за счет снижения потребности в удаленных вызовах;
 - для создания пользовательских представлений (известны под названием моделей чтения), принадлежащих другим микросервисам данных, с целью упрощения кода;
 - для создания моделей чтения на основе порождаемых другими микросервисами событий с целью предотвращения лишних запросов к ним.

Таким образом, вместо взаимодействия на основе запросов используется взаимодействие на основе событий. Как вы помните из главы 4, взаимодействие на основе событий предпочтительнее в силу уменьшения связанности.

- ❑ Какую базу данных или базы данных использует микросервис — личное дело этого микросервиса. Различные микросервисы могут использовать разные базы данных.
- ❑ Хранение принадлежащих микросервису данных схоже с хранением данных в других разновидностях систем.
- ❑ Для чтения данных из SQL базы данных и записи данных в нее можно использовать библиотеку Dapper.
- ❑ Сохранение событий, по сути, сводится к записи сериализованных событий в базу данных.
- ❑ Простейшая версия хранилища событий включает сохранение событий в таблице в SQL-базе данных.
- ❑ Можно также реализовать хранилище событий, сохранив события в базе данных Event Store с открытым исходным кодом, специально предназначенной для хранения событий.

6

Проектирование устойчивых к ошибкам систем

В этой главе:

- ❑ устойчивое к ошибкам взаимодействие между микросервисами;
- ❑ обеспечение на вызывающей стороне устойчивости к ошибкам при отказах;
- ❑ свертывание и развертывание;
- ❑ реализация устойчивого к ошибкам взаимодействия.

Эта глава познакомит вас со стратегиями обеспечения устойчивости системы микросервисов к ошибкам в случае отказов. В целом при любом взаимодействии одного микросервиса с другим существует вероятность сбоя связи. В этой главе вы изучите шаблоны, позволяющие справиться с подобными сбоями, и реализуете некоторые из них. Эти стратегии очень просты, но все же они существенно усиливают устойчивость системы к ошибкам.

Отказы и ошибки

Мы будем четко различать термины «отказ/сбой» (*failure*) и «ошибка» (*error*). Отказ происходит при сбое в системе, причина которого лежит вне системы. Вот некоторые типичные источники отказов.

- Потеря сетевых пакетов приводит к отказу связи.
- Потеря соединения приводит к отказу связи.
- Отказы аппаратного обеспечения приводят к отказам микросервисов.

Ошибка же означает, что система обслуживает пользователей неудовлетворительным образом. Вот некоторые примеры ошибок.

- Пользователь видит страницу с сообщением об ошибке.
- Система зависает и перестает реагировать на действия пользователя.
- Система возвращает некорректный ответ на действие пользователя.

Ошибки часто являются результатом отказов. В то же время отказы превращаются в ошибки только тогда, когда программное обеспечение не способно справиться с ними. Следовательно, в идеальной системе возможны отказы, но не ошибки. К сожалению, наши системы неидеальны, так что приходится мириться с вероятностью появления ошибок.

6.1. Ожидайте отказов

Нужно быть готовыми к отказам при работе с любой нетривиальной программной системой. Может произойти сбой аппаратного обеспечения. Или отказ программного обеспечения из-за непредусмотренного сценария использования или поврежденных данных. Один из отличительных признаков системы микросервисов состоит в активном взаимодействии между микросервисами. На рис. 6.1 продублирована диаграмма из главы 1, демонстрирующая взаимодействия, происходящие при добавлении пользователем товара в корзину заказов. Как видите, выполнение пользователем одного-единственного действия приводит к обширному взаимодействию, а в реальной системе, вероятно, будет множество пользователей, одновременно выполняющих много действий, а следовательно, масса взаимодействий. Нужно быть готовыми к случающимся время от времени сбоям связи. Вероятно, сбои связи между двумя конкретными микросервисами не будут слишком частыми, но в системе микросервисов в целом отказы связи будут нередким явлением в силу объема взаимодействий.

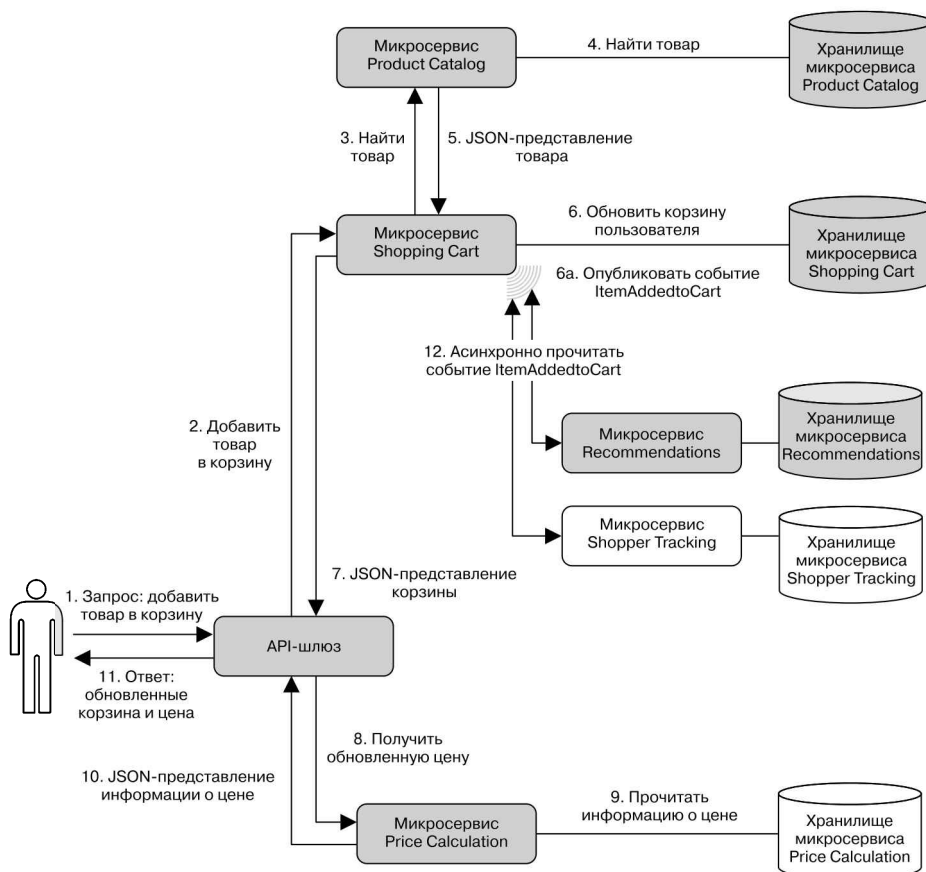


Рис. 6.1. В системе микросервисов существует множество путей взаимодействия

А раз мы ожидаем сбоев некоторых из этих взаимодействий в нашей системе микросервисов, то должны проектировать микросервисы так, чтобы они могли справиться с этими сбоями. Как обсуждалось в главе 4, взаимодействия микросервисов можно разделить на три категории.

- ❑ *Взаимодействие на основе запросов.* При неудачном выполнении запроса вызывающая сторона не получает нужной информации. Если вызывающая сторона способна с этим справиться, система продолжает функционировать, но с сокращенной функциональностью. Если же нет, результатом действия будет ошибка.
- ❑ *Взаимодействие на основе команд.* При сбое отправки сообщения отправитель не знает, дошла ли команда до получателя. Это может привести также к ошибке или сокращению функциональности в зависимости от того, насколько хорошо справляется с ситуацией сервер.
- ❑ *Взаимодействие на основе событий.* При сбое вызова во время опроса ленты событий подписчиком последствия не фатальны. Подписчик позднее снова опросит ленту событий и, если она уже будет работать, получит события. Другими словами, подписчик все равно получает все события, но некоторые из них — с задержкой. Это не должно вызывать проблем при взаимодействии на основе событий, поскольку оно все равно асинхронное.

Последствия отказа при взаимодействии зависят от типа взаимодействия и того, как микросервисы с ним справляются.

В следующем подразделе обсуждаются некоторые важные аспекты подготовки к надежной обработке отказов.

Наличие подробных журналов

Осознав, что отказы неизбежны и некоторые из них могут привести не только к ухудшению опыта взаимодействия конечного пользователя, но и к ошибкам, вы обязаны убедиться, что при возникновении ошибки сможете понять, что именно пошло не так. Это значит, что вам нужны подробные журналы, с помощью которых можно проследить, что происходило в системе и привело к ошибочной ситуации. Произошедшее зачастую охватывает несколько микросервисов, поэтому желательно ввести в систему централизованный микросервис Log (рис. 6.2). Все остальные микросервисы будут отправлять ему журнальные записи, доступные для просмотра и поиска в них при необходимости.

Микросервис Log — центральный компонент, используемый всеми остальными микросервисами. Нужно гарантировать, что сбой в микросервисе Log не приведет к краху всей системы по причине сбоя остальных микросервисов, утративших возможность журналирования сообщений. Следовательно, отправка сообщений микросервису Log должна работать по принципу «сделать и забыть»: сообщения отправляются, после чего отправляющий их микросервис про них забывает, а не ждет ответа.

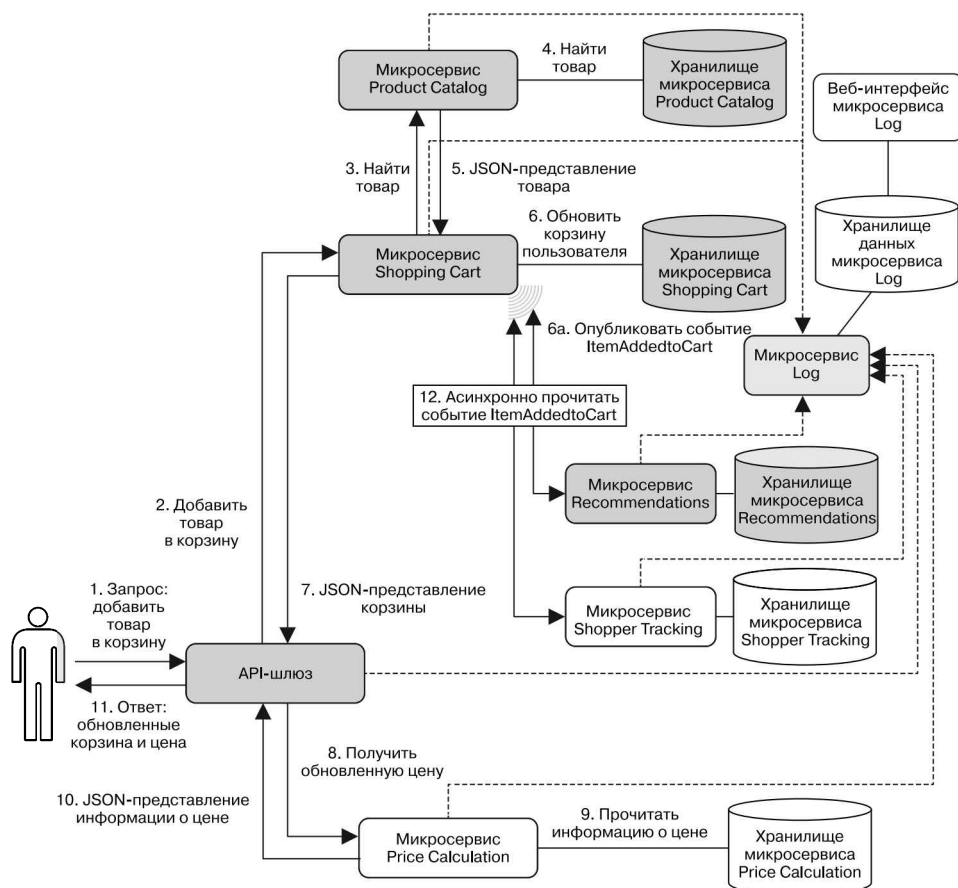


Рис. 6.2. Централизованный микросервис Log получает журнальные сообщения от всех остальных микросервисов и сохраняет их в базе данных или поисковой системе. Данные журнала доступны через веб-интерфейс. Пунктирные стрелки показывают отправку микросервисами журнальных сообщений в централизованный микросервис Log

Использование готового решения для микросервиса Log

Централизованный микросервис Log не реализует бизнес-возможность конкретной системы. Он представляет собой реализацию общей технической возможности. Другими словами, требования к микросервису Log системы А не слишком сильно будут отличаться от требований к микросервису Log системы В. Следовательно, я рекомендую использовать для реализации микросервиса Log готовое решение.

Например, журналы можно хранить в поисковой системе Elasticsearch (<https://github.com/elastic/elasticsearch>), а доступ к ним организовывать с помощью панели инструментов Kibana (<https://github.com/elastic/kibana>). Это широко известные и хорошо документированные продукты, и здесь я не буду углубляться в вопросы их установки и настройки. В главе 9 будет предполагаться наличие у вас микросервиса Log на основе Elasticsearch, и я продемонстрирую, как отправлять туда журнальные сообщения.

Далее в этой главе мы рассмотрим вопрос журналирования обработанных ошибок посредством добавления обработчиков в копвейер ошибок Nancy.

Использование маркеров корреляции

Для поиска всех журнальных сообщений, связанных с конкретным действием в системе, можно использовать *маркеры корреляции (correlation tokens)*. Маркер корреляции — это идентификатор, прилагаемый, например, к поступающему в систему запросу от конкретного пользователя. Маркер корреляции передается от микросервиса к микросервису при любом взаимодействии, в основе которого лежит этот запрос конкретного пользователя. Всякий раз, когда один из микросервисов отправляет журнальное сообщение микросервису Log, он должен включать в это сообщение маркер корреляции. Микросервис Log должен обеспечивать возможность поиска журнальных сообщений по маркеру корреляции. На рис. 6.2 шлюз API создает и присваивает маркер корреляции каждому входящему запросу, и этот маркер должен передаваться при каждом взаимодействии «микросервис — микросервис», включая события и журнальные сообщения.

ПРИМЕЧАНИЕ

Реализация журналирования запросов и включение маркеров корреляции при взаимодействии и записи журнальных сообщений обсуждается в главе 9.

Развертывание и свертывание

При возникновении ошибок во время эксплуатации в производственной среде возникает вопрос: как их исправить? В большинстве традиционных систем при возникновении ошибок вскоре после развертывания применяется стандартное решение: откатиться к предыдущей версии системы. В системе микросервисов стандартным может оказаться иное решение. Как обсуждалось в главе 1, микросервисы подходят для непрерывной доставки ПО. При непрерывной доставке ПО развертывание микросервисов происходит довольно часто, так что каждое развертывание должно выполняться легко и быстро. Более того, микросервисы достаточно малы и просты для того, чтобы большинство исправлений ошибок не представляло сложностей. Это дает возможность выполнить обновления до следующей версии — *развертывания (rolling forward)* вместо возврата к предыдущей — *свертывания (rolling backward)*.

Возникает вопрос: зачем делать развертывание решением по умолчанию? В некоторых ситуациях свертывание довольно сложно — в частности, когда дело связано с изменениями базы данных. При развертывании новой версии, меняющей базу данных, микросервисы начинают порождать данные, подходящие для обновленной базы данных. А данные, помещенные в базу данных, остаются там и могут оказаться несовместимыми с возвратом к предыдущей версии. В подобном случае лучше выполнить развертывание.

Не допускайте распространения сбоев

Иногда в окружающей микросервис системе происходит что-то, нарушающее его нормальное функционирование. Об этой ситуации мы будем говорить: данный микросервис испытывает *стрессовую нагрузку*. Существует множество источников стрессовой нагрузки, в том числе:

- ❑ фатальный сбой одной из машин в кластере, на которой хранятся данные этого микросервиса;
- ❑ потеря микросервисом сетевого соединения с одним из взаимодействующих с ним микросервисов;
- ❑ получение микросервисом необычно высоких объемов трафика;
- ❑ прекращение работы одного из взаимодействующих с ним микросервисов.

Во всех этих случаях испытывающий стрессовую нагрузку микросервис не может продолжать нормальное функционирование. Это не значит, что он перестает работать, но ему пужно как-то справиться с трудностями.

При отказе одного из микросервисов взаимодействующие с ним микросервисы оказываются под стрессовой нагрузкой. Это значит, что существует риск и их отказов. При сбое микросервиса взаимодействующие с ним микросервисы не могут выполнять запросы, отправлять команды или опрашивать ленту событий отказавшего микросервиса. Как показано на рис. 6.3, при их сбое возникает риск дальнейших отказов и других микросервисов — сбой начинает распространяться по системе микросервисов. Подобная ситуация может быстро перерасти из отказа одного микросервиса в отказ многих.

Вот несколько примеров того, как остановить распространение отказов.

- ❑ Когда микросервис отправляет команду другому микросервису, который в этот момент отказывает, запрос завершится неудачно. Если отправитель команды при этом тоже отказывает, мы получаем изображенную на рис. 6.3 ситуацию распространения отказов по всей системе. Для прекращения распространения отказов отправитель может, например, вести себя так, как будто команда была выполнена успешно, при этом сохраняя ее в списке команд, завершившихся неудачно. Отправляющий команды микросервис может периодически просматривать список завершившихся неудачно команд и пытаться отправить их снова. Это возможно не во всех случаях, поскольку команда может быть такой, которую необходимо обработать немедленно, по там, где такой подход допустим, он позволяет прекратить распространение сбоя в одном микросервисе по системе. Этот подход можно сочетать с применением шаблона «Предохранитель», о котором мы поговорим далее в этой главе.
- ❑ При выполнении одним микросервисом запроса к другому микросервису, который в этот момент отказывает, вызывающая сторона может использовать кэшированный ответ. В главе 5 вы видели, как выполнять кэширование ответов на запросы и обеспечивать учет задаваемых опрашиваемым микросервисом заголовков кэширования. Если паходящийся в кэше у вызывающей стороны ответ устарел, а запрос на более свежий ответ завершился неудачей, все равно можно использовать устаревший ответ. Не во всех ситуациях, но когда, возможно, это остановит распространение отказов.

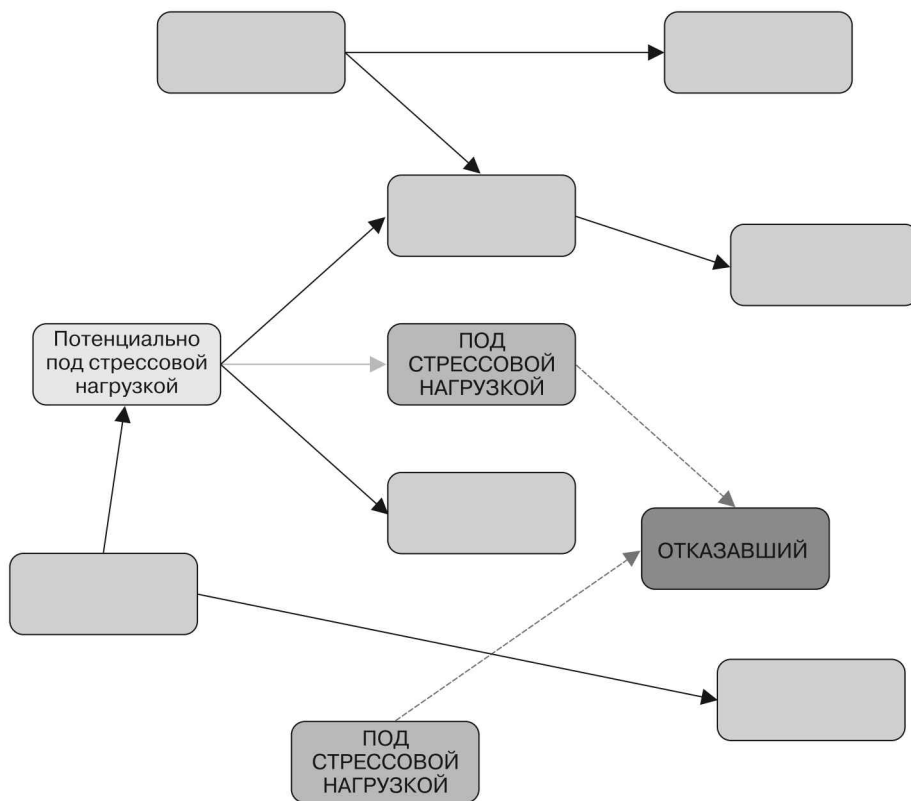


Рис. 6.3. Если микросервис, помеченный как отказавший, отказал, то перестает функционировать и взаимодействие с ним. Это значит, что взаимодействующие с ним микросервисы оказываются под стрессовой нагрузкой. Если эти микросервисы также отказывают, то взаимодействующие с ними микросервисы, в свою очередь, оказываются под стрессовой нагрузкой. В этом случае сбой в одном микросервисе может распространиться на множество других микросервисов

- ❑ Шлюз API, паходящийся под стрессовой пагрузкой из-за большого объема трафика от копкретного клиента, может пемного ограпичить последнего, отвечая только па определенное число запросов в секунду от него. Обратите внимание па то, что отправка этим клиентом пеобычно большого количества запросов может быть обусловлепа его впутренним сбоем. При ограпичении взаимодействия этого клиентпа будут песколько затруднены, но оп все равно стапет получать пекотрые ответы. Если же ограпичения пе будет, шлюз API постепенно пацпет медленпо обслуживать всех клиентпов или откажет полпостью. Более того, поскольку шлюз API взаимодействует с другими микросервисами, медленпая обработка всех входящих запросов поставит под стрессовую пагрузку и эти микросервисы. Так что ограпичение предотвращает распространение отказа одного клиентпа на другие микросервисы.

Как вы можете видеть из этих примеров, предотвращать распространение отказов можно различными способами. Отсюда можно сделать важный вывод: необходимо встраивать в свою систему специальные средства защиты от распространения ожидаемых вами видов отказов. Реализация этого зависит от специфики создаваемой системы. Встраивание средств защиты может потребовать определенных усилий, но часто оно вполне оправдывает себя, придавая системе в целом устойчивость к ошибкам.

6.2. Ответственность за устойчивость к ошибкам на стороне клиента

При взаимодействии двух микросервисов один из них играет роль сервера, а другой — клиента (рис. 6.4). Клиент — это микросервис, который отправляет HTTP-запрос, а сервер — микросервис, обрабатывающий данный запрос. Показанный на рис. 6.4 запрос завершается неудачно. Это может произойти из-за отказа сервера или сбоя связи с сервером. Если запрос завершился неудачно, сервер ничего с этим поделать не может. Сервер не может отправить ответ на этот запрос, ведь запрос к этому времени уже завершен. Следовательно, ответственность за обработку запросов должна ложиться на клиентские приложения. Другими словами, клиент отвечает за обеспечение устойчивости к ошибкам взаимодействия в случае неудачного завершения запросов.

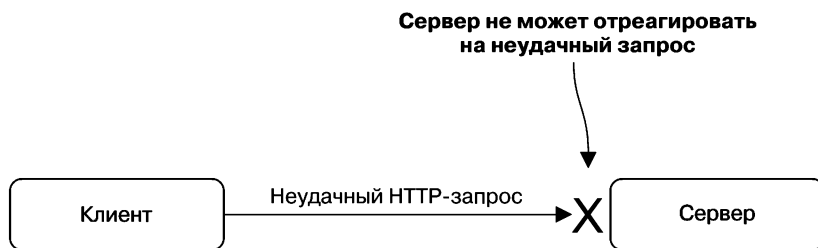


Рис. 6.4. Во всех взаимодействиях между микросервисами есть клиент и сервер. Клиент отправляет серверу HTTP-запросы. И он же отвечает за обработку запросов, которые завершились неудачно

Что касается взаимодействия на основе событий, некоторая степень устойчивости к ошибкам в случае неудачного завершения запросов встроена в саму схему взаимодействия. На рис. 6.5 показаны подписчик на события одного микросервиса и лемпа событий — другого. Как вы видели в главе 3, подписчик периодически опрашивает лемпу на предмет новых событий. Такой способ взаимодействия означает, что при неудачном завершении запроса о новых событиях подписчик запросит те же события снова в следующий раз, когда будет выполнять опрос. Подписчик сможет получить события из лемпы, несмотря на неудачное завершение части запросов, следовательно, подписчик устойчив к сбоям запросов событий.

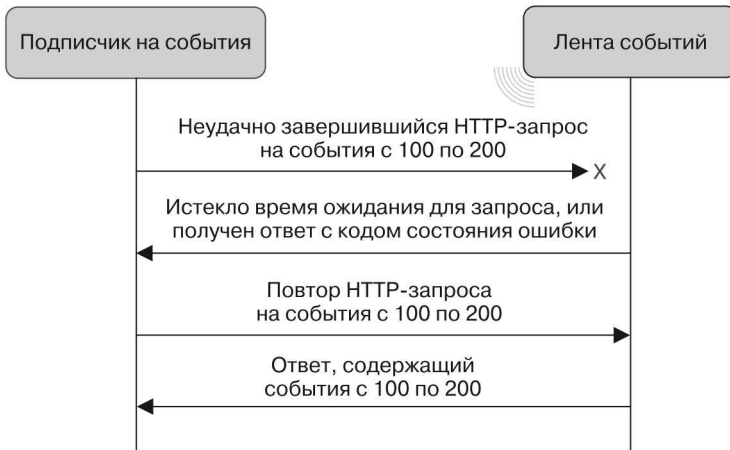


Рис. 6.5. В случае неудачного завершения запроса событий из ленты подписчик запрашивает те же события при следующем опросе

Как видите, в случае взаимодействия на основе команд или запросов устойчивости к ошибкам добиться не просто. В следующих двух разделах мы обсудим шаблоны обеспечения устойчивости к ошибкам при этих типах взаимодействия.

Шаблон устойчивости к ошибкам «Повтор»

Клиент при взаимодействии на основе команд или запросов может повторить попытку позднее, если запрос завершился неудачно. Если причина неудачи запроса была временной, следующая попытка может оказаться успешной. Временные отказы — частое явление, среди их причин следующие:

- ❑ перегруженность сети;
- ❑ микросервис-сервер находится в процессе развертывания. В зависимости от метода развертывания микросервиса в течение короткого промежутка времени, например при переключении балансировщика нагрузки на новую версию, он может быть недоступен или будет работать медленно. Даже если сервер работает медленнее только во время развертывания, запросы могут завершаться неудачно из-за истечения времени ожидания.

Повтор — палка о двух концах. Если причины сбоев не были временными, повторная отправка запросов не поможет. Напротив, повторная отправка всех запросов без разбора ставит под стрессовую нагрузку сервер, получающий, помимо обычного количества запросов, еще и повторные (рис. 6.6). Может показаться, что не стоит из-за этого волноваться, но представьте себе систему, уже падающую под высокой нагрузкой. При обычном функционировании клиент отправляет серверу множество запросов. Если запросы завершаются неудачно и клиент повторяет их все, то он отправляет серверу все больше и больше запросов. А если причина сбоя запросов и заключалась в том, что сервер не справлялся с поступающим количеством запросов, отправка дополнительных определенно не улучшит ситуацию.

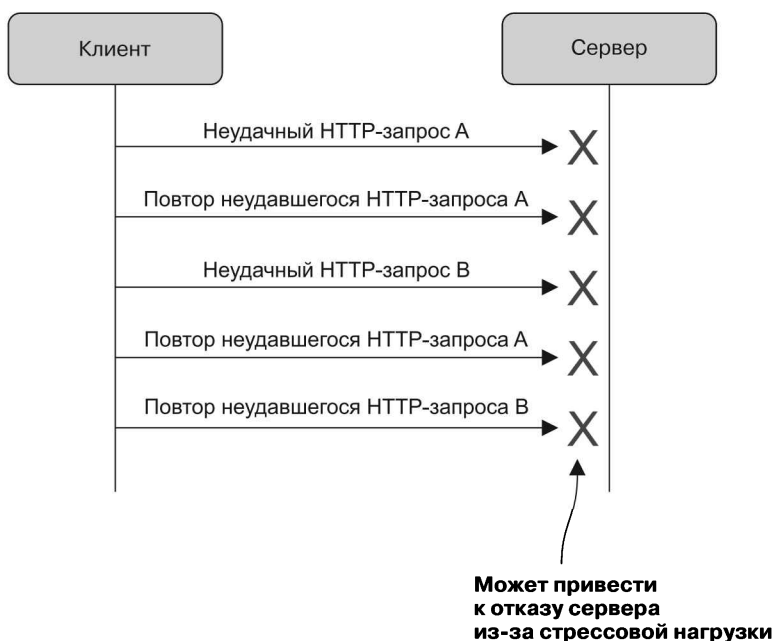


Рис. 6.6. Если клиент снова и снова повторяет запрос, каждый раз завершающийся неудачно, стрессовая нагрузка на сервер будет только расти, и в конце концов это приведет к полному отказу сервера

Значит ли это, что «Повтор» — плохой шаблон? Нет, это значит, что не следует выполнять повторы слишком упорно. Первое, что стоит обдумать: сколько раз имеет смысл повторять запрос. Если запрос завершается неудачно три раза подряд, то есть ли основания надеяться, что он завершится успешно в четвертый? Можно использовать экспоненциальную *задержку отправки* между повторами. То есть вместо того, чтобы выполнять повтор через неизменный промежуток времени (допустим, 100 мс), ожидать в два или три раза дольше после каждой последующей отправки, например, 100 мс — перед первым повтором, 200 мс — между первым и вторым и 400 мс — между вторым и третьим. Эти два простых дополнения обеспечивают более медленное наращивание стрессовой нагрузки на сервер. Лучше всегда использовать их при повторях команд и запросов. Далее в этой главе я покажу вам, как упростить настройку подобных стратегий повтора.

Возможно, вы даже захотите сделать интервал между повторами намного более длительным, например ждать перед повтором не долю секунды, а несколько минут или даже часов. На рис. 6.7 показана стратегия повтора, при которой интервалы длинные и возрастают экспоненциально. При подобной стратегии повтора сервер подвергается намного меньшей стрессовой нагрузке, чем при частых повторях, используемых во многих программных системах.

Очевидно, что этот подход не может работать во всех ситуациях. Если пользователь ждет ответа, не имеет смысла повторять запрос через час, поскольку он перестает ждать задолго до этого, так что программному обеспечению тоже следует отказаться от дальнейших попыток и выдать ответ, пусть и не такой качественный, поразнее.

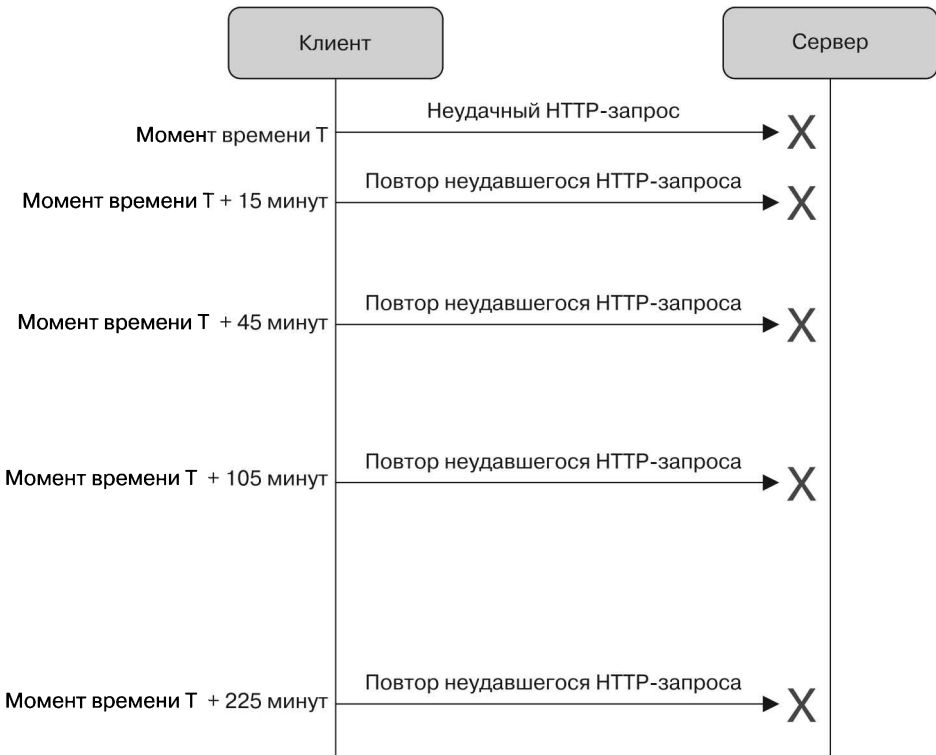


Рис. 6.7. Во избежание слишком сильной нагрузки на сервер клиент может сделать промежутки ожидания между повторами увеличивающимися в геометрической прогрессии

В то же время, если запрос оспаривается по чему-то, что система делает для своих собственных нужд, например, в качестве части обработки события, зачастую можно подождать длительное время.

Далее мы обсудим еще один удобный шаблон обеспечения устойчивости к ошибкам взаимодействия — «Предохранитель». Затем перейдем к коду и реализуем стиль как частых, так и редких повторов.

Шаблон устойчивости к ошибкам «Предохранитель»

Шаблон «Предохранитель» — еще один способ решения проблемы со сбоями запросами. Как вы видели в предыдущем разделе, повтор запросов может усугубить проблему — подвергнуть сервер стрессовой нагрузке, следовательно, необходимо ограничить число повторов. Шаблон «Предохранитель» идет в этом смысле еще дальше, предполагая, что, если несколько различных запросов подряд завершились неудачно, следующий запрос, вероятно, постигнет та же участь.

Эта ситуация проиллюстрирована на рис. 6.8. Клиент уже выполнил HTTP-запросы A, B, C и D. Можно ли ожидать, что запрос E завершится успешно? В большинстве случаев — нет. Тот факт, что несколько запросов завершились неудачно, показывает, что проблема не в самих запросах. Скорее речь идет о проблеме со

связью — сервер недоступен, происходит отказ сервера или клиент отправляет испорченные запросы. Проблема со связью может быть временной, но даже в этом случае запрос E часто завершается неудачно.

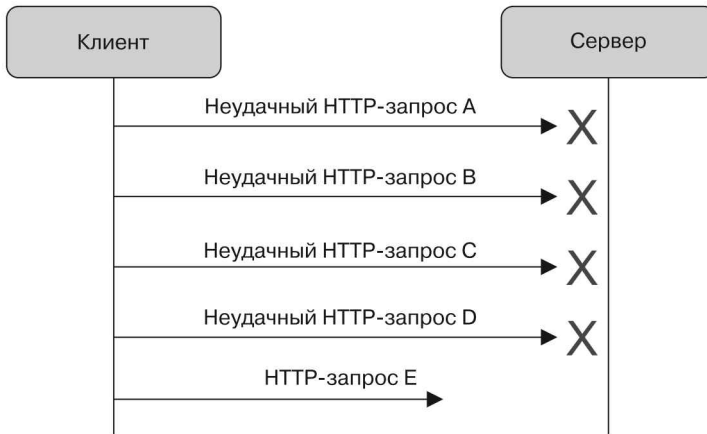


Рис. 6.8. Если несколько различных запросов подряд завершились неудачно, следует ли ожидать неудачного завершения следующего запроса? В большинстве случаев — да

Шаблон «Предохранитель» решает эту проблему, предполагая, что запрос E завершится неудачно, и вообще отменяя его отправку. Предотвращение отправки запросов с большой вероятностью неудачи снижает стрессовую нагрузку как па клиента, так и па сервер.

- ❑ Сервер получает меньше запросов.
- ❑ Клиенту не приходится ждать неудачного завершения запросов, он просто ранее предполагает, что это произойдет, благодаря чему расходует ресурсы не па ожидание, а па решение собственных задач.

Предохранитель обортывает HTTP-запросы в копечный автомат, подобный показанному па рис. 6.9. Микросервис, которому необходимо выполнить HTTP-запрос, делает это через предохранитель.

Копечный автомат предохранителя начинает выполнение с замкнутого состояния и работает следующим образом.

- ❑ Когда предохранитель паходится в *замкнутом состоянии (closed state)*, он выполняет по требованию HTTP-запросы. Если HTTP-запрос завершается неудачно, предохранитель увеличивает счетчик на единицу, если же успешно, то сбрасывает в поль. Если этот счетчик превышает предварительное заданное предельное значение, скажем пять неудачных запросов подряд, предохранитель переходит в разомкнутое состояние.
- ❑ Когда предохранитель паходится в *разомкнутом состоянии (open state)*, он не выполняет никаких HTTP-запросов. Вместо этого он немедленно возвращает ошибку. Предохранитель остается в разомкнутом состоянии в течение предварительно заданного промежутка времени — скажем, 30 секунд, — после чего переходит в полуразомкнутое состояние.

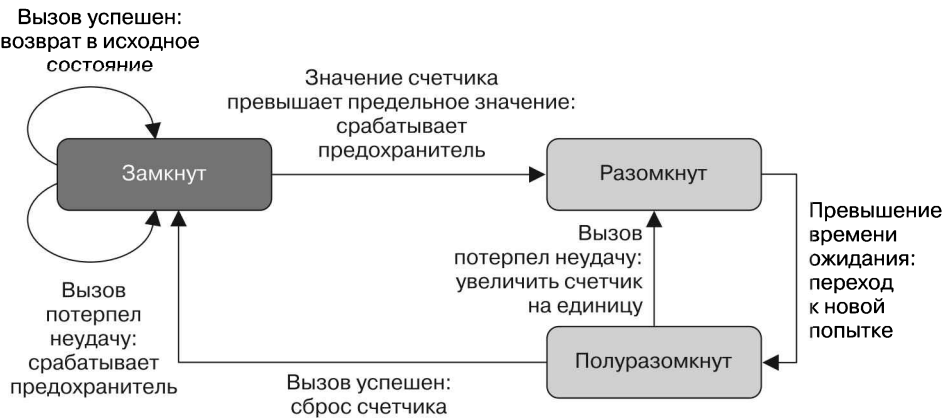


Рис. 6.9. Предохранитель — конечный автомат с тремя состояниями. Когда он замкнут, выполняются HTTP-запросы, когда разомкнут, запросы не производятся. Предохранитель помогает избежать выполнения HTTP-запросов, вероятность неудачного завершения которых высока

- Когда предохранитель находится в *полуразомкнутом состоянии (half-open state)*, он выполняет HTTP-запрос, когда от него требуют это в первый раз. После одного HTTP-запроса он переходит в замкнутое состояние, если запрос успешен, и в разомкнутое — если нет.

В результате выполнения этих простых правил мы получаем конечный автомат, предотвращающий выполнение тех HTTP-запросов, которые в любом случае, вероятно, завершатся неудачей. Далее в этой главе я покажу вам, как использовать библиотеку Polly для создания адаптеров-предохранителей для HTTP-запросов.

СОВЕТ

Предохранители могут принести пользу не только для HTTP-запросов, их можно использовать для повышения устойчивости к ошибкам любой операции, которая может завершиться неудачей.

В оставшейся части этой главы мы обратимся к исходному коду и рассмотрим реализацию стратегии повторов и предохранителей с помощью библиотеки Polly и общей обработки ошибок с помощью конвейеров ошибок фреймворка Nancy.

6.3. Реализация шаблонов устойчивости к ошибкам

Чтобы увидеть, как реализовать шаблоны «Повтор» и «Предохранитель», обсуждавшиеся в предыдущих разделах, вернемся к POS-системе из главы 3 и рассмотрим подробнее взаимодействия, происходящие вокруг микросервиса Loyalty Program. Эти взаимодействия мы определили в главе 4. На рис. 6.10 они показаны повторю, аннотированные стратегиями устойчивости, которые вы реализуете в следующих разделах. Мы не будем рассматривать код для стратегий устойчивости микросервисов Invoice и Log, поэтому они неактивны.

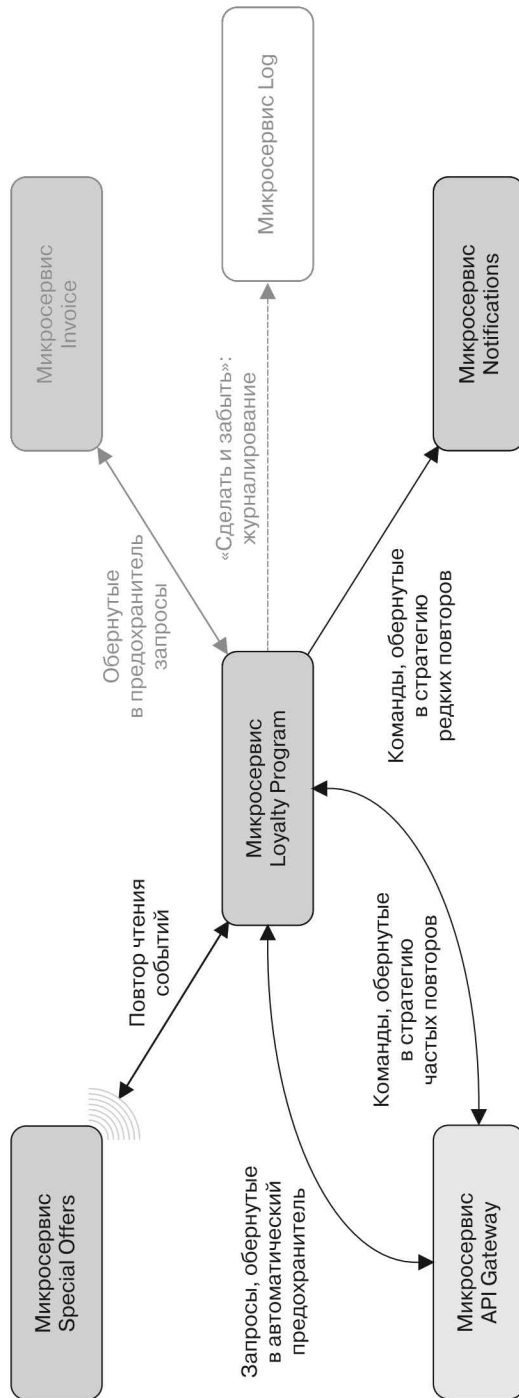


Рис. 6.10. Микросервис Loyalty Program взаимодействует с несколькими другими микросервисами. Для каждого взаимодействия указана стратегия устойчивости к ошибкам

Далее мы будем заниматься следующим.

- ❑ Реализовывать стратегию частых повторов в микросервисе API Gateway для отправляемых им микросервису Loyalty Program команд. Реализация будет основана на библиотеке Polly.
- ❑ Реализовывать шаблон «Предохранитель» в микросервисе API Gateway для запросов, адресуемых им микросервису Loyalty Program. Эта реализация также будет основана на библиотеке Polly.
- ❑ Реализовывать стратегию редких повторов в микросервисе Loyalty Program для отправляемых им микросервису Notifications команд, основанную на традиционном стиле функционирования подписки на события.
- ❑ Реализовывать общие обработчики исключений в компоненте HTTP API в микросервисе Loyalty Program с помощью встроенного в Nancy конвейера ошибок.

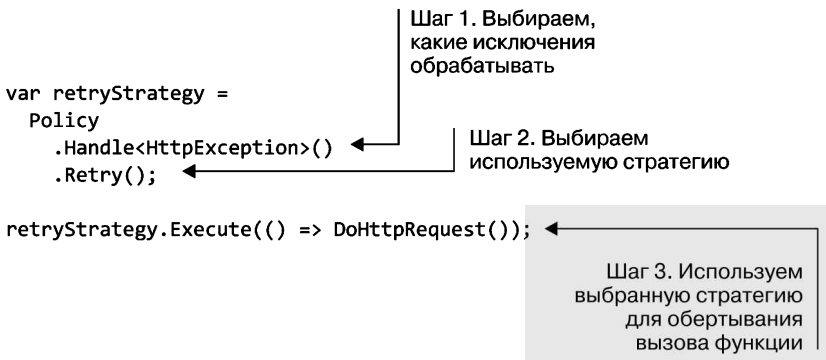
Polly

Polly — удобная библиотека для создания и использования стратегий обработки ошибок. Создаются стратегии с помощью Polly путем описания посредством переменного API. После создания стратегию можно использовать для любой Func или Action, то есть, по сути, для любого кода.

Вот три этапа использования библиотеки Polly.

1. Выбрать обрабатываемые исключения, например `HttpException`.
2. Выбрать используемую стратегию, например стратегию повторов.
3. Применить эту стратегию к функции.

Точка входа для работы с библиотекой Polly — класс `Policy`:



Библиотека Polly поставляется с множеством встроенных стратегий, включая несколько разновидностей стратегии повторов и различные стратегии предохранителей.

Микросервис API Gateway состоит из показанных на рис. 6.11 компонент. В следующих двух разделах рассмотрим подробнее компонент `LoyaltyProgramClient`.

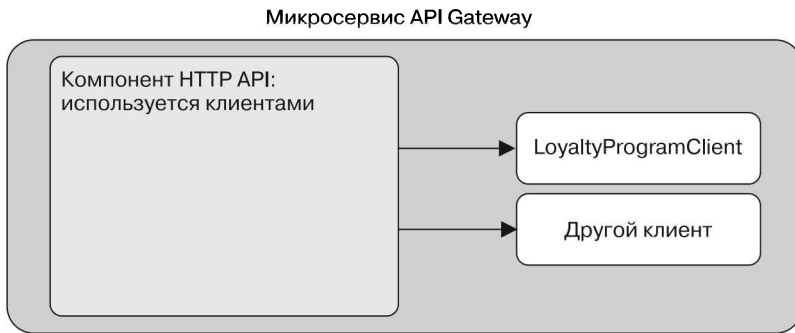


Рис. 6.11. Микросервис API Gateway состоит из того же стандартного набора компонентов, с которым вы уже несколько раз сталкивались

Реализация стратегии частых повторов с помощью библиотеки Polly

Как показано на рис. 6.10, микросервис API Gateway отправляет команды микросервису Loyalty Program. Мы рассмотрим только добавление стратегии повторов к команде регистрации пользователя, поскольку код добавления стратегии повторов к команде обновления данных пользователя ничем, по сути, не отличается.

Вначале необходимо добавить пакет Polly системы управления пакетами NuGet в микросервис API Gateway. Отправляющий команды микросервису Loyalty Program код находится в компоненте `LoyaltyProgramClient`. Мы воспользуемся библиотекой Polly для настройки стратегии повторов с применением экспоненциальной задержки отправки. Polly отделяет настройку стратегии от ее выполнения, то есть предоставляет возможность задавать различные стратегии, например стратегию повторов, а затем выполнять фрагмент кода под прикрытием этой стратегии. В случае стратегии повторов это означает повторное выполнение фрагмента кода в случае неудачи. Стратегия повторов для команды регистрации пользователя задается следующим образом (листинг 6.1).

Листинг 6.1. Стратегия повторов, реализованная с помощью библиотеки Polly

```
using System;
using Polly;

public class LoyaltyProgramClient
{
    private static Policy exponentialRetryPolicy =
        Policy
            .Handle<Exception>() ← Обрабатываются все исключения
            .WaitAndRetryAsync( ← Выбирается асинхронная стратегия, поскольку мы будем далее использовать ее для асинхронного кода
                3, ← Количество попыток
                attempt =>
                    TimeSpan.FromMilliseconds(100 * Math.Pow(2, attempt)), ← Время ожидания перед следующим повтором
            );
}
```

После пастройки этой стратегии ее можно использовать для обертывания обращения к микросервису Loyalty Program (листинг 6.2).

Листинг 6.2. Использование стратегии библиотеки Polly для HTTP-запроса

```
public async Task<HttpResponseMessage>
    RegisterUser(LoyaltyProgramUser newUser)
{
    return await exponentialRetryPolicy
        .ExecuteAsync(() => DoRegisterUser(newUser));
}

private async Task<HttpResponseMessage>
    DoRegisterUser(LoyaltyProgramUser newUser)
{
    using (var httpClient = new HttpClient())
    {
        httpClient.BaseAddress = new Uri($"http://{this.hostName}");
        var response = await
            httpClient.PostAsync("/users/",
                new StringContent(JsonConvert.SerializeObject(newUser),
                    Encoding.UTF8,
                    "application/json"));
        ThrowOnTransientFailure(response);
        return response;
    }
}

private static void ThrowOnTransientFailure(HttpResponseMessage response)
{
    if (((int) response.StatusCode) < 200 ||
        ((int) response.StatusCode) > 499)
        throw new Exception(response.StatusCode.ToString());
}
```

Выполняем действие с использованием стратегии повторов

Выполняем HTTP-запрос

Генерируем исключение для указания стратегии на необходимость повтора

Вот как просто можно настроить стратегию повторов с помощью библиотеки Polly. Далее мы воспользуемся Polly для создания предохранителя.

Реализация предохранителя с помощью библиотеки Polly

Теперь добавим предохранитель к запросам, которые микросервис API Gateway выполняет к микросервису Loyalty Program. На этот раз воспользуемся встроенной поддержкой стратегий предохранителей библиотеки Polly (листинг 6.3).

Листинг 6.3. Стратегия предохранителя библиотеки Polly

```
private static Policy circuitBreaker =
    Policy
        .Handle<Exception>()
        .CircuitBreaker(5, TimeSpan.FromMinutes(3));
```

Задаем равный 5 лимит неудач и равное 3 минутам время пребывания в разомкнутом состоянии

Хотя шаблон «Предохранитель» может показаться более сложным, чем шаблон «Повтор», библиотека Polly делает пастройку стратегии предохранителя ничуть

не более сложной, чем пастройка стратегии повторов. Использование стратегии не различается для разных стратегий, так что код для обертывания запросов к микросервису Loyalty Program в стратегию предохранителя не будет ничем отличаться от кода для обертывания команд регистрации пользователей в стратегию повторов.

Листинг 6.4. Обертывание запроса в предохранитель

```
private static Policy circuitBreaker =  
    Policy  
        .Handle<Exception>()  
        .CircuitBreaker(5), TimeSpan.FromMinutes(3));
```

Задаем равный 5
лимит неудач
и равное 3 минутам
время пребывания
в разомкнутом состоянии

С помощью этих двух стратегий микросервис API Gateway отвечает за устойчивость к ошибкам взаимодействия с микросервисом Loyalty Program. Далее перейдем к рассмотрению микросервиса Loyalty Program.

Реализация стратегии редких повторов с помощью библиотеки Polly

Микросервис Loyalty Program подписывается на события микросервиса Special Offers. Исходя из этих событий, Loyalty Program отправляет микросервису Notifications команды на оповещение пользователей о новых специальных предложениях. Если отправка команды микросервису Notifications завершается неудачно, желательно ее повторить. Поскольку отправка уведомлений — процесс не слишком критический по времени, повторять немедленно не обязательно. Вместо этого можно повторить отставку тогда, когда подписчик все равно будет выполнять опрос на предмет появления новых событий. Все, что нужно сделать для этого, — отследить последнее успешно обработанное событие.

Как вы помните из главы 4, подписчик периодически просыпается и опрашивает ленту на предмет появления новых событий. В каждом таком цикле происходят чтение и обработка очередного пакета событий. Данный пакет начинается с события, следующего за последним успешно обработанным. Это означает повторную обработку всех сбойных событий. Это также значит, что при неудаче обработки одного события вполне можно прекратить обработку оставшейся части пакета — все равно будет предпринята попытка повтора всех остальных позднее. Метод из главы 4, предназначенный для выполнения каждого из циклов, выглядит следующим образом (листинг 6.5).

Листинг 6.5. Цикл подписки для отдельного события

```
private async Task SubscriptionCycleCallback()  
{  
    var response = await ReadEvents().ConfigureAwait(false);  
    if (response.StatusCode == HttpStatusCode.OK)  
        HandleEvents(  
            await response.Content.ReadAsStringAsync().ConfigureAwait(false));  
    this.timer.Start();  
}
```

Этот метод полагается на другие методы, выполняющие чтение и обработку событий. Чтение событий в главе 4 происходило следующим образом (листинг 6.6).

Листинг 6.6. Цикл подписки для отдельного события

```
public async Task<HttpResponseBody> QueryUser(int userId)
{
    return await circuitBreaker
        .ExecuteAsync(() => DoUserQuery(userId));
}

private async Task<HttpResponseBody> DoUserQuery(int userId)
{
    var userResource = $"/users/{userId}";
    using (var httpClient = new HttpClient())
    {
        httpClient.BaseAddress = new Uri($"http://{this.hostName}");
        var response = await httpClient.GetAsync(userResource);
        ThrowOnTransientFailure(response);
        return response;
    }
}
```

Используем стратегию предохранителя

Извещаем предохранитель о неудаче путем генерации исключения

И наконец, обработка событий выглядит вот так (листинг 6.7).

Листинг 6.7. Цикл подписки для отдельного события

```
private async Task<HttpResponseBody> ReadEvents()
{
    using (var httpClient = new HttpClient())
    {
        httpClient.BaseAddress =
            new Uri($"http://{this.loyaltyProgramHost}");
        var resource =
            $"/events/?start={this.start}&end={this.start + this.chunkSize}";
        var response = await
            httpClient
                .GetAsync(resource)
                .ConfigureAwait(false);
        PrettyPrintResponse(response);
        return response;
    }
}
```

Сохраняет начальную точку пакета в памяти

Для начала реализации стратегии повторов изменим код чтения событий, чтобы использовать пачальный номер, прочитанный из базы данных (листинг 6.8).

Листинг 6.8. Чтение событий, начиная с сохраненного начального номера

```
private async Task<HttpResponseBody> ReadEvents()
{
    var startNumber = await ReadStartNumber().ConfigureAwait(false);
    using (var httpClient = new HttpClient())
    {
        httpClient.BaseAddress = new Uri($"http://{this.loyaltyProgramHost}");
        var resource =
```

Читаем начальный номер из базы данных

```

    $"/events/?start={startNumber}&end={this.start + this.chunkSize}";
    var response = await httpClient.GetAsync(resource).ConfigureAwait(false);
    PrettyPrintResponse(response);
    return response;
}
}

private async Task<long> ReadStartNumber()
{
    // Читаем начальный номер из базы данных

```

Используем
startNumber
при запросе
события из ленты

Далее внесем изменения в обработку событий, чтобы прерывать ее в случае неудачи и записывать номер последнего успешно обработанного события в базу данных (листинг 6.9).

Листинг 6.9. Отслеживаем, какие события уже были обработаны

Прерываем выполнение в случае неудачи обработки события

```

private async Task HandleEvents(string content)
{
    var lastSucceededEvent = 0L;
    var events = JsonConvert.DeserializeObject<IEnumerable<Event>>(content);
    foreach (var ev in events)
    {
        dynamic eventData = ev.Content;
        if (ShouldSendNotification(eventData))
        {
            var notificationSucceeded = await
                SendNotification(eventData).ConfigureAwait(false);
            if (!notificationSucceeded)
            {
                return;
            }
            lastSucceededEvent = ev.SequenceNumber + 1;
        }
        await WriteStartNumber(lastSucceededEvent).ConfigureAwait(false);
    }
}

private bool ShouldSendNotification(dynamic eventData)
{
    // Определяем на основе бизнес-правил,
    // нужно ли отправлять уведомление
}

private Task<bool> SendNotification(dynamic eventData)
{
    // Используем HttpClient для отправки команды микросервису Notifications
    // Возвращает True, если команда завершилась успешно,
    // False – в противном случае
}

private async Task WriteStartNumber()
{
    // Записываем начальный номер в базу данных
}

```

Отслеживаем успешно
обработанные события

Не все события
обязательно
оказываются
обработанными
успешно

Обновляем значение
для последнего успешно
обработанного события

Обновляем номер,
с которого должен
начаться
следующий пакет

Как видите, изменения, необходимые для использования стратегии редких повторов, невелики.

Заносим в журнал все необработанные исключения

Наконец, обратим внимание на компонент HTTP API микросервиса Loyalty Program. Как уже отмечалось, желательно иметь подробный журнал всего, что в системе пошло не так. А значит, необходимо журналировать все необработанные исключения, которые были сгеперированы в обработчиках для конечных точек микросервисов. Для этого можно воспользоваться копвейером ошибок уровня приложения фреймворка Nancy. Копвейер ошибок позволяет добавлять обработчики, вызываемые всякий раз, когда обработчик маршрута микросервиса геперирует необработанное исключение. Мы добавим обработчик в копвейер ошибок загрузчика Nancy, переопределив метод `ApplicationStartup` и обращаясь к копвейеру ошибок через интерфейс `IPipelines`.

Мы уже написали загрузчик Nancy для микросервиса Loyalty Program в главе 4. Теперь расширим его следующим кодом для добавления обработчика в копвейер ошибок (листинг 6.10).

Листинг 6.10. Регистрация глобального обработчика ошибок с помощью Nancy

```
namespace LoyaltyProgram
{
    using Nancy;
    using Nancy.Bootstrapper;
    using Nancy.TinyIoc;

    public class Bootstrapper : DefaultNancyBootstrapper
    {
        ...

        protected override void ApplicationStartup(
            TinyIoCContainer container,
            IPipelines pipelines)
        {
            pipelines.OnError += (ctx, ex) => {
                Log("Unhandled", ex);
                return null;
            };

            private void Log(string message, Exception ex)
            {
                // Отправляем message и ex в централизованное хранилище журналов
                // Мы увидим, как это делается, в главе 9
            }
        }
    }
}
```

Добавляем обработчик в конвейер ошибок

Не переопределяем ответ, возвращаемый по умолчанию

Добавленный в конвейер ошибок обработчик возвращает `null`. Он мог также возвращать объект `Nancy.Response`. В этом случае фреймворк Nancy использовал бы его в качестве ответа клиенту. Возвращая же `null`, как показано здесь, мы решили не переопределять этот ответ. Следовательно, клиент получает ответ по умолчанию с кодом состояния 500.

Интерфейс `IPipelines` обеспечивает доступ также к конвейерам `Before` и `After`. Можно аналогичным образом добавить обработчики в эти конвейеры, и они будут вызываться соответственно до и после вызова каждого из обработчиков маршрутов.

У фреймворка Nancy есть также конвейеры `Before`, `After` и `OnError` в классе `NancyModule`. Они работают аналогично, за исключением того, что используются только для обработчиков в том модуле, в котором были настроены.

На этом реализацию средств обеспечения устойчивости к ошибкам в связных с микросервисом `Loyalty Program` взаимодействиях можно считать завершенной. После принятия этих довольно простых мер взаимодействия, вероятно, будут намного более устойчивыми к ошибкам под реальной нагрузкой при эксплуатации в производственной среде.

6.4. Резюме

- ❑ Можно ожидать проблем с частью взаимодействий между микросервисами из-за их объема. Для надежной работы системы жизненно важно, чтобы микросервисы должным образом обрабатывали подобные отказы.
- ❑ Следует встраивать устойчивость к ошибкам в микросервисы на этапе проектирования так, чтобы отказы не распространялись по системе и не становились со временем ошибками.
- ❑ За обеспечение устойчивости к ошибкам взаимодействия в случае неудачного завершения запросов отвечает клиент.
- ❑ Вам понадобятся подробные журналы с возможностями удобного доступа к ним и поиска в случае необходимости расследования причин проблем при эксплуатации в производственной среде. Все журнальные сообщения должны поступать в централизованный микросервис `Log`, который обеспечивал бы доступ к ним.
- ❑ Важнейшие стратегии обеспечения устойчивости к ошибкам взаимодействий — шаблоны «Повтор» и «Предохранитель».
- ❑ Библиотека `Polly` упрощает настройку и использование стратегий как повторов, так и предохранителя.
- ❑ Во фреймворке Nancy имеются конвейеры ошибок уровня как приложения, так и модуля, обеспечивающие возможность реагировать на необработанные исключения. Как минимум их необходимо записать в централизованный журнал.

7

Написание тестов для микросервисов

В этой главе:

- ❑ написание хороших автоматизированных тестов;
- ❑ пирамида тестов и ее приложение к микросервисам;
- ❑ тестирование микросервисов извне;
- ❑ написание быстрых внутривещисных тестов для конечных точек;
- ❑ использование фреймворка Nancy;
- ❑ комплексные тесты и модульное тестирование.

На текущий момент мы разработали несколько микросервисов и палили взаимодействия между ними. Реализации неплохи, но мы еще не создали для них никаких тестов. По мере написания все большего количества микросервисов дальнейшая разработка системы без хороших автоматизированных тестов становится невозможной. В первой половине этой главы мы обсудим, что необходимо для тестирования отдельного микросервиса. Затем займемся кодом и рассмотрим сначала тестирование конечных точек с помощью библиотеки `Nancy.Testing`, а затем тестирование микросервиса в целом, имитируя отправку ему запросов от другого микросервиса.

7.1. Что и как тестировать

В главе 1 приводились три отличительных признака микросервиса, благодаря которым он хорошо подходит для непрерывной доставки ПО.

- ❑ *Микросервисы можно развертывать по отдельности.* Микросервис можно развернуть в производственной среде сразу же после выполнения любого небольшого безопасного изменения. Но откуда мы знаем, что это изменение безопасно? Именно для этого и нужны тестирование и, в частности, автоматизация тестирования. Свою роль в подтверждении безопасности изменения играют и некоторые другие операции, в том числе экспертиза кода, статический анализ кода и проектирование общедоступных API для обратной совместимости, но в основном именно тестирование придает вам уверенности.

- ❑ *Микросервис можно легко заменить.* Следует стремиться к тому, чтобы можно было заменить реализацию микросервиса другой функционально эквивалентной реализацией при обычном режиме работы. Тесты играют важную роль и в этом процессе, поскольку хороший набор тестов дает возможность проверить, действительно ли новая реализация эквивалентна старой.
- ❑ *Для сопровождения микросервиса достаточно небольшой команды разработчиков.* Микросервисы довольно малы и узко специализированы, так что команда разработчиков может сопровождать несколько микросервисов сразу. Преимущество этого в том, что вы можете писать тесты, охватывающие все части ваших микросервисов.

Чтобы можно было быстро убедиться в безопасности изменений и заменить плохо реализованный микросервис, тестирование должно быть быстрым и воспроизводимым. А чтобы обеспечить быстроту и воспроизводимость тестирования, необходимо автоматизировать значительную его часть — в этом и состоит задача данной главы.

Пирамида тестов: что необходимо тестировать в системе микросервисов

Показывая на рис. 7.1 *пирамида тестов* — инструмент, которым можно руководствоваться при определении того, какие тесты следует писать и сколько именно тестов каждого вида. В различных источниках можно найти разные варианты пирамиды тестов, но во всех них тесты разделяются на уровни, причем на верхнем находятся тесты с самой широкой областью действия, а на нижнем — с более узкой. Пирамида тестов иллюстрирует то, что разработчику следует стремиться иметь большое количество узкоспециализированных тестов (тех, что располагаются в широкой нижней части пирамиды) и только несколько тестов с более широкой областью действия (тех, что располагаются в узкой верхней части пирамиды).



Рис. 7.1. Пирамида тестов иллюстрирует, что необходимо несколько системных тестов, много эксплуатационных тестов и еще больше модульных тестов

Приведенная версия пирамиды тестов включает три уровня.

- ❑ *Системные тесты (верхний уровень).* Тесты, охватывающие всю систему микросервисов, реализуются обычно через GUI.
- ❑ *Эксплуатационные тесты (средний уровень).* Тесты, служащие для проверки одного и только одного микросервиса целиком.
- ❑ *Модульные тесты (нижний уровень).* Тесты, проверяющие один маленький фрагмент функциональности микросервиса. Модульные тесты обращаются к коду тестируемого микросервиса изнутри процесса и обычно затрагивают только часть микросервиса.

Обратите внимание на то, что в термине «*модульный тест*» слово «*модуль*» относится к небольшому фрагменту функциональности. Область действия модульного теста определяется не в терминах конкретных структурных компонентов кода, таких как класс или метод, а скорее в терминах функциональности. Когда в дальнейшем вы увидите реализацию модульных тестов, обратите внимание на то, что они способны охватывать все слои микросервиса, например начиная от модуля Nancy через предметно-ориентированный объект до класса доступа к данным.

Хотя пирамида тестов указывает на то, что по мере движения по уровням вниз нужно все больше тестов, конкретное количество тестов на каждом уровне зависит от обстоятельств. Его определяют такие факторы, как размер системы, ее сложность и издержки, которые придется понести в случае сбоя.

Системные тесты: тестирование системы микросервисов в целом

Область действия тестов, располагающихся вверху пирамиды, очень широка, следовательно, всего несколько тестов охватывают огромное количество кода. В силу столь широкой области действия эти тесты довольно неточны. При неудачном завершении теста сразу непонятно, в чем проблема. Тест может охватывать всю систему, так что проблема может быть где угодно.

Примером системного теста может послужить тест, использующий пользовательский веб-интерфейс обсуждавшейся в предыдущих главах POS-системы для добавления нескольких товаров в счет-фактуру, учета скидочного кода и оплаты с помощью тестовой кредитной карты. Успешное прохождение теста подтверждает, что счета-фактуры созданы, скидки учтены, а также реализована возможность получения платежей с помощью кредитных карт. Во время выполнения этого системного теста можно добавить контроль того, что сумма к оплате по счету-фактуре равна ожидаемой. Если этот оператор контроля не проходит, причина проблемы может заключаться, например, в том, что цена одного или нескольких товаров неверна, или скидка была учтена неправильно, или данные счета-фактуры были приняты не так, как следовало. Другими словами, подобная проблема может быть вызвана как минимум несколькими различными микросервисами. Чтобы найти виновника, необходимо провести соответствующие изыскания.

Источник проблемы, вероятно, станет понятен из того, как именно системный тест потерпел неудачу, по этому источнику может заключать в себе значительное количество кода. Из одного системного теста непонятно даже, какой именно микросервис вызвал сбой. В то же время успешно пройденные системные тесты действительно придают уверенности в работоспособности системы.

Второй недостаток системных тестов — их медлительность. Это тоже оборотная сторона охвата ими всей системы: выполняются настоящие HTTP-запросы, информация записывается в постоянные хранилища данных, опрашиваются постоянные ленты событий.

С учетом того, что системные тесты в случае успешного выполнения дают уверенность в работоспособности системы, но медленно работают и неточны, рекомендую вам *писать их для сценариев успешного выполнения наиболее важных бизнес-процессов*. Это позволяет охватить сценарии успешной работы всех наиболее важных частей системы. При желании можно добавить также тесты для наиболее часто встречающихся и важных сценариев неудачного выполнения. Требуемое для этого конкретное количество системных тестов зависит от ситуации. Этот совет в равной степени подходит для микросервисов, традиционной архитектуры SOA и монолитов. Ничего относящего только к микросервисам в системных тестах нет. Поэтому я не буду демонстрировать здесь реализацию системных тестов.

Эксплуатационные тесты: тестирование микросервиса извне процесса

Тесты на среднем уровне пирамиды тестов взаимодействуют с одним микросервисом как единым целым, причем изолированно, другие взаимодействующие с ним микросервисы во время теста заменяются *имитациями микросервисов — заглушками (mocks)*. Подобно системным тестам, они взаимодействуют с тестируемым микросервисом извне. Но в отличие от системных тестов взаимодействуют непосредственно с общедоступным API микросервиса и используют операторы контроля ответов микросервиса, равно как и взаимодействия микросервиса с другими, например контроля команд, отправляемых тестируемым микросервисом другим микросервисам.

Имитации микросервисов моделируют настоящий микросервис и фиксируют информацию о его взаимодействиях

В эксплуатационных тестах вместо постоянного микросервиса можно использовать имитацию микросервиса. Она реализует те же конечные точки, что и настоящий микросервис, но вместо настоящей бизнес-логики конечных точек в ней используются предельно упрощенные реализации. Обычно конечные точки в имитациях возвращают жестко зашитые ответы. Кроме того, имитации фиксируют информацию о запросах к конечным точкам, чтобы код тестов мог исследовать сделанные во время теста запросы.

Это очень похоже на широко используемые при тестировании объектно-ориентированного кода имитационные объекты. Отличие только в том, что имитационный объект заменяет настоящий объект, а имитация микросервиса заменяет постоянный микросервис.

Аналогично системным тестам эксплуатационные тесты проверяют сценарии, а не отдельные запросы. То есть они выполняют последовательность запросов, составляющих вместе осмысленный сценарий. Выполняемые микросервисом во время теста запросы к имитациям взаимодействующих с ним микросервисов — это настоящие HTTP-запросы, а ответы — настоящие HTTP-ответы.

Например, вернемся к микросервису Loyalty Program из примера POS-системы. В главе 4 вы видели, что он, как показано на рис. 7.2, взаимодействовал с несколькими другими микросервисами, используя все три стиля взаимодействия: события, запросы и команды.



Рис. 7.2. Микросервис Loyalty Program взаимодействует с несколькими другими микросервисами посредством всех трех типов взаимодействия: событий, запросов и команд

Для изолированной проверки микросервиса Loyalty Program можно создать имитационные версии взаимодействующих с ним микросервисов. Как показано на рис. 7.3, взаимодействующий с микросервисом Loyalty Program имитационный микросервис возвращает ему жестко зашитый ответ.

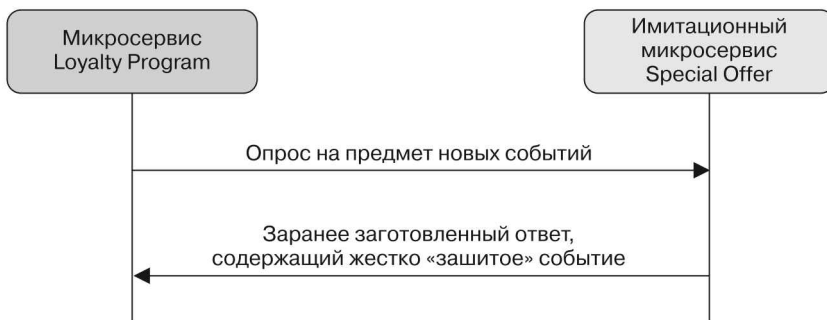


Рис. 7.3. При эксплуатационном тестировании микросервис Loyalty Program взаимодействует с имитационными версиями других микросервисов. Имитационные микросервисы возвращают жестко зашитые ответы на его запросы

Эксплуатационный тест для микросервиса Loyalty Program может выполнять следующие действия:

- ❑ отправлять команду создания пользователя;
- ❑ ждать запроса на предмет появления событий от микросервиса Loyalty Program к имитационному микросервису Special Offer и получать жестко зашитое событие, относящееся к повому специальному предложению;
- ❑ регистрировать все отправляемые микросервису Notifications команды и контролировать, была ли отправлена команда на уведомление нового пользователя о новом специальном предложении.

Если подобный тест выполняется успешно, можно быть уверенными, что все важнейшие компоненты микросервиса Loyalty Program работают. Если же нет, значит, проблему надо искать внутри самого Loyalty Program.

Эксплуатационные тесты намного более точны, чем системные, поскольку они охватывают только отдельный микросервис: если подобный тест завершается неудачно, проблема заключается в самом тестируемом микросервисе, если, конечно, в тест не вкрались ошибки. В силу малого размера микросервисы заменяемы, в конце концов, информация о том, что проблема лежит в конкретном микросервисе, намного более точна по сравнению с результатами системных тестов.

Однако эксплуатационные тесты все еще довольно медленные, поскольку взаимодействуют с тестируемым микросервисом посредством протокола HTTP, а микросервис использует реальную базу данных и взаимодействует с имитационными микросервисами по HTTP.

Контрактные тесты

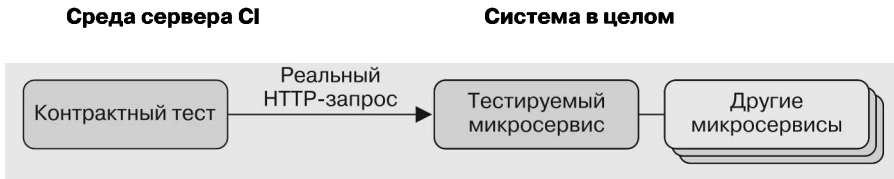
Как вам уже известно, между микросервисами в системе микросервисов происходит множество взаимодействий. Мы реализуем взаимодействия в виде запросов от одного микросервиса к другому. Нужно соблюдать осторожность, чтобы изменения в конечной точке не вызвали сбоев вызывающих ее микросервисов. Тут нам и пригодятся контрактные тесты.

При взаимодействии двух микросервисов в системе микросервис, выполняющий запрос, ожидает от второго определенного поведения. Говорят, что при взаимодействии вызывающий микросервис ожидает, что вызываемый микросервис реализует определенный контракт. *Контрактный тест* — это тест, цель которого заключается в выяснении того, реализует ли вызываемый микросервис контракт, выполнения которого ожидает вызывающий.

Контрактные тесты пишутся с точки зрения вызывающей стороны и создаются ради нее: если контрактный тест выполняется успешно, допущения вызывающей стороны относительно контракта справедливы. Следовательно, контрактные тесты — часть базы кода вызывающего микросервиса. Они не относятся к той же базе кода, что и тестируемые ими конечные точки. Контрактным тестам не требуется никакой информации о способе реализации проверяемых с их помощью микросервисов. Этим контрактные тесты отличаются от эксплуатационных. В случае выполнения эксплуатационных тестов тестируемый микросервис изолируется с помощью замены взаимодействующих с ним микросервисов на имитации. Контрактным тестам это не нужно, ведь они не должны

знать ничего о микросервисах, взаимодействующих с тестируемым ими. Другими словами, контрактные тесты имеют дело с системой в целом.

Поскольку контрактные тесты — часть базы кода одного микросервиса, но тестируют части других микросервисов и поскольку они имеют дело с системой в целом, может оказаться хорошей идеей использовать их для QA или предэксплуатационного тестирования. Более того, неплохой идеей будет автоматическое выполнение их при каждом развертывании тестируемого микросервиса. Нарушение контракта является серьезным сигналом о сбое взаимодействия между микросервисом, к которому относится контрактный тест, и тестируемым микросервисом.



Контрактный тест имеет дело с системой в целом. Его можно использовать, например, в средах QA или предэксплуатационного тестирования, в которых развернута система целиком

С точки зрения реализации контрактный тест немного напоминает эксплуатационные тесты, которые мы будем писать далее в этой главе. Различие заключается в том, что контрактные тесты располагаются несколько выше в пирамиде тестов — между системными и эксплуатационными тестами. Для контрактных тестов, в отличие от эксплуатационных, не нужны имитационные микросервисы. Но как и при выполнении эксплуатационных тестов, в ходе них выполняются постоянные HTTP-запросы к тестируемому микросервису.

Что касается эксплуатационных тестов, рекомендую вам писать их для всех сценариев успешного использования предоставляемой тестируемым микросервисом функциональности. Подобные тесты естественным образом задействуют все конечные точки микросервиса, равно как и подписки на события. Другими словами, они охватывают все пути успешного выполнения в микросервисе. В целом я бы рекомендовал создавать эксплуатационные тесты только для самых важных сценариев неудачного выполнения. К тому же необходимое число эксплуатационных тестов и количество охватываемых ими сценариев неудачного выполнения зависит от системы и того, насколько дорого обходится сбой в конкретной системе.

Модульные тесты: тестирование конечных точек изнутри процесса

Располагающиеся внизу пирамиды тесты также работают с отдельным микросервисом, но они не работают через протокол HTTP и не имеют дела с микросервисом целиком. Эти модульные тесты взаимодействуют с частями тестируемого микросервиса непосредственно и в оперативной памяти. Для обращения к реализованным в модулях Nancy конечным точкам воспользуемся одной из вспомогательных

библиотек фреймворка Nancy — Nancy.Testing. С помощью Nancy.Testing можно создавать тесты, обращающиеся к конечным точкам Nancy в оперативной памяти. Эти обращения выполняются Nancy аналогично HTTP-запросам, но без прохождения через сетевой стек. Реализуемые с помощью библиотеки Nancy.Testing вызовы выглядят для кода в модулях Nancy точно так же, как постоянные HTTP-запросы.

Я продемонстрирую два вида модульных тестов (рис. 7.4): использующие базу данных и использующие имитацию базы данных. Я рассматриваю обе разновидности как модульные тесты, несмотря на то что первая из них применяет базу данных. Модульным тест делают две вещи: во-первых, его область действия — это небольшой фрагмент функциональности, во-вторых, тестовый код и код, предназначенный для эксплуатации в производственной среде, работают в одном процессе.

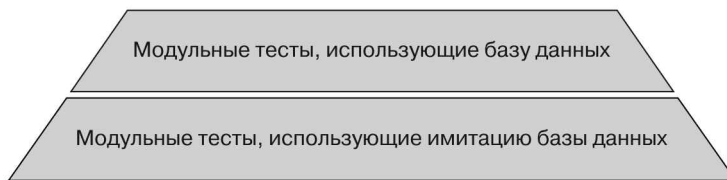


Рис. 7.4. Существуют две разновидности модульных тестов: использующие базу данных и не использующие

Узкая область действия модульных тестов обеспечивает их точность: в случае сбоя, понятно, что проблема заключается в небольшом фрагменте кода. Узкая область действия также позволяет писать тесты, которые должным образом охватывают сценарии неудачного выполнения. Оба типа модульных тестов работают быстрее, чем эксплуатационные тесты, но, конечно, имитирующие базы данных тесты быстрее использующих реальную базу данных. Следовательно, у вас могут быть и те и другие, причем, вероятно, имитирующих базу данных тестов будет больше, чем не имитирующих.

Иногда возможно использование модульных тестов с еще более узкой областью действия, тестирующих бизнес-логику микросервисов путем непосредственного создания экземпляров предметно-ориентированных объектов и тестирования их напрямую. Я обычно занимаю прагматическую позицию при определении того, насколько узкой должна быть область действия наиболее специализированных модульных тестов: начинаю процесс разработки извне, с тестов, которые с помощью библиотеки Nancy.Testing обращаются к обработчикам конечных точек в модулях Nancy. Я начинаю с тестов, охватывающих функциональность конечной точки в общем, и постепенно добавляю тесты для частных элементов. И лишь тогда, когда тестировать конкретный элемент через обработчик конечной точки становится неудобно, начинаю писать более узкоспециализированные модульные тесты. Например, чтобы охватить отдельный кейс бизнес-логики обращающимися через обработчик конечной точки тестами, может потребоваться значительное количество подготовительного кода. Это знак того, что следует перейти к тесту с более узкой областью действия, включающей в себя только соответствующие кейсы бизнес-логики. Я пишу тесты для кейсов, непосредственно работающих с классами, которые должны реализовывать конкретную часть бизнес-логики.

Для микросервиса Loyalty Program нам понадобятся модульные тесты для тестирования конечной точки создания пользователей с набором различных входных данных, которые бы охватывали как допустимые, так и недопустимые входные данные. Понадобятся и модульные тесты, которые пытались бы прочитать из соответствующей конечной точки как существующих, так и несуществующих пользователей. Нужны также аналогичные тесты для других конечных точек микросервиса. Микросервис Loyalty Program довольно прост, так что тесты, чья область действия была бы уже, чем конечные точки микросервиса, нам не понадобятся. Итак, все последующие модульные тесты работают, обращаясь к обработчикам конечных точек посредством библиотеки `Nancy.Testing`.

7.2. Тестировочные библиотеки `Nancy.Testing` и `xUnit`

В этой главе мы воспользуемся двумя новыми библиотеками:

- ❑ `Nancy.Testing` (<https://github.com/NancyFx/Nancy/wiki/Testing-your-application>);
- ❑ `xUnit` (<https://xunit.github.io/>).

Я кратко познакомлю вас с обеими, после чего мы реализуем тесты для некоторых из созданных в предыдущих главах микросервисов.

Познакомьтесь: библиотека `Nancy.Testing`

Библиотека `Nancy.Testing` — вспомогательная библиотека фреймворка `Nancy`, упрощающая тестирование реализованных в модулях `Nancy` конечных точек. Основная точка входа `Nancy.Testing` — тип `Browser`, в котором возможны такие вызовы методов, как `Get("/")`, `Post("/user")`, `Put("/user/42")` и `Delete("/user/42")`, что позволяет тестам вызывать конечные точки `GET`, `POST`, `PUT` и `DELETE` соответственно. При обращении теста к конечной точке через тип `Browser` вызов проходит через настоящий копвейер `Nancy`. Это значит, что маршруты разрешаются точно так же, как и для реальных HTTP-запросов, создается и используется стандартным образом коптейпер внедрения зависимости, выполняются, как обычно, сериализация и десериализация. Короче говоря, для конечной точки это обращение выглядит точно так же, как и настоящий HTTP-запрос. Самое замечательное то, что все это происходит внутри процесса, а значит, намного быстрее, чем выполнялся бы настоящий HTTP-запрос. Возвращаемое значение всех методов — объект типа `NancyResponse`, содержащий все то же, что содержал бы настоящий HTTP-запрос, включая заголовки, коды состояния и тело.

Помимо типа `Browser`, в библиотеке `Nancy.Testing` имеется класс `ConfigurableBootstrapper` — замечательный API для создания используемых в тестах загрузчиков для конкретных случаев. Кроме прочего, `ConfigurableBootstrapper` дает возможность:

- ❑ создавать объекты типа `Browser`, которым был бы виден только один модуль `Nancy`, а не все модули приложения;

- ❑ перекрывать регистрации объектов в контейнере внедрения зависимостей, например подставлять имитационные объекты вместо настоящих;
- ❑ добавлять в конвейер Nancy точки подключения, например обработчики ошибок.

Наконец, библиотека `Nancy.Testing` поставляется с множеством удобных методов, облегчающих написание операторов контроля для объектов `NancyResponse`.

Библиотека `Nancy.Testing` включает массу функциональности, облегчающей написание тестов. Обсуждение всего этого выходит за рамки данной главы, но часть ее возможностей вы увидите. API этой библиотеки представляются мне весьма информативными, так что как только вы начнете с ними работать, то сразу обнаружите многое из того, что может библиотека `Nancy.Testing`.

Дополнительную информацию о библиотеке `Nancy.Testing` можно найти в ее документации (<https://github.com/NancyFx/Nancy/wiki/Testing-your-application>), но можно и просто начать ее использовать. Думаю, что с помощью `IntelliSense` можно хорошо разобраться с ее API.

Познакомьтесь: фреймворк xUnit

Фреймворк `xUnit` (<http://xunit.github.io>) — это инструмент модульного тестирования для платформы .NET. Он состоит из библиотечной части, позволяющей создавать автоматизированные тесты, и выполняющей части, с помощью которой можно эти тесты выполнять. Для написания тестов с помощью `xUnit` необходимо создать метод с указанием перед ним атрибута `Fact` и поместить в него выполняющий тест код. Выполняющая часть `xUnit` ищет методы с атрибутом `Fact` и выполняет их. Кроме того, у `xUnit` имеется API для вставки в тесты операторов контроля. Если контроль оказывается неудачным, выполняющая часть `xUnit` запоминает отказ и сообщает о нем после окончания выполнения тестов. Выполняющую тесты часть `xUnit` можно запустить посредством утилиты `dotnet`, а значит, она хорошо подходит для создаваемых в этой книге проектов.

Существуют и другие доступные для использования тестовые утилиты для платформы .NET, например пакет `NUnit`. В этой книге мы будем использовать `xUnit`, поскольку она применяется для тестовых проектов, создаваемых с помощью `Yeoman` и `Visual Studio`. Если вам больше нравится другая утилита, то вы можете использовать ее, главное, чтобы она работала с `dotnet`.

Совместная работа xUnit и Nancy.Testing

С помощью библиотеки `Nancy.Testing` вместе с фреймворком `xUnit` можно писать весьма лаконичные тесты для реализованных в модулях Nancy конечных точек. В разделе «Создание проекта для модульного тестирования» мы создадим проект для этих модульных тестов и выполним их с помощью утилиты `dotnet`, а пока я хочу лишь показать вам, как эти тесты выглядят. Следующий тест обращается к конечной точке типа `GET` в классе `TestModule` и контролирует то, что код состояния ответа должен быть `200 OK` (листинг 7.1).

Листинг 7.1. Создание простого теста с помощью Nancy.Testing и xUnit

```

namespace LoyaltyProgramUnitTests
{
    using Nancy;
    using Nancy.Testing;
    using Xunit;

    public class TestModule_should
    {
        public class TestModule : NancyModule
        {
            public TestModule()
            {
                Get("/", _ => 200;) ← Используемая в тесте конечная точка
            }

            [Fact]
            public async Task respond_ok_to_request_to_root()
            {
                var sut = new Browser(with => with.Module<TestModule>()); ←
                var actual = await sut.Get("/");
                Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
            }
        }
    }
}

```

Указание для загрузчика Nancy модуля TestModule

Обращение к конечной точке типа Get из модуля TestModule

Контролируем возвращение конечной точкой ответа 200 OK

Соглашения о наименованиях

В моих тестах применяются следующие соглашения о наименованиях.

- Тесты работают с объектом, носящим название sut — сокращение от *system under test* (*тестируемая система*). В предыдущем тесте sut представлял собой объект типа `Browser`, которым мы воспользовались для обращения к конечной точке.
- Названия тестовых классов соответствуют тому элементу, который они тестируют (в данном примере это `TestModule`), за которым следует `_should`.
- Метод `Fact` был назван по тестируемому сценарию и результату, который мы ожидаем получить. Я буду разделять слова в названиях методов `Fact` символами подчеркивания и постараюсь, чтобы они формировали законченное предложение (на английском языке) в сочетании с именем охватывающего класса. Например, в данном тесте при конкатенации имени класса и названия метода `Fact` с заменой символов подчеркивания на пробелы мы получаем `TestModule should respond ok to request to root` (`TestModule` должен возвращать ответ `ok` на запрос к корневому URL).

Нравятся вам эти соглашения или нет, дело вкуса. Мне они нравятся, но для написания хороших тестов они необязательны.

Выполнить предыдущий тест можно с помощью утилиты командной строки `dotnet`, он будет выполняться в оперативной памяти и обеспечит неплохой охват функциональности, поскольку обращение к `sut.Get("/")` выполняет реальный коп-вейер Nancy, включая реализацию конечной точки в классе `TestModule`. Строковый аргумент `"/"` представляет собой относительный URL, к которому выполняется фиктивный запрос. В подразделе «Создание проекта для модульного тестирования» раздела 7.3 мы рассмотрим настройки проекта для подобных модульных тестов, а также их выполнение с помощью утилиты `dotnet`.

В оставшейся части данной главы мы будем работать с кодом и реализовывать модульные и эксплуатационные тесты для микросервиса `Loyalty Program`. При реализации микросервиса `Loyalty Program` в главе 4 у него не было лепты событий, но для данных примеров придется добавить лепту событий, на которую могли бы подписываться другие микросервисы.

7.3. Написание модульных тестов с помощью библиотеки `Nancy.Testing`

В этом разделе реализуем несколько модульных тестов для конечных точек микросервиса `Loyalty Program`. В главе 4 мы видели, что у микросервиса `Loyalty Program` имеется три конечные точки для команд и запросов:

- ❑ конечная точка HTTP типа `GET`, располагающаяся по URL вида `/users/{userId}` и возвращающая представление пользователя;
- ❑ конечная точка HTTP типа `POST`, располагающаяся по URL вида `/users/`, ожидающая получения в теле запроса представления пользователя и регистрирующая этого пользователя в программе лояльности;
- ❑ конечная точка HTTP типа `PUT`, располагающаяся по URL вида `/users/{userId}`, ожидающая получения в теле запроса представления пользователя и обновляющая данные этого уже зарегистрированного пользователя.

Напишем тесты для этих конечных точек. У микросервиса `Loyalty Program` имеется лепта событий, для которой мы также создадим тест. Мы не станем писать всеобъемлющие тесты для конечных точек и лепты событий микросервиса `Loyalty Program` — лишь такие, которых будет достаточно, чтобы понять, как писать тесты для конечных точек `Nancy`.

В следующих подразделах мы сделаем вот что.

- ❑ Подготовим тестовый проект для модульных тестов для микросервиса `Loyalty Program`.
- ❑ Напишем тесты, которые используют класс `Browser` библиотеки `Nancy.Testing` для тестирования конечных точек микросервиса `Loyalty Program`, причем код микросервиса сможет использовать реальную базу данных. Мы напишем три таких теста, по одному для каждого элемента функциональности:
 - тест, который пытается прочитать данные несуществующего пользователя;
 - тест, который создает пользователя, после чего читает его данные;
 - тест, который модифицирует пользователя, после чего читает его данные.

- ❑ Напишем тесты, также использующие класс **Browser** для тестирования конечной точки, по области действия которых ограничена имитационной базой данных, внедренной в тестируемую конечную точку. Эти тесты будут проверять цепку событий микросервиса.

После выполнения всего этого вы будете знать, как писать модульные тесты для конечных точек Nancy как с реальной базой данных, так и без нее.

Создание проекта для модульного тестирования

Прежде чем мы начнем писать тесты, нам понадобится проект, в котором они будут располагаться. Для этого создадим новый проект рядом с проектом **LoyaltyProgram** и назовем его **LoyaltyProgramUnitTests**. Если вы создаете проект с помощью Visual Studio, выберите шаблон **Class Library (.NET Core)** в диалоговом окне, а если используете Yeoman, выберите в меню пункт **Unit Test Project (xUnit.net)**.

Ваше решение должно выглядеть примерно так:

```
C:\.
├──LoyaltyProgram
│   ├──Bootstrapper.cs
│   ├──project.json
│   ├──README.md
│   ├──Startup.cs
│   ├──UsersModule.cs
│   └──YamlSerializerDeserializer.cs
├──LoyaltyProgram
│   └──EventFeed
│       ├──Event.cs
│       ├──EventsFeedModule.cs
│       ├──EventStore.cs
│       └──IEventStore.cs
├──LoyaltyProgramEventConsumer
│   ├──Program.cs
│   └──project.json
└──LoyaltyProgramUnitTests
    ├──project.json
    └──Class1.cs
```

Если вы воспользовались Yeoman для создания нового проекта **LoyaltyProgramUnitTests**, то можете запускать свои первые тесты. Но если использовали шаблон Visual Studio, придется немного исправить классы **Class1.cs** и **project.json**. В листинге 7.2 показано, как должен выглядеть файл **Class1.cs**.

Листинг 7.2. Файл Class1.cs

```
using Xunit;

namespace UnitTest
{
    // См. пояснения к примеру на веб-сайте xUnit.net:
```

```
// https://xunit.github.io/docs/getting-started-dotnet-core.html
public class Class1
{
    [Fact]
    public void PassingTest()
    {
        Assert.Equal(4, Add(2, 2));
    }

    [Fact]
    public void FailingTest()
    {
        Assert.Equal(5, Add(2, 2));
    }

    int Add(int x, int y)
    {
        return x + y;
    }
}
```

Добавьте в файл `project.json` для настройки тестовой комапды, ссылающейся на выполняющий тесты механизм `xunit`, следующее:

```
"testRunner": "xunit",
```

Мы добавили в проект выполняющий тесты механизм `xunit` с помощью NuGet-пакета `dotnet-test-xunit` и установили пакет `xUnit`. Вот все зависимости:

```
"dependencies": {
  "dotnet-test-xunit": "2.2.0-preview2-build1029",
  "Microsoft.NETCore.App": {
    "version": "1.0.0",
    "type": "platform"
  },
  "xunit": "2.1.0"
},
```

Теперь перейдите в каталог `LoyaltyProgramUnitTests` в командной оболочке PowerShell и восстановите обычным способом пакеты NuGet с помощью утилиты `dotnet`:

```
PS> dotnet restore
```

Файл `Class1.cs` теперь содержит два маленьких теста `xUnit`: завершающийся успешно и неудачно. Выполните их с помощью утилиты `dotnet` следующим образом:

```
PS> dotnet test
```

После их запуска добавьте зависимость на `Nancy.Testing`, чтобы можно было использовать объект `Browser`, а в дальнейшем и `ConfigurableBootstrapper`. Добавьте также зависимость на проект `LoyaltyProgram`, чтобы можно было приступить к его тестированию. Теперь зависимости будут выглядеть следующим образом:

```
"dependencies": {
  "dotnet-test-xunit": "2.2.0-preview2-build1029",
  "Microsoft.NETCore.App": {
```

```

    "version": "1.0.0",
    "type": "platform"
  },
  "xunit": "2.1.0",
  "Nancy.Testing": "2.0.0--barneyrubble",
  "LoyaltyProgram": {"target": "project"} ← Ссылка на проект
},

```

Последняя строка представляет собой ссылку на проект `LoyaltyProgram`. Как видите, ссылки на проекты в файле `project.json` очень похожи на ссылки NuGet. Указывать версию проекта `LoyaltyProgram` не требуется, так как мы собираемся выполнять тест для той версии кода проекта `LoyaltyProgram`, которая лежит возле проекта `LoyaltyProgramUnitTests`.

Использование объекта `Browser` для модульного тестирования конечных точек

Теперь, закопчив с пастройкой тестового проекта, можно приступить к добавлению в него тестов. Первый из них очень прост: допустим, что в микросервисе `LoyaltyProgram` нет зарегистрированных пользователей, при этом тест выполняет запрос пользователя и ожидает получить в ответ код состояния `404 Not Found` (404 Не найдено). Добавьте в проект `LoyaltyProgramUnitTests` файл `userModule_should.cs` и вставьте в него следующий код (листинг 7.3).

Листинг 7.3. Первый тест для конечной точки `users`

```

namespace LoyaltyProgramUnitTests
{
    using LoyaltyProgram;
    using Nancy;
    using Nancy.Testing;
    using Xunit;

    public class UserModule_should
    {
        private Browser sut;

        public UserModule_should()
        {
            this.sut = new Browser(
                new Bootstrapper(),
                defaultsTo => defaultsTo.Accept("application/json"));

            [Fact]
            public void respond_not_found_when_queried_for_unregistered_user()
            {
                var actual = await sut.Get("/users/1000");
                Assert.Equal(HttpStatusCode.NotFound, actual.StatusCode);
            }
        }
    }
}

```

Как вы помните, `sut` — сокращение от `system under test` («тестируемая система»)

Настоящий загрузчик для `LoyaltyProgram`

Все запросы принимают JSON

Запрос несуществующего пользователя

Самая интересная часть этого тестового класса — конструктор, в котором мы создаем объект **Browser**. При запуске **xUnit** создается экземпляр класса **UserModule_should** с последующим вызовом метода этого экземпляра с атрибутом **Fact**. В отличие от большинства других тестовых фреймворков для платформы .NET, **xUnit** создает новый, чистый экземпляр для каждого метода с атрибутом **Fact**.

Объект **Browser** в листинге 7.3 инициализируется с помощью настоящего загрузчика из приложения **LoyaltyProgram**. Это значит, что вызываемое объектом **Browser** приложение **LoyaltyProgram** подключается точно так же, как и при работе поверх настоящего веб-сервера с получением настоящих HTTP-запросов. Более того, для удобства мы задали для **Browser** заголовок **Accept** по умолчанию. Этот заголовок будет добавляться ко всем выполняемым через объект **Browser** запросам, если только такое поведение не будет явным образом переопределено. Например, у команды `sut.Get("/users/1000")` заголовок **Accept** задан.

Обратимся теперь к тесту, который регистрирует нового пользователя, после чего выполняет запрос для проверки правильности регистрации. Добавьте приведенный далее тест в класс **UserModule_should** (листинг 7.4).

Листинг 7.4. Тестирование регистрации пользователя посредством конечной точки `users`

Читаем данные нового пользователя из тела полученного от конечной точки **POST** ответа

```
[Fact]
public void allow_to_register_new_user()
{
    var expected =
        new LoyaltyProgramUser() { Name = "Chr" };
    var registrationResponse = await
        sut.Post("/users", with => with.JsonBody(expected));
    var newUser =
        registrationResponse.Body.DeserializeJson<LoyaltyProgramUser>();
```

Регистрируем
нового пользователя
через конечную
точку типа **POST**

→ var actual = await sut.Get(\$" /users/{newUser.Id}");

```
Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
Assert.Equal(
    expected.Name,
    actual.Body.DeserializeJson<LoyaltyProgramUser>().Name);
// Другие операторы контроля полученного от GET ответа
}
```

Проверяем, что полученный
от конечной точки **GET** ответ корректен

Читаем данные нового пользователя
через конечную точку **GET**

Тут мы видим еще один вариант использования объекта **Browser**. Например, мы добавляем тело в обращение к **POST** с помощью лямбда-выражения во втором аргументе. В этом лямбда-выражении можно делать с запросом множество вещей, например добавлять заголовки, cookie-файлы, значения форм, имя хоста или идентификационные данные или выбирать между **HTTP** и **HTTPS**. В данном случае добавили в запрос тело.

Последний из тестов, который мы собираемся добавить, регистрирует пользователя, после чего проверяет его данные через конечную точку **PUT** микросервиса **LoyaltyProgram** (листинг 7.5). Добавьте его в файл **UserModule_should.cs**.

Листинг 7.5. Тестирование изменения данных пользователя посредством конечной точки `users`

```
[Fact]
public void allow_modifying_users()
{
    var expected = "jane";
    var user = new LoyaltyProgramUser() { Name = "Chr" };
    var registrationResponse = await
        sut.Post("/users", with => with.JsonBody(user));
    var newUser =
        registrationResponse.Body.DeserializeJson<LoyaltyProgramUser>();

    newUser.Name = expected;
    var actual = await
        sut.Put($" /users/{newUser.Id}", with => with.JsonBody(newUser));

    Assert.Equal(
        expected,
        actual.Body.DeserializeJson<LoyaltyProgramUser>().Name);
}
```

Регистрируем пользователя

Обновляем данные пользователя

Проверяем, что обновление было выполнено

Ничего нового в этом коде по сравнению с уже виденным вами в двух предыдущих тестах нет. Но я решил включить его сюда как хорошую иллюстрацию того, какие модульные тесты следует, по моему мнению, писать для конечных точек в микросервисах: модульные тесты, уделяющие основное внимание логике работы конечных точек, а не тестированию лишь одной конечной точки в отрыве от остальных.

Использование настраиваемого загрузчика для внедрения имитаций в конечные точки

После тестирования конечных точек в `UserModule` обратим внимание на ленту событий микросервиса `LoyaltyProgram`. Лента событий представляет собой модуль `Nancy`, зависящий от интерфейса `IEventStore`, через который выполняются сохранение и чтение событий. Далее приведем интерфейс `IEventStore` (листинг 7.6).

Листинг 7.6. Интерфейс `IEventStore`

```
using System.Collections.Generic;

namespace LoyaltyProgram.EventFeed
{
    public interface IEventStore
    {
        IEnumerable<Event> GetEvents(
            long firstEventSequenceNumber,
            long lastEventSequenceNumber);
        void Raise(string eventName, object content);
    }
}
```

Читаем события из хранилища

Сохраняем события в хранилище

Вы уже видели ленту событий в главе 4, но здесь я повторяю ее код, чтобы напомнить, как она работает (листинг 7.7).

Листинг 7.7. Лента событий

```

namespace LoyaltyProgram.EventFeed
{
    using Nancy;

    public class EventsFeedModule : NancyModule
    {
        public EventsFeedModule(IEventStore eventStore) : base("/events")
        {
            Get("/", _ =>
            {
                long firstEventSequenceNumber, lastEventSequenceNumber;
                if (!long.TryParse(this.Request.Query.start.Value,
                    out firstEventSequenceNumber))
                {
                    firstEventSequenceNumber = 0;
                }
                if (!long.TryParse(this.Request.Query.end.Value,
                    out lastEventSequenceNumber))
                {
                    lastEventSequenceNumber = 50;
                }

                return
                    eventStore.GetEvents(
                        firstEventSequenceNumber,
                        lastEventSequenceNumber);
            });
        }
    }
}

```

Получаем начальное значение из строки запроса

Получаем конечное значение из строки запроса

Читаем из хранилища события с начального по конечное

Как видите, лента событий представляет собой модуль Nancy, отвечающий па запросы к URL events-событиями, прочитанными им из `IEventStore`. Нам пужно написать тест, который проверял бы, возвращает ли лента событий в точности событие `IEventStore`. Для этой цели следует проверять, какие события возвращает `IEventStore`. Итак, создадим фиктивную реализацию `IEventStore` и воспользуемся ею в тесте (листинг 7.8).

Листинг 7.8. Фиктивный `IEventStore`, предназначенный для использования в тестах

```

public class FakeEventStore : IEventStore
{
    public IEnumerable<Event> GetEvents(
        long firstEventSequenceNumber,
        long lastEventSequenceNumber)
    {
        if (firstEventSequenceNumber > 100)
        {
            return Enumerable.Empty<Event>();
        }
        else
        {
            return
                Enumerable
                    .Range((int) firstEventSequenceNumber,
                        (int) (lastEventSequenceNumber - firstEventSequenceNumber))
                    .Select(i =>
                        new Event(
                            i,

```

Если `firstEventSequenceNumber` не превышает 100, возвращаем список фиктивных событий

```

        DateTimeOffset.Now,
        "some event",
        new Object()));
    }

    public void Raise(string eventName, object content) {}
}

```

Используя фиктивную реализацию хранилища событий, мы знаем, что хранилище событий будет возвращать список событий только в том случае, если аргумент `firstEventSequenceNumber` не превышает 100. В противном случае `FakeEventStore` вернет пустой список событий. Если внедрить эту фиктивную реализацию в `EventsFeedModule`, мы будем точно знать, какие события `EventsFeedModule` получит от хранилища событий, а значит, какие события он должен вернуть.

Можно воспользоваться еще одной возможностью `Nancy.Testing` для внедрения фиктивного `IEventStore` в `EventsFeedModule`, а именно классом `ConfigurableBootstrapper`, который дает возможность менять конфигурацию тестируемого приложения `Nancy`. В данном случае используем класс `ConfigurableBootstrapper` для задания `FakeEventStore` в качестве реализации `IEventStore` при создании объекта `Browser`. Для этой цели служит следующий фрагмент кода (листинг 7.9).

Листинг 7.9. Использование фиктивного `IEventStore` при тестировании

```

this.sut = new Browser(
    with => with
        .Module<EventsFeedModule>()
        .Dependency<IEventStore>(typeof(FakeEventStore)),
    withDefault => withDefault.Accept("application/json"));

```

С помощью этого кода в экземпляры конструкторов класса `EventsFeedModule` в тестах будет внедряться `FakeEventStore`. Воспользуемся этим (листинг 7.10) для написания:

- ❑ теста, контролирующего, что лента возвращает события, если начальный номер в запросе не превышает 100;
- ❑ теста, контролирующего, что если начальный номер в запросе превышает 100, то никакие события возвращены не будут.

Листинг 7.10. Реализация тестов для ленты событий с использованием фиктивного хранилища событий

```

using System;
using System.Collections.Generic;
using System.Linq;
using LoyaltyProgram.EventFeed;
using Nancy;

```

```

using Nancy.Testing;
using Xunit;

public class EventFeed_should
{
    private Browser sut;

    public EventFeed_should()
    {
        this.sut = new Browser(
            with => with
                .Module<EventsFeedModule>()
                .Dependency<IEventStore>(typeof(FakeEventStore)),
            withDefault => withDefault.Accept("application/json"));
    }

    [Fact]
    public void return_events_when_from_event_store()
    {
        var actual = await sut.Get("/events/", with =>
        {
            with.Query("start", "0");
            with.Query("end", "100");
        });
        Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
        Assert.StartsWith("application/json", actual.ContentType);
        Assert.Equal(100,
            actual.Body.DeserializeJson<IEnumerable<Event>>().Count());
    }

    [Fact]
    public void return_empty_response_when_there_are_no_more_events()
    {
        var actual = wait sut.Get("/events/", with =>
        {
            with.Query("start", "200");
            with.Query("end", "300");
        });

        Assert.Empty(actual.Body.DeserializeJson<IEnumerable<Event>>());
    }
}

```

Создаем Browser, сконфигурированный так, чтобы использовать FakeEventStore

Выполняет запрос к URL /events со строкой запроса "start=0&end=100"

Выполняет запрос к URL /events со строкой запроса "start=200&end=300"

После подготовки пескольных модульных тестов можно выполнить их с помощью утилиты dotnet, как было показано ранее. При этом xUnit выполнит поиск классов, в которых есть методы с атрибутом Fact, после чего выполнит все методы с атрибутом Fact. Результаты тестов отражают информацию о том, сколько было запущено тестов, сколько ошибок выявлено, сколько тестов завершилось неудачей и сколько тестов было пропущено:

```

PS > dotnet test
xUnit.net .NET CLI test runner (64-bit .NET Core win10-x64)
  Discovering: LoyaltyProgramUnitTests
  Discovered:  LoyaltyProgramUnitTests

```

```

Starting: LoyaltyProgramUnitTests
Finished: LoyaltyProgramUnitTests
=== TEST EXECUTION SUMMARY ===
LoyaltyProgramUnitTests Total: 6, Errors: 0, Failed: 0, Skipped: 0, Time:
2.375s
SUMMARY: Total: 1 targets, Passed: 1, Failed: 0.

```

Как вы можете видеть, были запущены шесть тестов, причем ни один из них не завершился неудачно. Другими словами, все тесты были пройдены успешно.

Этими тестами для `EventsFeedModule` и `UsersModule` мы положили хорошее начало написанию модульных тестов для конечных точек наших микросервисов. На практике этих тестов недостаточно, я бы написал дополнительные тесты для граничных случаев и сценариев ошибок. Но главное, что вы уже знаете, как писать такие тесты с помощью `Nancy.Testing`.

7.4. Написание эксплуатационных тестов

Перейдем к написанию эксплуатационных тестов для микросервиса `Loyalty Program` в целом. Эксплуатационные тесты взаимодействуют с микросервисом снаружи, обеспечивая для него имитацию взаимодействующих с ним микросервисов.

Микросервис `Loyalty Program` выполняет запросы к двум другим микросервисам: ленте событий микросервиса `Special Offers` и API микросервиса `Notifications`. Эксплуатационные тесты для микросервиса `Loyalty Program` делают следующее.

1. Размещают в том же процессе, что и тест, две конечные точки: первая будет служить имитацией ленты событий специальных предложений, вторая — имитацией конечной точки для уведомлений.
2. Запускают микросервис `Loyalty Program` в отдельном процессе и настраивают его на использование имитационных конечных точек вместо постоянных микросервисов. Это значит, что всякий раз, когда микросервису `Loyalty Program` понадобится обратиться к взаимодействующему с ним микросервису, он будет обращаться к одной из имитационных конечных точек.
3. Выполняют сценарий тестирования микросервиса `Loyalty Program` в виде последовательности HTTP-запросов.
4. Фиксируют информацию обо всех обращениях к имитационным конечным точкам.
5. Контролируют ответы от микросервиса `Loyalty Program` и запросы к имитационным микросервисам.

На рис. 7.5 показана конфигурация среды выполнения эксплуатационных тестов для микросервиса `Loyalty Program`.

Для создания показанной на рис. 7.5 тестовой конфигурации необходимо сделать следующее.

1. Создать тестовый проект для эксплуатационных тестов.
2. Создать имитационные конечные точки для ленты событий специальных предложений и конечной точки уведомлений.

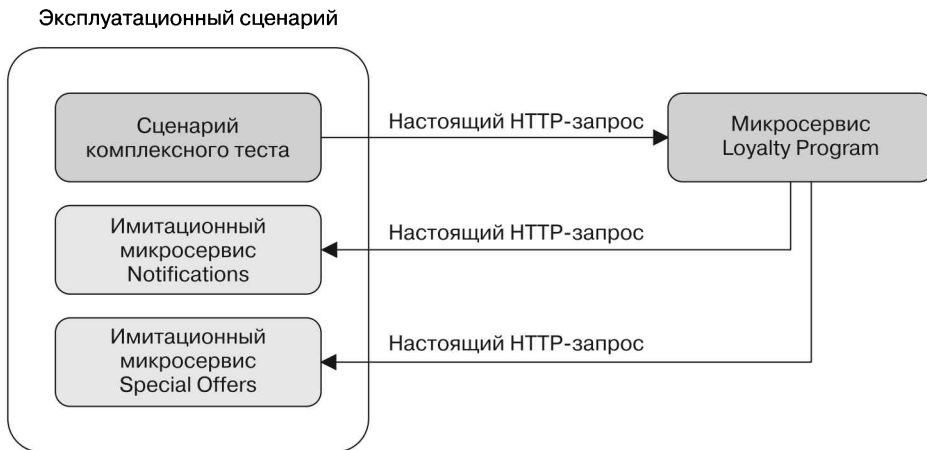


Рис. 7.5. Эксплуатационный тест выполняет тестовый сценарий для API проверяемого микросервиса, но настраивает микросервис таким образом, чтобы использовать имитационные конечные точки, работающие в одном процессе с тестом, вместо реальных микросервисов.

Эксплуатационный тест после запуска отправляет настоящие запросы к тестируемому микросервису, который при необходимости выполняет настоящие HTTP-запросы к имитационным конечным точкам. Тест может проверять как ответы тестируемого микросервиса, так и обращения его к имитационным конечным точкам

3. Запустить оба процесса микросервиса Loyalty Program: содержащее HTTP API приложение Nancy и потребитель событий.
4. Написать исходный код тестов, который выполнял бы сценарий тестирования микросервиса Loyalty Program.

Затем можно написать использующий это все тест.

Создание проекта для эксплуатационного теста

Создадим новый тестовый проект для эксплуатационных тестов, совершенно такой же, как созданный ранее проект для модульных тестов. Так что создайте проект на основе или шаблона ASP.NET Test Project Template в Visual Studio, или Unit Test Project Template в Yeoman и назовите его `LoyaltyProgramIntegrationTest`. Аналогично проекту для модульного теста поместите его рядом с `LoyaltyProgram`. Теперь у нас есть четыре проекта:

Mode	LastWriteTime	Length	Name
-----	-----	-----	-----
d-----	4/6/2016 8:53 PM		LoyaltyProgram
d-----	4/6/2016 8:53 PM		LoyaltyProgramEventConsumer
d-----	4/6/2016 8:53 PM		LoyaltyProgramIntegrationTest
d-----	8/6/2016 10:59 PM		LoyaltyProgramUnitTests

Среди них два проекта, составляющих микросервис Loyalty Program: приложение Nancy и потребитель событий, а также тестовые проекты для этого микросервиса.

Создание имитационных конечных точек

Как показано на рис. 7.5, необходимо создать имитационные версии конечных точек используемых Loyalty Program микросервисов Special Offers и Notifications. Сделаем это, написав два простых модуля Nancy, каждый из которых реализует конечную точку, возвращающую жестко зашитый ответ. В листинге 7.11 показана имитационная конечная точка ленты событий специальных предложений, а в листинге 7.12 — имитационная конечная точка уведомлений.

Листинг 7.11. Имитация ленты событий, возвращающая жестко зашитые ответы

```
public class MockEventFeed : NancyModule
{
    public static AutoResetEvent polled =
        new AutoResetEvent(initialState: false);

    public MockEventFeed()
    {
        this.Get("/events", _ =>
        {
            polled.Set();
            return new [] ← Возвращает жестко зашитый ответ
            {
                new
                {
                    SequenceNumber = 1,
                    Name= "baz",
                    Content = new
                    {
                        OfferName = "foo",
                        Description = "bar",
                        item = new { ProductName = "name" }
                    }
                }
            }
        });
    }
}
```

Оповещает тест о выполнении опросами микросервиса Loyalty Program на предмет появления новых событий

Листинг 7.12. Имитационная конечная точка, регистрирующая обращения к ней

```
public class MockNotifications : NancyModule
{
    public static AutoResetEvent notificationWasSent =
        new AutoResetEvent(initialState: false);

    public MockNotifications()
    {
        this.Get("/notify", _ =>
        {
            notificationWasSent.Set();
            return 200; ← Возвращает жестко зашитый ответ
        });
    }
}
```

Используется в дальнейшем в тесте для выполнения контроля

Мы собираемся запустить эти два модуля в тестовом процессе. Для этого воспользуемся фреймворком Nancy поверх платформы ASP.NET Core, как обычно и делаем. Необходимо добавить пакет NuGet Microsoft.AspNetCore.Owin, а также вставить Nancy и LoyaltyProgram в список зависимостей. Раздел зависимостей в файле project.json теперь выглядит следующим образом (листинг 7.13).

Листинг 7.13. Зависимости из проекта комплексного тестирования, включая Nancy

```
"dependencies": {
  "dotnet-test-xunit": "2.2.0-preview2-build1029",
  "Microsoft.NETCore.App": {
    "version": "1.0.0",
    "type": "platform"
  },
  "xunit": "2.1.0",
  "Microsoft.AspNetCore.Owin": "1.0.0",
  "Nancy": "2.0.0-barneyrubble",
  "LoyaltyProgram": { "target": "project" }
},
```

Далее добавьте файл RegisterUserAndGetNotification.cs, содержащий следующий код (листинг 7.14), использующий пакет Nancy.Hosting.Self для запуска приложения Nancy в тестовом процессе.

Листинг 7.14. Запуск Nancy внутри тестового процесса

```
public class RegisterUserAndGetNotification : IDisposable
{
  private readonly NancyHost hostForMockEndpoints;

  public RegisterUserAndGetNotification()
  {
    StartFakeEndpoints();
  }

  private void StartFakeEndpoints()
  {
    this.hostForFakeEndpoints = new WebHostBuilder()
    .UseKestrel()
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseStartup<FakeStartup>()
    .UseUrls("http://localhost:5001")
    .Build();

    new Thread(() => this.hostForFakeEndpoints.Run()).Start();
  }

  public class FakeStartup
  {
    public void Configure(IApplicationBuilder app)
    {
      app.UseOwin(buildFunc => buildFunc.UseNancy());
    }
  }
}
```

Создаем приложение ASP.NET Core

Предоставляем приложению ASP.NET Core возможность прослушивания на порте 5001

Добавляем Nancy в приложение ASP.NET Core

Используем FakeStartup для загрузки приложения ASP.NET Core

Далее добавим в этот класс метод с атрибутом `Fact`: в дальнейшем `xUnit` после запуска пойдет этот класс и создаст его экземпляр для выполнения метода с атрибутом `Fact`. Конструктор запустит `Nancy`, который автоматически обнаружит модули `MockEventsFeed` и `MockUsersModule`, и предоставит для использования описанные в них конечные точки. Вот и все, что нужно сделать для создания имитационных конечных точек в процессе эксплуатационного тестирования.

Запуск всех процессов тестируемого микросервиса

После запуска имитационных конечных точек можно запустить и микросервис `Loyalty Program`. Микросервис состоит из двух процессов: приложения `Nancy` и потребителя событий. Добавим код для их запуска в тестовую конфигурацию в файле `RegisterUserAndGetNotification`. В листинге 7.15 показан только новый код — существующий код для запуска и останова `Nancy` мы опустим.

Листинг 7.15. Запуск микросервиса в отдельном процессе

```
public class RegisterUserAndGetNotification : IDisposable
{
    ...
    private Process eventConsumer;
    private Process web;

    public RegisterUserAndGetNotification()
    {
        StartLoyaltyProgram();
        ...
    }

    private void StartLoyaltyProgram()
    {
        StartEventConsumer();
        StartLoyaltyProgramApi();
    }

    private void StartLoyaltyProgramApi()
    {
        var apiInfo = new ProcessStartInfo("dotnet.exe")
        {
            Arguments = "run",
            WorkingDirectory = "../LoyaltyProgram"
        };
        this.api = Process.Start(apiInfo);

        var eventConsumerInfo = new ProcessStartInfo("dotnet.exe")
        {
            Arguments = "run localhost:5001",
```

Настройки для выполнения команды `dotnetrun` из каталога проекта `LoyaltyProgram`

Запуск процесса `LoyaltyProgram`

Настройки для запуска потребителя событий

```

    WorkingDirectory = "../LoyaltyProgramEventConsumer"
};
this.eventConsumer = Process.Start(eventConsumerInfo);
}
                                Запуск процесса потребителя событий

public void Dispose()
{
    this.eventConsumer.Dispose();
    this.api.Dispose();
}
}

```

← Завершение процессов и освобождение ресурсов

Этот код порождает два процесса `dotnet`, по одному для каждого процесса в микросервисе `Loyalty Program`. Это схоже с выполнением команды `dotnet` из командной строки, так что приложение `Nancy` запускается так же, как обычно. Запуск потребителя событий происходит иначе, и нам необходимо для этого решить две проблемы.

- ❑ Ожидается, что потребитель событий будет выполняться как процесс `Windows`. Теперь же необходимо, чтобы он мог выполняться и как обычный процесс.
- ❑ В строке из листинга 7.15:

```
Arguments = "run localhost:5001",
```

потребителю событий неопытен аргумент командной строки `localhost:5001`, представляющий собой имя хоста для имитационных конечных точек, которые потребитель событий должен использовать вместо настоящих микросервисов.

Решение обеих проблем сложностей не представляет. Необходимо просто изменить метод `Main` потребителя событий следующим образом (листинг 7.16).

Листинг 7.16. Обеспечиваем возможность запуска потребителя в виде как `Windows`-процесса, так и обычного процесса

```

public static void Main(string[] args) => new Program().Entry(args);

public void Entry(string[] args)
{
    this.subscriber = new EventSubscriber(args[0]);
    if (args.Length >= 2 && args[1].Equals("--service"))
    {
        Run(this);
    }
    else
    {
        OnStart(null);
        Console.ReadLine();
    }
}

```

← Читаем имя хоста из аргумента командной строки

← Если среди аргументов командной строки имеется `--service`, то запускаем в виде сервиса

← Запускаем метод `start` вручную

Теперь оба процесса микросервиса `Loyalty Program` запускаются из кода инициализации теста. В качестве приятного побочного эффекта изменений в потребителе событий становится удобнее запускать их вручную для целей тестирования.

Выполнение тестового сценария для тестируемого микросервиса

Итак, мы наконец-то готовы написать сам тест. Он состоит из трех шагов.

1. Выполнение HTTP-запроса для регистрации пользователя.
2. Ожидание выполнения микросервисом Loyalty Program опроса на предмет появления новых событий.
3. Контроль выполнения запроса к конечной точке уведомлений.

Код теста помещается в файл `RegisterUserAndGetNotification` и имеет следующий вид (листинг 7.17).

Листинг 7.17. Эксплуатационный тест с использованием внешней программы лояльности

```
[Fact]
public void Scenario()
{
    RegisterNewUser();
    WaitForConsumerToReadSpecialOffersEvents();
    AssertNotificationWasSent();
}

private async Task RegisterNewUser()
{
    using (var httpClient = new HttpClient())
    {
        httpClient.BaseAddress = new Uri("http://localhost:5000");
        var response = await
            httpClient.PostAsync(
                "/users/",
                new StringContent(
                    JsonConvert.SerializeObject(new LoyaltyProgramUser()),
                    Encoding.UTF8,
                    "application/json")).ConfigureAwait(false);
        Assert.Equal(HttpStatusCode.Created, response.StatusCode);
        Console.WriteLine("registered users");
    }
}

private static void WaitForConsumerToReadSpecialOffersEvents()
{
    Assert.True(MockEventFeed.pollled.WaitOne(30000));
    Thread.Sleep(100);
}

private static void AssertNotificationWasSent()
{
    Assert.True(MockNotifications.NotificationWasSent);
}
```

Вставляем данные пользователя в запрос

Отправляем запрос для регистрации пользователя

Ждем опроса микросервисом ленты событий. Если этого не произойдет, считаем выполнение неудачным

Ждем, чтобы у микросервиса было время для обработки полученного из ленты события

Запустить тест в оболочке Powershell можно с помощью команды `dotnet`:

```
PS> dotnet test
```

Эта команда приведет к открытию двух командных окон, но одному для каждого из процессов микросервиса Loyalty Program. Тест выполняется, и по его завершении оба окна закрываются. Выводимые xUnit результаты выглядят следующим образом:

```
Discovering: LoyaltyProgramIntegrationTest
  Discovered: LoyaltyProgramIntegrationTest
  Starting: LoyaltyProgramIntegrationTest
    LoyaltyProgramIntegrationTests.RegisterUserAndGetNotification.Scenario
  Finished: LoyaltyProgramIntegrationTest
=== TEST EXECUTION SUMMARY ===
  LoyaltyProgramIntegrationTest Total: 1, Errors: 0, Failed: 0, Skipped:
  ➡ 0, Time: 12.563s
```

Этот тест выполняется медленно, и нам пришлось проделать некоторые подготовительные работы, прежде чем написать его. Именно поэтому подобные тесты находятся выше в пирамиде тестов, чем написанные ранее модульные тесты. Таких тестов для микросервиса должно быть лишь несколько, а модульных может быть много.

7.5. Резюме

- ❑ Пирамида тестов показывает, что у вас должно быть небольшое количество системных тестов для тестирования системы в целом, несколько эксплуатационных тестов для каждого микросервиса и множество модульных тестов для каждого микросервиса.
- ❑ Системные тесты окажутся, вероятно, медленными и очень неточными.
- ❑ Писать системные тесты следует для важнейших сценариев успешного использования, чтобы охватить ими большую часть системы в целом.
- ❑ Эксплуатационные тесты, скорее всего, будут медленными, но они быстрее и точнее, чем системные тесты.
- ❑ Желательно писать эксплуатационные тесты для сценариев успешного выполнения и важнейших сценариев неудачного выполнения для каждого микросервиса. Это позволяет охватить тестами все микросервисы полнее, чем при наличии одних системных тестов.
- ❑ Технологию написания эксплуатационных тестов можно использовать как основу для написания контрактных тестов, служащих для проверки допущений, принимаемых в одном микросервисе относительно API и функционирования другого. Если говорить на языке пирамиды тестов, контрактные тесты располагаются между системными и эксплуатационными тестами.
- ❑ Модульные тесты быстрые, и желательно, чтобы они таковыми и оставались. Они также точны, поскольку их объект — конкретный узко очерченный элемент функциональности.
- ❑ Следует писать модульные тесты для сценариев как успешного, так и неудачного выполнения. С их помощью можно охватить граничные случаи, которые сложно охватить тестами более высокого уровня.

- ❑ Я рекомендую работать с каждым из микросервисов в стиле «снаружи внутрь»: сначала создавать эксплуатационные тесты и приступить к написанию модульных тестов тогда, когда становится неудобно работать с эксплуатационными.
- ❑ Библиотека `Nancy.Testing` — замечательное дополнение к фреймворку `Nancy`, значительно упрощающее написание конечных точек в модулях `Nancy`.
- ❑ Для тестирования конечных точек в библиотеке `Nancy.Testing` через удобный API, позволяющий имитировать HTTP-запросы, можно использовать тип `Browser`. Выполняемые через объект `Browser` вызовы выглядят точно так же, как настоящие HTTP-запросы к обработчикам конечных точек в модулях `Nancy`.
- ❑ Тестировать конечные точки через объект `Browser` следует как с настоящими, так и с имитационными хранилищами данных.
- ❑ Можно создавать эксплуатационные тесты, в которых:
 - писать имитационные конечные точки для микросервисов, взаимодействующих с тестируемым, и использовать `Nancy` для их размещения в тестовом процессе;
 - запускать все процессы тестируемого микросервиса, передавая настройки посредством аргументов командной строки;
 - писать сценарии, взаимодействующие с тестируемым микросервисом посредством HTTP-запросов;
 - контролировать как ответы тестируемого микросервиса, так и запросы, сделанные им к взаимодействующим с ним микросервисам.
- ❑ Для написания и запуска автоматизированных тестов можно использовать фреймворк тестирования `xUnit`.
- ❑ Запускать `xUnit` можно с помощью утилиты командной строки `dotnet`.

Часть III

Сквозная функциональность: создание платформы многоразового кода для микросервисов

В этой части книги мы создадим платформу для решения некоторых важных вопросов сквозной функциональности, пригодной для использования сразу во многих микросервисах. Эта сквозная функциональность включает мониторинг, журналирование, передачу маркеров корреляции с запросами от микросервиса к микросервису и связанные с ней вопросы безопасности. Все это обеспечивает нормальное функционирование микросервисов при эксплуатации в производственной среде, предоставляя информацию о состоянии всех микросервисов и позволяя отслеживать бизнес-операции между ними.

Можно реализовать подобную функциональность в каждом из микросервисов или создать повторно используемую реализацию, которую потом можно будет применить в нескольких микросервисах. Первый вариант, очевидно, приводит к дублированию затрат усилий, а повторное использование реализации приводит к связанности между микросервисами. Другими словами, желаемая степень повторного использования представляет собой компромисс между взаимной независимостью микросервисов и снижением объема требуемых работ. Решение об этом компромиссе вы должны принять сами с учетом контекста системы. В этой книге я продемонстрирую возможность повторного использования кода для журналирования запросов и показателей производительности, передачи маркеров корреляции и обеспечения безопасности обращений между микросервисами. (Обратите внимание на то, что повторно применять функциональность следует осмотрительно, делая повторно используемыми только те элементы, которые в практически одинаковом виде встречаются во многих микросервисах.)

Я кратко упомянул промежуточное ПО OWIN в главе 1, а в главе 8 мы рассмотрим подробнее, как OWIN функционирует и что вообще такое промежуточное ПО. В главах 9 и 10 будем реализовывать сквозную функциональность в виде промежуточного ПО OWIN. В главе 11 создадим на основе уже реализованного промежуточного ПО повторно используемую платформу микросервисов. Эту платформу можно будет с легкостью добавлять к новым микросервисам, что даст возможность быстро создавать микросервисы с хорошо реализованной сквозной функциональностью.

8

Знакомство с OWIN: написание и тестирование промежуточного ПО OWIN

В этой главе:

- ❑ функциональность, сквозная для нескольких микросервисов;
- ❑ основные сведения о стандарте OWIN, промежуточное ПО OWIN и конвейеры OWIN;
- ❑ написание промежуточного ПО OWIN;
- ❑ тестирование промежуточного ПО OWIN и конвейеров OWIN.

При реализации системы микросервисов некоторые элементы функциональности оказываются сквозными для всей системы. Существуют действия, которые должен выполнять любой микросервис, причем зачастую они тождественны поддержанию работоспособности системы при эксплуатации в производственной среде:

- ❑ мониторинг;
- ❑ журналирование ошибок, запросов, показателей производительности и т. д.;
- ❑ соблюдение безопасности;
- ❑ реализация стратегий, относящихся к технологиям, используемым сразу во многих микросервисах, например обработки соединений с базой данных.

Все они хорошо подходят для реализации в виде промежуточного ПО OWIN. В этой главе рассмотрим использование промежуточного ПО OWIN для реализации сквозной функциональности.

8.1. Реализация сквозной функциональности

Отдельный микросервис состоит из множества компонентов. Например, в главе 2 говорится, что микросервис Shopping Cart состоит из показанных на рис. 8.1 компонентов.

Ни один из этих компонентов не нацелен на воплощение упомянутой во введении к данной главе сквозной функциональности: мониторинга, журналирования запросов и т. д. Более того, реализация в любом из них этой функциональности не станет удачным решением. Почему? Потому что все показанные на рис. 8.1 компоненты

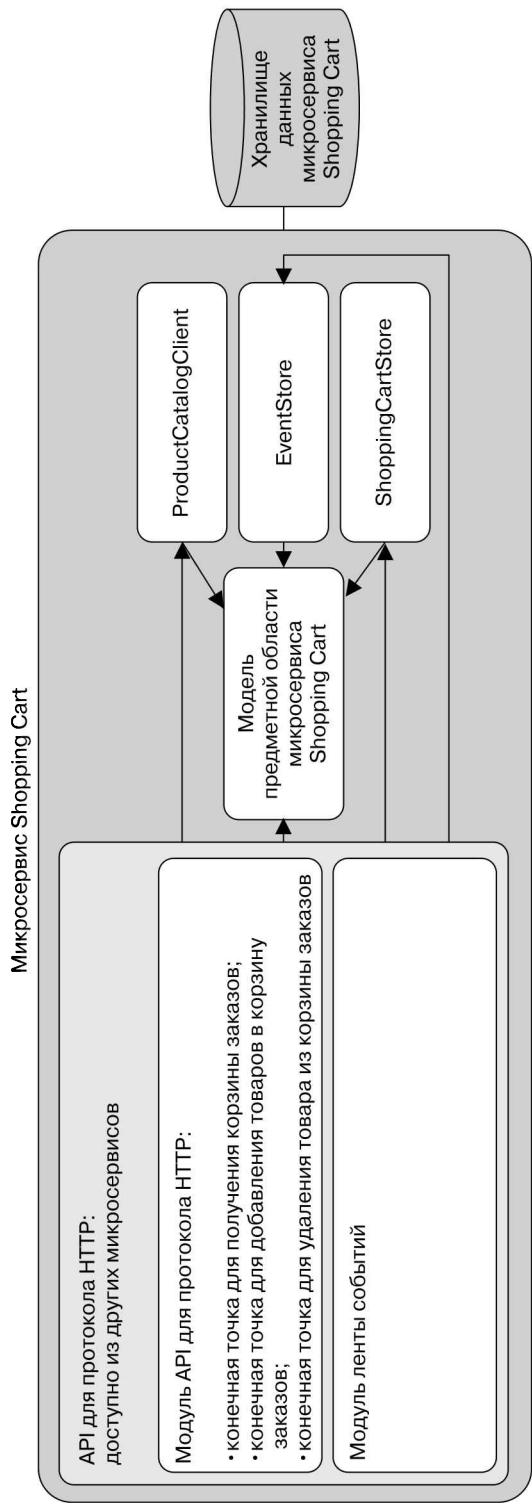


Рис. 8.1. Приведенное в предыдущих главах представление микросервиса Shopping Cart демонстрирует несколько компонентов, которые совместно воплощают функциональность микросервиса Shopping Cart

реализуют относящуюся к микросервису Shopping Cart функциональность. А сквозная функциональность, напротив, не относится к какому-то конкретному микросервису. Следовательно, ее нужно реализовать в компонентах, не связанных с показанными на рис. 8.1. Посмотрев на микросервис Shopping Cart под другим углом (рис. 8.2), мы видим, что он получает HTTP-запросы от веб-сервера, обрабатывает их с помощью различных своих компонентов и возвращает ответы веб-серверу, который затем отправляет их обратно вызывающей стороне.

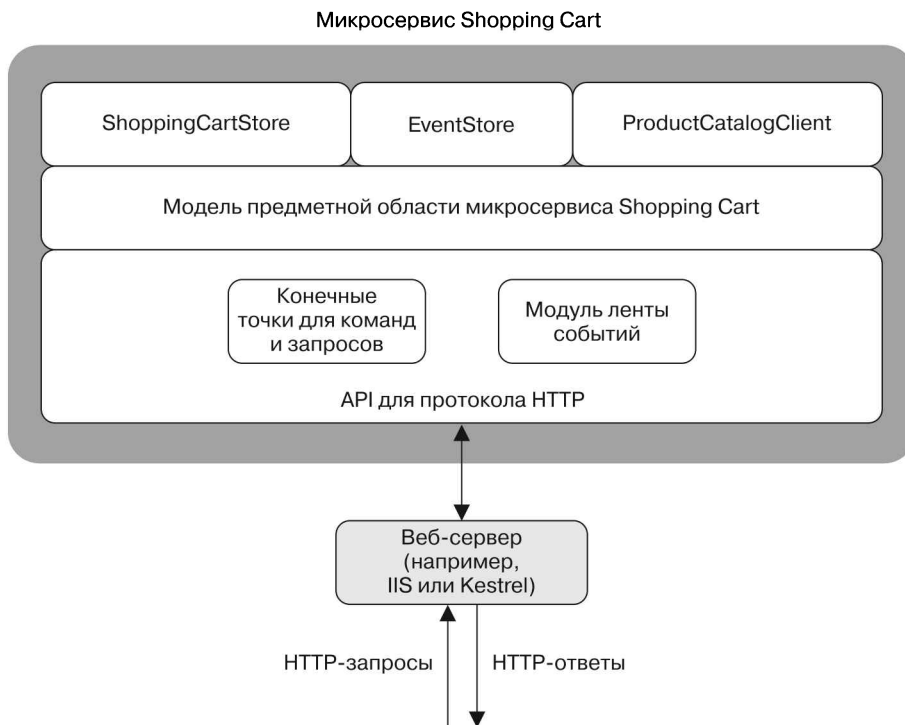


Рис. 8.2. Микросервис Shopping Cart принимает HTTP-запросы через веб-сервер

Как я упомянул ранее, желательно, чтобы код сквозной функциональности был реализован отдельно от показанных на рис. 8.1 компонентов. Более того, сквозная функциональность работает с микросервисом в целом. А значит, удачным решением будет разместить код сквозной функциональности между веб-сервером и обработчиками конечных точек в модулях Nancy (рис. 8.3).

Элементы кода между веб-сервером и обработчиками конечных точек формируют конвейер: все запросы проходят через каждый из этих элементов по очереди, прежде чем достичь обработчика конечной точки в модуле Nancy. Аналогично ответ обработчика конечной точки проходит через этот же конвейер, прежде чем достичь веб-сервера, пересылающего ответ обратно вызывающей стороне. В этой главе мы будем использовать для реализации подобного конвейера интерфейс OWIN, отдельные элементы конвейера называются *промежуточным ПО OWIN*.

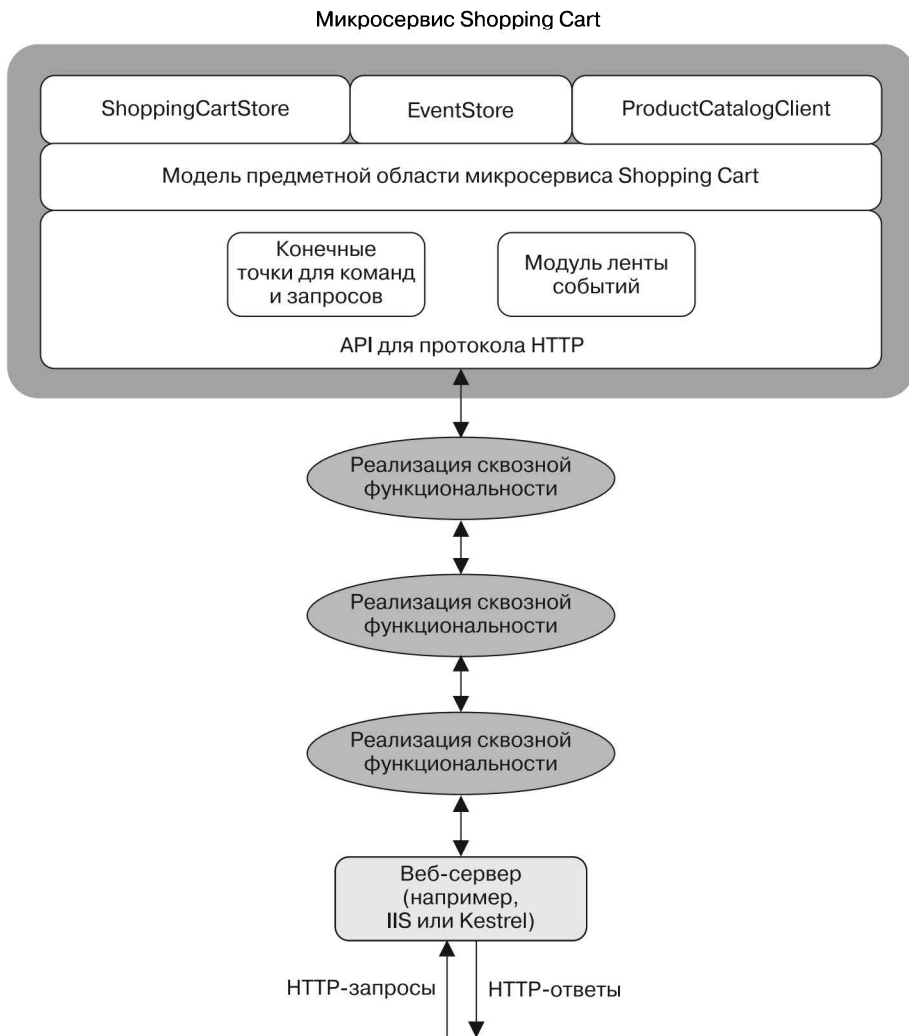


Рис. 8.3. Для реализации сквозной функциональности можно использовать конвейер промежуточного ПО между веб-сервером и обработчиком конечной точки

8.2. Конвейер OWIN

Мы вскользь упомянули стандарт OWIN (<http://owin.org/>) в главе 1, теперь настало время взглянуть на него поближе. OWIN — формируемый сообществом разработчиков стандарт взаимодействия между веб-серверами .NET, веб-фреймворками .NET и элементами конвейера, называемыми *промежуточным ПО (middleware)*. В данном случае ASP.NET Core играет роль веб-сервера, а Nancy — веб-фреймворка. OWIN позволяет размещать между ними элементы промежуточного ПО, выполняемые при каждом запросе. Все запросы проходят по всем элементам промежуточного программного обеспечения по очереди.

Схема на рис. 8.4 представляет собой конвейер OWIN, включающий один элемент промежуточного ПО OWIN. Стандарт OWIN определяет используемый для связи между всеми частями конвейера интерфейс. Этот единообразный интерфейс реализуют все части конвейера: все элементы промежуточного ПО и веб-фреймворк. Веб-сервер не реализует этот интерфейс, а взаимодействует через него с остальными частями конвейера. Благодаря единообразному интерфейсу можно компоновать конвейер так, как вам нужно. Можно вставлять в него дополнительное промежуточное ПО, удалять из него отдельные элементы или менять их местами. Так как они используют один и тот же интерфейс, их можно переставлять местами произвольным образом.

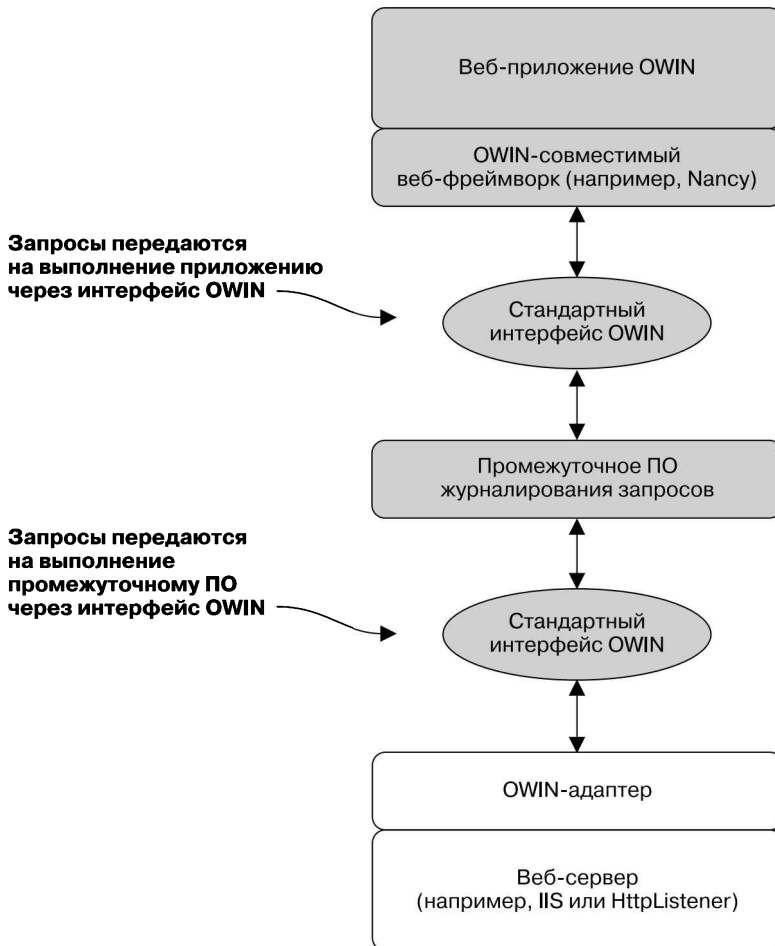
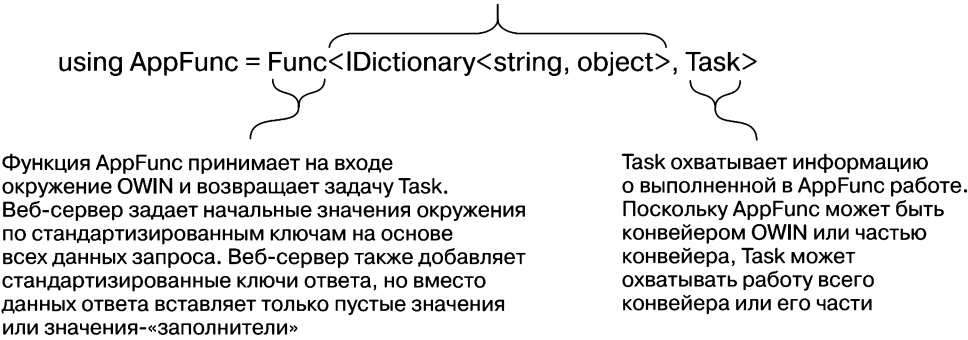


Рис. 8.4. Веб-сервер OWIN с промежуточным ПО OWIN и веб-приложением OWIN поверх него. Веб-сервер передает входящий запрос на выполнение в расположенные выше слои — в данном случае в журналирующее запросы промежуточное ПО, которое записывает в журнал сообщение относительно запроса, после чего передает его на выполнение веб-приложению. Для веб-сервера промежуточное ПО выглядит OWIN-совместимым приложением, а для приложения промежуточное ПО выглядит OWIN-совместимым веб-сервером

При прохождении запроса и ответа через промежуточное ПО оно может их читать и даже менять. Используемый между элементами конвейера OWIN интерфейс представляет собой не интерфейс C#, а сигнатуру функции. Каждый элемент промежуточного ПО и даже веб-фреймворк в конце конвейера представляет собой функцию с сигнатурой, совместимой со следующим определением типа AppFunc:

Окружение OWIN — это словарь, содержащий всю информацию о запросе и ответе. Эти данные содержатся в окружении OWIN в соответствии с набором стандартизированных ключей



using AppFunc = Func<IDictionary<string, object>, Task>

Так как название AppFunc часто используется для этой сигнатуры функции, то я тоже буду использовать это условное обозначение. Идея заключается в последовательном соединении нескольких AppFunc с передачей окружения OWIN от одного элемента промежуточного ПО к следующему при обработке запроса.

Окружение OWIN содержит все данные запроса и ответа по специфицированным стандартом OWIN ключам. В табл. 8.1 приведен краткий список специфицированных стандартом OWIN ключей запроса. Более подробное описание всех ключей можно найти в стандарте, но для всего того, что мы будем делать здесь, этого достаточно. Аналогичный список ключей ответа приведен в табл. 8.2.

Таблица 8.1. Ключи запроса окружения OWIN

Обязательный	Ключ	Значение
Да	owin.RequestBody	Stream с телом запроса, если таковое имеется
Да	owin.RequestHeaders	IDictionary<string, string[]> заголовков запроса
Да	owin.RequestMethod	Метод HTTP-запроса в виде строки (например, "GET", "POST")
Да	owin.RequestPath	Объект типа string, содержащий путь запроса относительно корневого [URL]
Да	owin.RequestPathBase	Объект типа string, содержащий корневую часть пути запроса
Да	owin.RequestProtocol	Название и версия протокола запроса в виде строки (например, "HTTP/1.1")
Да	owin.RequestQueryString	Объект типа string, содержащий строку запроса HTTP-запроса
Да	owin.RequestScheme	URI-схема запроса в виде объекта типа string (например, "http", "https")

Таблица 8.2. Ключи ответа окружения OWIN

Обязательный	Ключ	Значение
Да	owin.ResponseBody	Stream, используемый при записи тела ответа, если таковое имеется
Да	owin.ResponseHeaders	IDictionary<string, string[]> заголовков ответа
Нет	owin.ResponseStatusCode	Код состояния HTTP-ответа в виде объекта типа int. По умолчанию равен 200
Нет	owin.ResponseReasonPhrase	Объект типа string, содержащий пояснение к соответствующему коду состояния
Нет	owin.ResponseProtocol	Объект типа string, содержащий название и версию протокола (например, "HTTP/1.1"). Если таковые не указаны, то по умолчанию используется значение ключа owin.RequestProtocol

Поскольку окружение OWIN передается по конвейеру, любой из встречающихся на его пути элементов промежуточного ПО может при необходимости изменить значение любого из ключей. Например, чтобы задать код состояния ответа, промежуточному ПО достаточно лишь задать значение ключа `owin.ResponseStatusCode`.

Отмечу, что `AppFunc` может быть не только отдельным элементом промежуточного ПО OWIN, но и конвейером. Например, вызывающему функцию `AppFunc` веб-серверу совершенно безразлично, отдельный это элемент промежуточного ПО или длинный конвейер, оканчивающийся веб-фреймворком.

При настройке конвейера из элементов промежуточного ПО OWIN, по сути, происходит последовательное присоединение функций `AppFunc`: первый элемент промежуточного ПО получает ссылку на второй и т. д. Располагающееся перед веб-фреймворком промежуточное ПО содержит ссылку на реализацию `AppFunc` в веб-фреймворке. После построения конвейера и поступления запроса сервер формирует окружение OWIN и передает его в конвейер.

Для упрощения настройки цепочки промежуточного ПО OWIN применяются функции другого типа, `MidFunc`:

Функция `MidFunc` принимает на входе функцию `AppFunc`, соответствующую оставшейся части конвейера. Как правило, этот аргумент получает название `next`

`usingMidFunc = Func<AppFunc, AppFunc>`

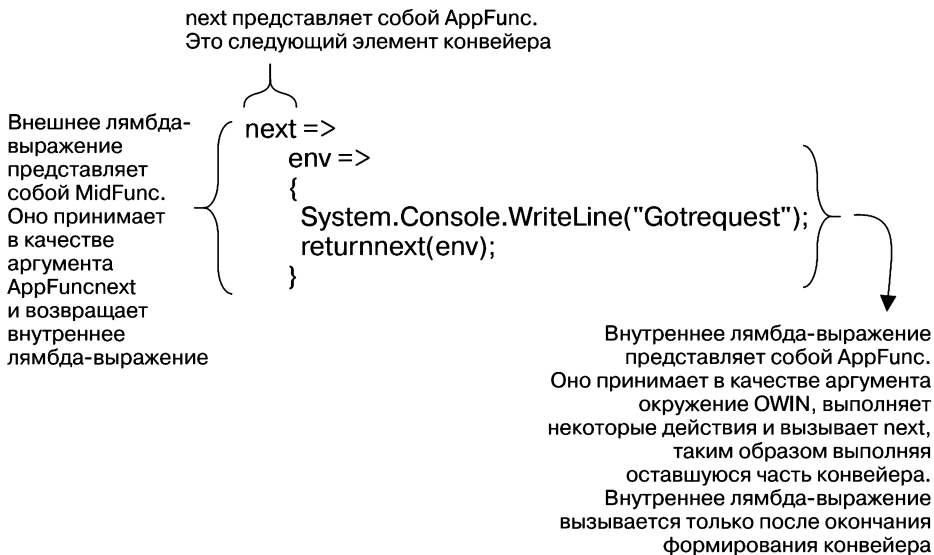
`MidFunc` — функция для преобразования одной `AppFunc` в другую `AppFunc`. Она используется для сборки конвейеров OWIN

`MidFunc` возвращает новую `AppFunc`, представляющую собой поступившую на входе `AppFunc` с добавленным перед ней промежуточным ПО

Принимающая на входе `AppFunc` функция возвращает другую `AppFunc`. `MidFunc` — это функция, которая принимает на входе функцию `AppFunc` с реализацией конвейера

OWIN, состоящего из нуля или более элементов промежуточного ПО OWIN, ставит перед ней элемент промежуточного ПО и возвращает полученную `AppFunc` в качестве нового конвейера.

Несмотря на кажущуюся сложность, реализация `MidFunc` довольно проста. В качестве примера рассмотрим следующий элемент промежуточного ПО OWIN, с которым вы уже встречались в главе 1:



Теперь мы знаем достаточно для того, чтобы разобраться в этом коде: лямбда-функция реализует `MidFunc`. Она принимает аргумент (`next`), реализующий `AppFunc`, и возвращает другую функцию `AppFunc`, которая вызывает передаваемую в `next` исходную функцию, но перед этим выполняет еще и запись в консоль. После настройки конвейера значение (`next`) не меняется, но при каждом запросе формируется и передается в аргументе `env` новое окружение.

К идее промежуточного ПО OWIN необходимо привыкнуть, но, усвоив ее, вы оцените ее простоту и возможности, показанные в следующих главах.

Что должно обрабатываться в промежуточном ПО OWIN, а что — в модулях Nancy

Стандарт OWIN позволяет создавать конвейеры, через которые будут проходить все запросы. Различные части конвейера смогут реагировать на запрос и вносить изменения в ответ. Аналогично можно обрабатывать запросы и создавать ответы, используя фреймворк Nancy. С помощью Nancy обработка запросов происходит в основном в обработчиках маршрутов в модулях Nancy, но можно также добавлять код, выполняемый до и после обработчика маршрута. Сделать это можно, добавляя код в конвейер `Before` или `After`, как упоминалось в главе 6. По сути, Nancy также предоставляет возможность создания конвейеров. Но где и что должно обрабатываться?

Чтобы определиться с тем, что нужно реализовывать в промежуточном ПО OWIN, а что — в модулях Nancy, можно воспользоваться несколькими простыми правилами.

- ❑ Нацеленный на вонложение сквозной функциональности (и предназначенный для использования во многих микросервисах) код должен быть реализован в промежуточном ПО OWIN. Эти повторно выполняемые фрагменты кода не должны иметь много зависимостей, чтобы они не навязывали слишком много технологических решений использующим их микросервисам. Следовательно, желательно не делать их зависимыми от Nancy.
- ❑ Нацеленный на воплощение бизнес- или предметно-ориентированного правила отдельного микросервиса код должен быть частью кода приложения, располагающегося за модулем Nancy. Подобный код не основывается на протоколе HTTP, а значит, не должен зависеть от OWIN или Nancy. Размещение его в отдельном компоненте, например модели предметной области, позволяет сохранить аккуратность и читабельность реализаций бизнес- и предметно-ориентированных правил.
- ❑ Код, предназначенный для обработки HTTP-запросов и ответов с учетом специфики конкретной конечной точки, должен находиться в модуле Nancy. Этот код интерпретирует входящие HTTP-запросы, после чего передает управление модели предметной области. Данная интерпретация тесно связана как с протоколом HTTP, так и со спецификой конечной точки, поэтому модуль Nancy подходит для нее лучше всего.
- ❑ Код, нацеленный на вонложение функциональности, сквозной для всех конечных точек в одном микросервисе, но не для нескольких микросервисов, обычно должен располагаться в промежуточном ПО OWIN, однако не всегда. Чем более элемент функциональности относится к техническим возможностям, тем больше я склоняюсь к реализации его в промежуточном ПО, и, наоборот, чем более он относится к бизнес-возможностям, тем больше я склоняюсь к реализации его в модулях Nancy или в редких случаях в обработчиках `Before` или `After` Nancy.

Мы познакомимся со стандартом OWIN и обсудили, для чего он предназначен. В оставшейся части главы займемся написанием и тестированием промежуточного ПО OWIN.

8.3. Написание промежуточного ПО

Рассмотрим два способа написания промежуточного ПО OWIN.

- ❑ *В виде лямбда-выражения.* Вы уже встречали промежуточное ПО в виде лямбда-выражения в главе 1, а также ранее в данной главе. В следующем разделе снова займемся этим вариантом.
- ❑ *В виде класса, содержащего реализующий `AppFunc` метод.* Вы увидите этот вариант в разделе «Классы промежуточного ПО» далее в этом разделе.

В обоих случаях можно воспользоваться удобной библиотекой *LibOwin*, с помощью которой можно работать с окружением OWIN через тины, вместо того чтобы использовать непосредственно окружение `IDictionary<string, object>`.

LibOwin

LibOwin — маленькая библиотека, которая поможет нам в работе с OWIN. Она несколько отличается от других используемых нами библиотек: это *библиотека исходного кода*, то есть состоит из исходного кода, а не из сборок для платформы .NET. Библиотека LibOwin состоит из одного файла, `LibOwin.cs`, содержащего несколько упрощающих работу с OWIN типов. В основном мы будем использовать тип `OwinContext`, служащий адаптером для словаря окружения OWIN и предоставляющий возможность доступа к ключам в словаре окружения через строго типизированные свойства. Следующий фрагмент кода, например, извлекает HTTP-метод HTTP-запроса из окружения:

```
// env — словарь окружения OWIN
var context = new OwinContext(env);
var method = context.Request.Method
// Выполняем какие-то действия с переменной method
```

Помимо реализаций промежуточного ПО, можно использовать библиотеку LibOwin при написании тестов для промежуточного ПО, для создания окружений OWIN, передаваемых в промежуточное ПО для его тестирования, и работы с ними.

На момент написания данной книги команда `dotnet restore` не поддерживает распространение исходного кода посредством пакетов NuGet, так что вместо использования пакета LibOwin системы управления пакетами NuGet вам придется скачать файл `LibOwin.cs`. LibOwin можно найти на GitHub по адресу <https://github.com/damianh/LibOwin>, а `LibOwin.cs` — по адресу <http://mng.bz/8pRq>. Для скачивания этого файла из командной оболочки PowerShell можно воспользоваться командой `wget`:

```
PS> wget https://raw.githubusercontent.com/<lineArrow
➡ />damianh/LibOwin/master/src/LibOwin
```

```
/LibOwin.cs -OutFile LibOwin.cs
```

После выполнения этой команды у вас появится своя копия файла `LibOwin.cs`, которую можно добавить в проект аналогично любому другому исходному файлу. Для кода из файла `LibOwin.cs` необходимы два пакета NuGet — `System.Security.Claims` и `System.Globalization`, которые можно добавить точно так же, как любые другие пакеты NuGet.

Промежуточное ПО в виде лямбда-выражений

Вы уже видели реализацию промежуточного ПО с помощью лямбда-функции. В этом разделе я покажу, как можно воспользоваться библиотекой LibOwin внутри лямбда-функции промежуточного ПО, после чего наномню, как добавлять лямбда-промежуточное ПО в конвейеры OWIN в файле `startup.cs`.

Вот лямбда-промежуточное ПО, с которым вы уже неоднократно сталкивались:

```
next => ← next представляет собой
           AppFunc. Лямбда-выражение
           в целом представляет собой MidFunc
env => ←
{
    System.Console.WriteLine("Got request");
    return next(env);
}
```

Внутреннее лямбда-выражение представляет собой AppFunc. Оно принимает в качестве аргумента окружение OWIN в переменной env

Если нам нужно записать в консоль не просто статическую строку, а, скажем, путь и метод запроса, можно воспользоваться типом `OwinContext` из библиотеки `LibOwin` (листинг 8.1).

Листинг 8.1. Использование типа `OwinContext` для извлечения подробных данных запроса

```
next =>
  env =>
  {
    var context = new OwinContext(env);
    var method = context.Request.Method;
    var path = context.Request.Path;
    System.Console.WriteLine($"Got request: {method} {path}");
    return next(env);
  }
```

env представляет собой словарь окружения OWIN

Создаем строго типизированный объект `OwinContext` на основе окружения OWIN

С легкостью извлекаем данные запроса через свойства `OwinContext`

Мы создали объект `OwinContext` на основе окружения OWIN и извлекаем данные через его свойства. Для добавления подобного промежуточного ПО в конвейер OWIN воспользуемся для интерфейса `IApplicationBuilder` методом расширения `UseOwin`, предоставляемым в классе `Startup` платформы ASP.NET Core (листинг 8.2).

Листинг 8.2. Использование метода `UseOwin` для создания конвейера OWIN

```
public class Startup
{
  public void Configure(IApplicationBuilder app)
  {
    app.UseOwin(
      buildFunc => buildFunc(next => env =>
      {
        var context = new OwinContext(env);
        var method = context.Request.Method;
        var path = context.Request.Path;
        System.Console.WriteLine($"Got request: {method} {path}");
        return next(env);
      }
    ));
  }
}
```

Предоставляет возможность использования OWIN на платформе ASP.NET Core

ASP.NET Core вызывает этот метод во время загрузки

buildFunc формирует конвейер OWIN из `MidFunc`

Мы использовали метод расширения `UseOwin` в предыдущих главах в каждом проекте с HTTP API для добавления фреймворка Nancy в конвейер OWIN.

Классы промежуточного ПО

Создание промежуточного ПО в виде лямбда-функций вполне возможно, но оно сильно затрудняется по мере усложнения промежуточного ПО. В подобных случаях удобнее использовать для реализации промежуточного ПО класс.

Чтобы использовать класс для реализации промежуточного ПО, необходимо создать класс, в котором есть метод с сигнатурой `AppFunc` (листинг 8.3).

Листинг 8.3. Реализация промежуточного ПО в виде класса

```
public class ConsoleMiddleware
{
    private AppFunc next;

    public ConsoleMiddleware(AppFunc next)
    {
        this.next = next;
    }

    public Task Invoke(IDictionary<string, object> env)
    {
        var context = new OwinContext(env);
        var method = context.Request.Method;
        var path = context.Request.Path;
        System.Console.WriteLine($"Got request: {method} {path}");
        return next(env);
    }
}
```

При создании экземпляра получаем ссылку на следующий элемент

Соответствует сигнатуре AppFunc

Подобное промежуточное ПО получает в конструкторе ссылку на следующую AppFunc и сохраняет ее в закрытой переменной, чтобы можно было вызвать ее при необходимости в методе Invoke, реализующем функциональность промежуточного ПО. Для добавления такого промежуточного ПО в конвейер OWIN необходимо создать экземпляр этого класса, после чего делегировать выполнение методу Invoke:

```
app.UseOwin(buildFunc =>
    buildFunc(next => new ConsoleMiddleware(next).Invoke));
```

Вы познакомились с реализацией промежуточного ПО в виде как лямбда-выражений, так и классов. Далее займемся тестированием промежуточного ПО.

8.4. Тестирование промежуточного ПО и конвейеров

Тестирование промежуточного ПО не представляет сложностей: достаточно воспользоваться LibOwin для создания окружения OWIN для тех сценариев, которые нужно протестировать, после чего вызвать промежуточное ПО, передав ему это окружение.

ПРИМЕЧАНИЕ

Как описывалось в предыдущем разделе, создавать тестовые проекты можно как из Visual Studio, так и с помощью Yeoman. Эти проекты, как и приведенный далее, использовали xUnit в качестве фреймворка тестирования.

Сначала нужно решить одну маленькую проблему: чтобы вызвать промежуточное ПО, необходимо передать ему значение для next. Обойти эту проблему можно, передав ему AppFunc, которая ничего не делает:

```
AppFunc noOp = env => Task.FromResult(0);
```

noOp — сокращение от «холостая команда» (no operation)

Подобное фиктивное промежуточное ПО используется в тестах в качестве заглушки, передаваемой тестируемому промежуточному ПО.

Теперь можно писать тесты для промежуточного ПО. Допустим, мы хотим протестировать следующее промежуточное ПО на базе лямбда-выражений (листинг 8.4).

Листинг 8.4. Пример элемента промежуточного ПО

```
namespace Middleware
{
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using LibOwin;

    using AppFunc = Func<IDictionary<string, object>, Task>;

    public class Middleware
    {
        public Func<AppFunc, AppFunc> Impl = ← Промежуточное ПО
            next => async env =>                в виде лямбда-выражения
        {
            var ctx = new OwinContext(env);
            if (ctx.Request.Path.Value == "/test/path") ←
                ctx.Response.StatusCode = 404;           Если путь запроса равен
            else                                           /test/path, возвращаем
                await next(env);                          404 NotFound (404 Не найдено).
                                                         В противном случае
                                                         вызываем оставшуюся
                                                         часть конвейера
        }
    }
}
```

Тест для этого промежуточного ПО можно написать следующим образом (листинг 8.5).

Листинг 8.5. Тест, непосредственно вызывающий промежуточное ПО

```
namespace OwinMiddlewareTests
{
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Xunit;
    using LibOwin;

    using AppFunc = Func<IDictionary<string, object>, Task>;

    public class Middleware_should
    {
        private AppFunc noOp = env => Task.FromResult(0); ← Холодная AppFunc

        [Fact]
        public void Return404_for_test_path() ←
        {                                           Задаёт значения окружения
            var ctx = new OwinContext();          OWIN для данного теста
            ctx.Request.Scheme =
                LibOwin.Infrastructure.Constants.Https;
        }
    }
}
```

```
ctx.Request.Path = new PathString("/test/path");
ctx.Request.Method = "GET";
```

```
var pipeline = Middleware.Middleware.Impl(noOp));
```

```
var env = ctx.Environment;
pipeline(env);
```

```
Assert.Equal(404, ctx.Response.StatusCode);
```

```
}
}
}
```

Создает конвейер
из тестируемого
промежуточного ПО
и холостой AppFunc

Контролирует содержимое
окружения OWIN

Вызывает конвейер с тестируемым промежуточным ПО

Этот тест, но сути, выполняет следующий шаблон.

1. Формирует окружение OWIN, соответствующее тестируемому сценарию. Для этого используются вспомогательные типы из библиотеки LibOwin.
2. Создает небольшой конвейер OWIN, состоящий из тестируемого промежуточного ПО с носледующим холостым промежуточным ПО. Результат представляет собой AppFunc, которую можно вызвать, передав ей окружение OWIN.
3. Вызывает конвейер, передавая ему окружение OWIN, сформированное в начале тестирования.
4. Контролирует содержимое окружения OWIN и другой код, с которым работает тестируемое промежуточное ПО. Если, например, тестируемое промежуточное ПО создает ответ, он должен быть частью окружения, а следовательно, должен быть доступным нсредством объекта OwinContext.

Этот же шаблон можно использовать при тестировании промежуточного ПО на основе классов. Отличается только строка формирования конвейера. В листинге 8.5 она выглядит следующим образом:

```
var pipeline = Middleware.Middleware.Impl(noOp));
```

В тесте же с промежуточным ПО на основе класса в листинге 8.4 — вот так:

```
var pipeline = new ConsoleMiddleware(noOp).Invoke)
```

Обратите внимание на то, что мы тоже передаем холостую AppFunc — noOp — в тестируемое промежуточное ПО.

Чтобы протестировать более длинный конвейер, состоящий из нескольких элементов промежуточного ПО, конвейер формируем передачей ссылки на следующий элемент в предыдущий в виде next так же, как мы передавали в промежуточное ПО функцию noOp. После создания конвейера оставшаяся часть теста аналогична тестированию промежуточного ПО в листинге 8.5. Если, например, необходимо протестировать вместе промежуточное ПО на основе лямбда-выражения и промежуточное ПО на основе класса, то строка кода для формирования конвейера изменится на следующую:

```
var pipeline =
    new ConsoleMiddleware(Middleware.Middleware.Impl(noOp)))
    .Invoke)
```

Мы объединили два элемента промежуточного ПО, сформировав короткий конвейер и получив в результате `AppFunc`, которую затем можно протестировать, вызвав ее с окружением OWIN в качестве аргумента.

Теперь вы знаете, как писать и тестировать промежуточное ПО OWIN, и готовы к созданию в следующих главах промежуточного ПО, нацеленного на воплощение важной сквозной функциональности.

8.5. Резюме

- ❑ Некоторые элементы функциональности являются сквозными для нескольких или даже всех микросервисов в системе. Повторное использование предназначенного для воплощения этой функциональности кода может сэкономить немало усилий.
- ❑ Повторное применение кода, нацеленного на реализацию элементов сквозной функциональности, гарантирует единообразие микросервисов. Ведь единообразие важно для сквозной функциональности — для например, журналирования запросов и мониторинга, — делающей микросервисы «законопослушными гражданами» в производственной среде.
- ❑ OWIN обеспечивает аккуратный и гибкий способ формирования конвейеров из промежуточного ПО.
- ❑ Промежуточное ПО OWIN четко отделено от бизнес-логики приложения.
- ❑ Промежуточное ПО OWIN пригодно для повторного использования во многих микросервисах.
- ❑ Промежуточное ПО OWIN может быть реализовано в виде лямбда-функции.
- ❑ Эта лямбда-функция реализует сигнатуру `MidFunc`.
- ❑ Промежуточное ПО OWIN может быть реализовано с помощью класса. Этот класс получает в качестве аргумента конструктора `AppFunc` и содержит метод `Invoke`, реализующий другую `AppFunc`. При использовании промежуточного ПО на базе класса `MidFunc` формируется из конструктора и метода `Invoke`.
- ❑ `LibOwin` — библиотека, предоставляющая удобные типы, облегчающие работу с OWIN. Например, в `LibOwin` есть тип `OwinContext` — строго типизированный адаптер для словаря окружения OWIN.
- ❑ Тестирование промежуточного ПО OWIN не представляет сложностей: его можно вызвать, передав ему окружение OWIN, подождать возвращения ответа и проверить содержимое окружения.
- ❑ Тестирование конвейеров OWIN не представляет сложностей. Аналогично отдельным элементам промежуточного ПО можно вызвать их со словарем OWIN в качестве аргумента и проконтролировать окружение после создания конвейера.

9

Сквозная функциональность: мониторинг и журналирование

В этой главе:

- ❑ мониторинг в системе микросервисов;
- ❑ структурное журналирование и библиотека журналирования Serilog;
- ❑ добавление маркеров корреляции в журнальные сообщения;
- ❑ журналирование запросов и показателей производительности запросов;
- ❑ журналирование необработанных исключений.

В этой главе мы начнем применять полученные в предыдущей главе знания стандарта OWIN для создания новаторно используемых элементов промежуточного ПО OWIN, нацеленных на решение таких важных задач сквозной функциональности, как мониторинг и журналирование. И то и другое необходимо во всех микросервисах и играет важную роль в обеспечении удобства эксплуатации системы микросервисов. При эксплуатации в производственной среде системы необходимо знать, все ли микросервисы работают, а значит, требуется выполнять их мониторинг. Кроме того, как говорилось в главе 6, для диагностики системы необходимо журналирование.

В этой главе мы будем создавать промежуточное ПО в контексте одного микросервиса. Далее, в главе 11, поместим это ПО в пакеты NuGet, что даст возможность повторно использовать его во всех наших микросервисах.

9.1. Мониторинг микросервисов

При внедрении любой серверной системы в производственную среду необходима возможность проверки состояния системы. Нужно знать, работает ли система, происходят ли отказы или ошибки, а также достигает ли производительность своего обычного уровня. Это справедливо для любой системы. При традиционной монолитной системе реализуется мониторинг и в систему встраивается журналирование так, как показано на рис. 9.1. Обычно журналирование выполняется во многих местах базы данных кода — везде, где происходит что-либо важное, требующее регистрации, — а сообщения сохраняются в базе данных.

С системой микросервисов дела обстоят аналогично. Точно так же насущно необходимо отслеживать состояние системы в смысле доступности, производительности, пропускной способности и частоты возникновения ошибок. Отличие заключается

в том, что система микросервисов состоит из множества выполняемых и развертываемых отдельно маленьких составных частей и необходимо контролировать функционирование их всех. Судя по рис. 9.2, эта задача непростая; но на самом деле это не так, если создать инфраструктуру, которая облегчала бы мониторинг микросервисов. Очень скоро мы этим займемся.

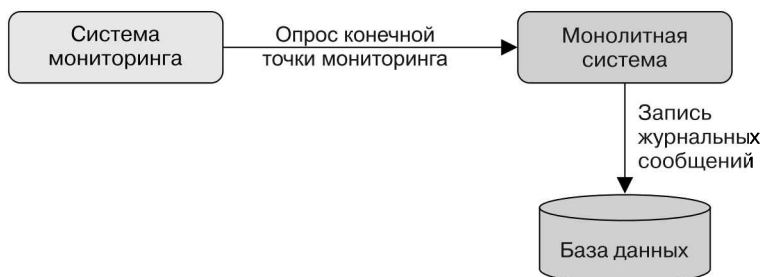


Рис. 9.1. Обычно мониторинг осуществляется вне системы, а журналирование добавляется в системный код с записью сообщений в базу данных

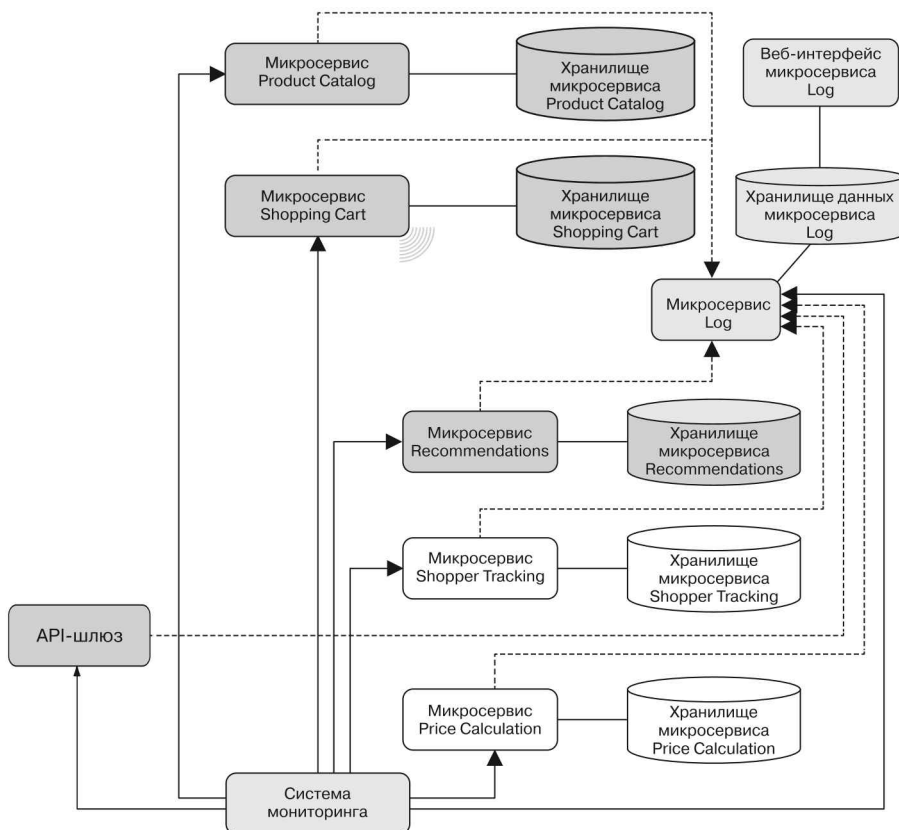


Рис. 9.2. Мониторинг всех микросервисов осуществляется опросом конечных точек

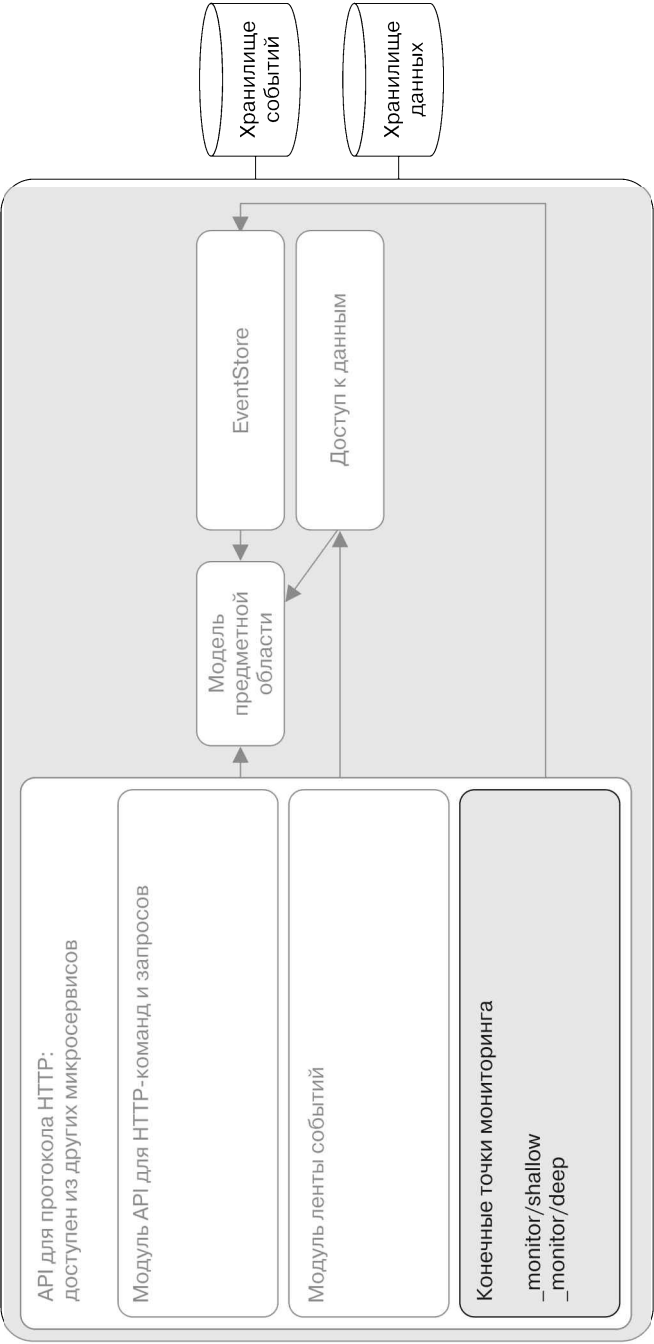


Рис. 9.3. У каждого микросервиса должны быть две конечные точки мониторинга: одна по адресу `/_monitor/shallow`, а вторая — по адресу `/_monitor/deep`. Система мониторинга опрашивает обе. Если обе возвращают ответ с успешным кодом состояния, контролирующая программа считает, что микросервис функционирует нормально

Чтобы контролировать функционирование микросервиса, необходимо добавить две конечные точки, которые будет опрашивать система мониторинга. Считается: если микросервис успешно отвечает на опрос обеих конечных точек, то он работает нормально. На рис. 9.3 показан микросервис с двумя добавленными конечными точками для мониторинга: одна по адресу `/_monitor/shallow`¹, а вторая — по адресу `/_monitor/deep`².

Первая конечная точка, по адресу `/_monitor/shallow`, ничего не делает, кроме возврата в случае успешного выполнения на все запросы ответа с кодом состояния 204 No Content (204 Нет содержимого):

HTTP/1.1 204 No Content

Если же эта конечная точка не возвращает ожидаемый ответ, то микросервис, вероятнее всего, вообще не работает.

Вторая конечная точка, по адресу `/_monitor/deep`, проверяет внутреннее состояние микросервиса и, если все в порядке, также возвращает ответ с кодом состояния 204 No Content (Нет содержимого). Детали проверки внутреннего состояния различаются от микросервиса к микросервису, но обычно выполняется простой запрос к базе данных микросервиса и проверяется, имеет ли смысл результат.

Обе конечные точки мониторинга можно реализовать в промежуточном ПО OWIN, в результате чего получится конвейер OWIN (рис. 9.4).

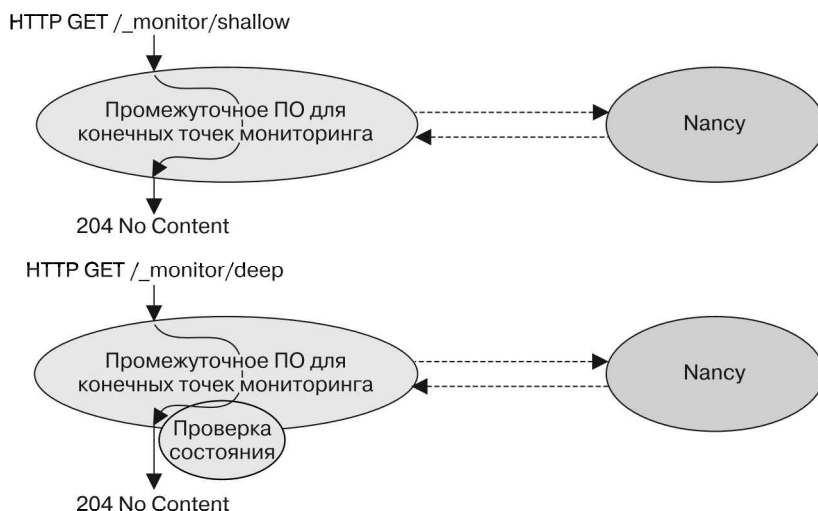


Рис. 9.4. Конечные точки мониторинга можно поместить в элемент промежуточного ПО OWIN перед Nancy. При поступлении запроса к конечной точке `/_monitor/shallow` промежуточное ПО для мониторинга возвращает ответ 204 No Content (204 Нет содержимого). При поступлении запроса к конечной точке `/_monitor/deep` промежуточное ПО мониторинга вызывает функцию, проверяющую состояние микросервиса и в случае успешной проверки возвращающую ответ 204 No Content (204 Нет содержимого). Остальные запросы проходят через промежуточное ПО для мониторинга и поступают в Nancy

¹ От англ. shallow — «поверхностный». — *Примеч. пер.*

² От англ. deep — «углубленный». — *Примеч. пер.*

Мы реализуем конечные точки мониторинга в промежуточном ПО OWIN, поскольку это дает возможность отделить функциональность мониторинга от бизнес-логики микросервиса. Более того, поскольку мы помещаем промежуточное ПО в пакет NuGet и повторно используем его в нескольких микросервисах, желательно, чтобы у него было как можно меньше зависимостей, а следовательно, оно не навязывало бы выбор технологий использующим его микросервисам. Далее в этой главе мы реализуем промежуточное ПО для мониторинга и рассмотрим, как сделать так, чтобы каждый микросервис обеспечивал свою собственную реализацию выполняемой в конечной точке `/_monitor/deep` проверки состояния.

9.2. Журналирование микросервисов

Помимо мониторинга каждого из микросервисов, необходимо отправлять сообщения по поводу отказов, ошибок, производительности и всего остального, что только может понадобиться. Как обсуждалось в главе 6, можно включить в систему централизованный микросервис Log, получающий журнальные сообщения от всех остальных микросервисов, сохраняющий их (например, в поисковой системе) и предоставляющий удобный доступ к ним посредством информационных панелей и поисковых пользовательских интерфейсов (рис. 9.5). Хотелось бы иметь возможность легко настраивать любой микросервис на отправку журнальных сообщений микросервису Log.

А еще хотелось бы, чтобы все микросервисы выполняли хотя бы минимальное журналирование. Все микросервисы должны журналировать HTTP-запросы, HTTP-ответы, а также время обработки каждого запроса. Эти журналы позволяют получить представление о рабочем состоянии микросервиса.

- ❑ Журнал HTTP-запросов предоставляет ценнейшую информацию о происходящем в производственной среде при выполнении отладки приложения. Хотя одних журналов запросов для отладки недостаточно. Бизнес-логика микросервиса должна также отправлять в микросервис Log журнальные сообщения обо всех нештатных происшествиях.
- ❑ Записанные в журнал HTTP-ответы показывают, не происходит ли отказ микросервиса. В частности, микросервис, часто возвращающий ответы с кодами состояния в диапазоне 5xx, используемом для ошибок сервера, например необработанных исключений, вероятнее всего, находится под стрессовой нагрузкой или содержит ошибки.
- ❑ Занесенное в журнал время обработки запросов можно использовать в качестве эталона производительности микросервиса. После установления эталона можно сравнивать с ним текущие значения времени обработки запросов. Если они существенно выросли, то микросервис, вероятно, находится под стрессовой нагрузкой.

Для выполнения журналирования запросов и показателей производительности мы воспользуемся промежуточным ПО OWIN. В результате у каждого микросервиса будет собственный конвейер OWIN, подобный показанному на рис. 9.6.

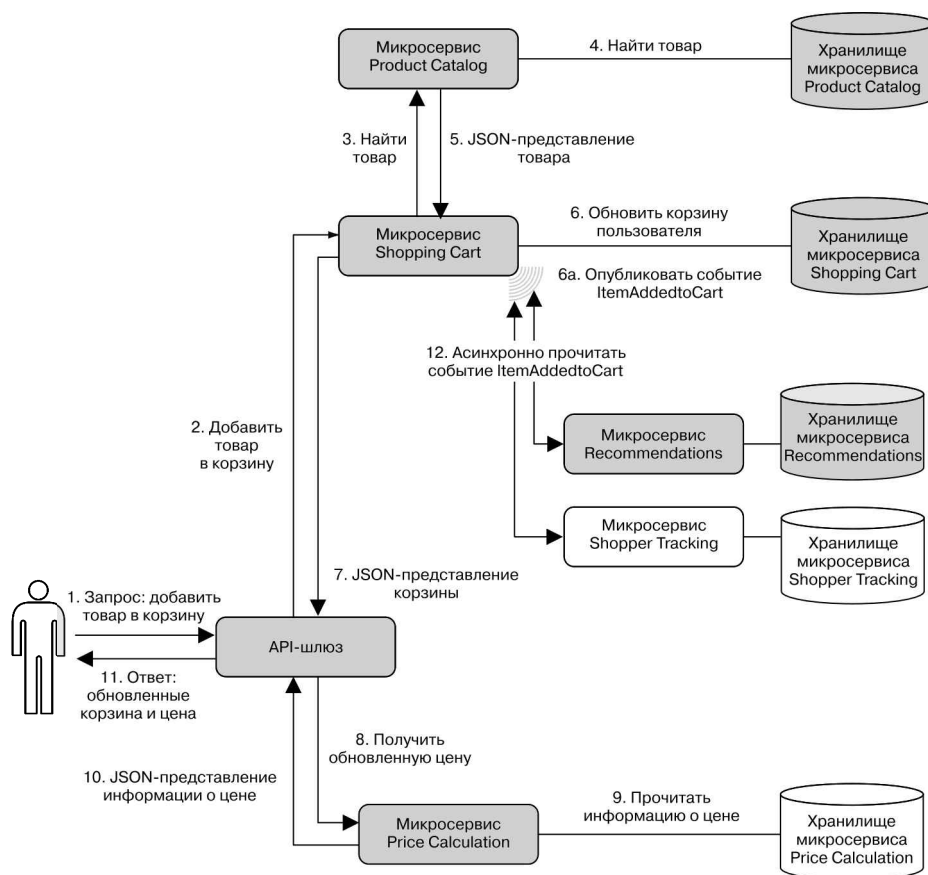


Рис. 9.5. Все микросервисы отправляют журнальные сообщения в централизованный микросервис Log

Конвейер на рис. 9.6 включает промежуточное ПО для мониторинга, журналирования запросов и журналирования показателей производительности. Перед ними располагается элемент промежуточного ПО, отвечающий за вставку во все журнальные сообщения маркеров корреляции. Хотя все элементы промежуточного ПО автономны и не зависят от остального промежуточного ПО, порядок их расположения в конвейере существенен. На рис. 9.6 первым идет промежуточное ПО для вставки маркеров корреляции, чтобы журнальные сообщения в последующих элементах промежуточного ПО содержали эти маркеры. Аналогично промежуточному ПО для мониторинга предшествуют два элемента промежуточного ПО для журналирования, чтобы журналировать выполняемые к конечным точкам мониторинга запросы. Если требуется, чтобы система вела себя по-другому, можно переставить элементы конвейера: например, если нам не нужно журналировать выполняемые к конечным точкам мониторинга запросы, можно переместить мониторинг конечных точек ближе к началу конвейера.

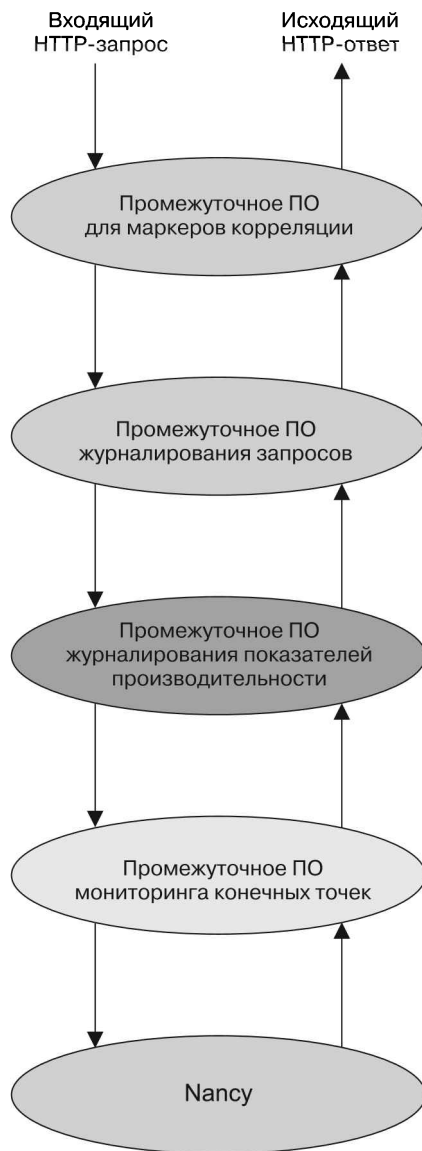


Рис. 9.6. Формируем конвейер OWIN из промежуточного ПО для добавления маркеров корреляции, журналирования и мониторинга. В конце конвейера располагается Nancy

Благодаря конвейеру эксплуатация данного микросервиса становится гораздо более приятной: можно легко контролировать его функционирование, а с помощью журналов получить общее представление о его рабочем состоянии. В главе 11 мы займемся созданием накетов NuGet, содержащих элементы промежуточного ПО OWIN, чтобы упростить снабжение новых микросервисов стандартными конечными точками для мониторинга и базовым журналированием.

Структурное журналирование с помощью Serilog

Обычные библиотеки журналирования рассматривают журнальные сообщения как простые строки, возможно, с присоединенным к ним исключением. Но мы воспользуемся более интересной возможностью — библиотекой структурного журналирования (structured logging) Serilog. Идея структурного журналирования заключается в том, чтобы журнальные сообщения могли содержать структурированные данные. То есть журнальное сообщение может содержать не только текстовое сообщение, например, «что-то пошло не так», но и объекты.

Библиотека Serilog предоставляет дополнительный синтаксис — надстройку над строками формата .NET — для журналирования сообщений. Этот дополнительный синтаксис дает возможность назначать параметрам имена, а также выбирать, будет ли значение параметра преобразовано в строку с помощью вызова метода `.ToString()`, или в журнальное сообщение будет включен объект целиком. Именованные параметры заключаются в фигурные скобки, например, `"{RequestTime}"`. Для указания библиотеке Serilog на необходимость включить в сообщение значение целиком нужно добавить символ `@` перед именем: `"{@RequestTime}"`. Как и в случае строк формата .NET, к параметрам без предшествующего символа `@` можно добавлять директивы форматирования.

Например, можно воспользоваться библиотекой Serilog для отправки журнального сообщения следующим образом (здесь `log` — объект типа `Ilogger` библиотеки Serilog):

```
var simplifiedRequest = new
{
    Path = "/foo",
    Method = "GET",
    Protocol = "HTTP",
};

var requestTime = 200;
log.Information(
    "Processed request: {@Request} in {RequestTime:000} ms.",
    simplifiedRequest,
    requestTime);
```

`simplifiedRequest` и `requestTime` — включенные в сообщение объекты. Как и другие фреймворки журналирования, Serilog предоставляет возможность настройки различных *получателей данных* (*sinks*) для записи журнальных сообщений. В этом случае в качестве получателя данных мы используем консоль. При записи предыдущих журнальных сообщений в консоль объекты `simplifiedRequest` и `requestTime` сериализуются в формат JSON и вставляются в строку журнального сообщения. Serilog добавляет туда и некоторые метаданные, так что записанное в консоль сообщение выглядит следующим образом:

```
09:14:22 [Information] Processed request { Path: "/foo", Method: "GET",
➡ Protocol: "HTTP" } in 200 ms.
```

Как видите, никакие данные из исходного журнального сообщения не пропали, поскольку объекты `simplifiedRequest` и `requestTime` целиком включены в записанное в консоль сообщение. Это удобно, но еще удобнее использовать в качестве получателя данных поисковую систему, например Elasticsearch. При этом включаемые

в журнальные сообщения объекты, к примеру `simplifiedRequest`, сохраняются и индексируются в поисковой системе. Это дает возможность искать журнальные сообщения по значениям конкретных свойств включенных в сообщение объектов. Так, можно найти журнальные сообщения с объектом `Request`, значение свойства `Path` которого равно `/foo`. Предыдущее сообщение при этом будет найдено, а сообщение по поводу запроса к URL `/bar` — нет.

Сохранение структуры включаемых в журнальное сообщение данных означает возможность его представления клиентской части с веб-интерфейсом микросервиса `Log` в структурированном виде. В результате упрощаются поиск в журналах, их просмотр и обзор при необходимости, они включают больше данных. Одна из подобных клиентских частей с веб-интерфейсом — `Kibana`, отлично работающая с хранимыми в `Elasticsearch` журналами и предоставляющая массу возможностей для поиска и визуализации.

Как вы увидите далее, при реализации промежуточного ПО журналирования `Serilog` можно настроить так, чтобы в каждое журнальное сообщение включались дополнительные данные. Мы воспользуемся этой возможностью для добавления во все журнальные сообщения маркеров корреляции.

В оставшейся части этой главы реализуем мониторинг и журналирование. Нам нужно сделать это во всех микросервисах, так что можем работать в контексте любого из уже реализованных микросервисов. В качестве примера возьмем микросервис `Shopping Cart` и будем основываться на коде из главы 5, но наш код подходит и для остальных микросервисов.

9.3. Реализация промежуточного ПО для мониторинга

В этом разделе создадим элемент промежуточного ПО, реализующий две обсуждавшиеся ранее конечные точки. Они будут функционировать следующим образом:

- ❑ `/_monitor/shallow` возвращает ответ на все запросы с кодом состояния `204 No Content` (204 Нет содержимого);
- ❑ `/_monitor/deep` выполняет первичную проверку внутреннего состояния микросервиса. Если все в порядке, также возвращает ответ с кодом состояния `204 No Content` (204 Нет содержимого). В ином случае возвращает ответ с кодом состояния `503 Service Unavailable` (503 Сервис недоступен). При проверке выполняет запрос к базе данных микросервиса `Shopping Cart`, чтобы определить количество имеющихся там корзин заказов. Если это количество выше определенного порогового значения, проверка состояния считается пройденной успешно, если ниже — нет. Подобную проверку нужно выполнять с учетом специфики системы: конкретно эта проверка имеет смысл, если объем трафика интернет-магазина достаточен для того, чтобы число активных корзин заказов всегда превышало пороговое значение. Со временем вы научитесь лучше понимать, как работает система и как она выглядит в полностью рабочем состоянии. Эту информацию нужно использовать для нериодической проверки состояния системы.

План реализации конечной точки мониторинга выглядит следующим образом.

1. Добавить библиотеку LibOwin в микросервис Shopping Cart.
2. Добавить в микросервис новый файл `MonitoringMiddleware.cs`.
3. Создать элемент промежуточного ПО, реализующий конечную точку `/_monitor/shallow`.
4. Добавить в промежуточное ПО для мониторинга реализацию конечной точки `/_monitor/deep`.
5. Создать проверку состояния для микросервиса Shopping Cart.
6. Добавить промежуточное ПО для мониторинга в конвейер микросервиса Shopping Cart.

Что ж, приступим.

Реализация конечной точки для поверхностного мониторинга

Прежде всего необходимо добавить в микросервис Shopping Cart библиотеку LibOwin, как описывалось в главе 8.

Далее добавьте в Shopping Cart файл `MonitoringMiddleware.cs`. Вставьте в него следующий код, чтобы реализовать конечную точку `/_monitor/shallow` в виде элемента промежуточного ПО OWIN (листинг 9.1).

Листинг 9.1. Промежуточное ПО OWIN, реализующее конечную точку для поверхностного мониторинга

```
namespace ShoppingCart.Infrastructure
{
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using LibOwin;

    using AppFunc = 
        System.Func<System.Collections.Generic.IDictionary<string, object>,
        System.Threading.Tasks.Task>;

    public class MonitoringMiddleware
    {
        private AppFunc next;

        public MonitoringMiddleware(AppFunc next)
        {
            this.next = next; 
        }

        public Task Invoke(IDictionary<string, object> env)
        {
            var context = new OwinContext(env);
            if (context.Request.Path.Equals("/_monitor/shallow")) 
                return ShallowEndpoint(context);
        }
    }
}
```

Сигнатура OWIN AppFunc, обсуждавшаяся подробнее в главе 8

Ссылка на оставшуюся часть конвейера

Проверяем, предназначен ли входящий запрос для конечной точки `/_monitor/shallow`

```

    else
        return this.next(env);
    }

    private Task ShallowEndpoint(OwinContext context)
    {
        context.Response.StatusCode = 204;
        return Task.FromResult(0);
    }
}

```

Если запрос не предназначен для конечной точки мониторинга, вызываем оставшуюся часть конвейера

Устанавливаем значение кода состояния в 204 и прерываем выполнение конвейера

Этот код реализует элемент промежуточного ПО OWIN, отвечающий за конечную точку для поверхностного мониторинга. Это ПО реализовано в виде класса, чему мы научились в главе 8. Для упрощения работы с окружением OWIN мы воспользовались типом `OwinContext` из библиотеки `LibOwin`.

Реализация конечной точки для углубленного мониторинга

Расширим элемент промежуточного ПО, который реализует конечную точку для поверхностного мониторинга, так, чтобы он реализовывал и вторую конечную точку для мониторинга — `/_monitor/deep` (листинг 9.2). Для этого допустим, что выполняющая проверку состояния функция внедряется в объект `MonitoringMiddleware` при его создании. Проверка состояния должна быть функцией без аргументов, возвращающей значение типа `bool`, которое отражает, была ли проверка успешной. Необходимо, чтобы проверка состояния была асинхронной, так что вместо `bool` она будет возвращать `Task<bool>`.

Чтобы функция проверки состояния могла быть внедрена, добавим параметр в конструктор класса `MonitoringMiddleware`. Добавим также закрытое поле для хранения функции проверки состояния.

Листинг 9.2. Промежуточное ПО для мониторинга с внедренной проверкой состояния

```

public class MonitoringMiddleware
{
    private AppFunc next;
    private Func<Task<bool>> healthCheck;

    public MonitoringMiddleware(AppFunc next, Func<Task<bool>> healthCheck)
    {
        this.next = next;
        this.healthCheck = healthCheck;
    }
    ...
}

```

После добавления этой функции проверки состояния в класс `MonitoringMiddleware` можно приступить к реализации конечной точки `/_monitor/deep`. Мы мно-

го переделаем метод `Invoke` из класса `MonitoringMiddleware` так, чтобы он искал все пути, начинающиеся с `/_monitor`, вместо начинающихся только с `/_monitor/shallow`. Вторая проверка пути отделяет конечную точку `/_monitor/shallow` от `/_monitor/deep` (листинг 9.3).

Листинг 9.3. Конечные точки мониторинга и вызов метода `healthCheck`

```
private static readonly PathString monitorPath =
    new PathString("/_monitor");
private static readonly PathString monitorShallowPath =
    new PathString("/_monitor/shallow");
private static readonly PathString monitorDeepPath =
    new PathString("/_monitor/deep");

public Task Invoke(IDictionary<string, object> env)
{
    var context = new OwinContext(env);
    if (context.Request.Path.StartsWithSegments(monitorPath))
        return HandleMonitorEndpoint(context);
    else
        return this.next(env);
}

private Task HandleMonitorEndpoint(OwinContext context)
{
    if (context.Request.Path.StartsWithSegments(monitorShallowPath))
        return ShallowEndpoint(context);
    else if (context.Request.Path.StartsWithSegments(monitorDeepPath))
        return DeepEndpoint(context);
    return Task.FromResult(0);
}

private async Task DeepEndpoint(OwinContext context)
{
    if (await this.healthCheck())
        context.Response.StatusCode = 204;
    else
        context.Response.StatusCode = 503;
}

private Task ShallowEndpoint(OwinContext context)
{
    context.Response.StatusCode = 204;
    return Task.FromResult(0);
}
```

В этой ветке кода обрабатываются все запросы к конечным точкам мониторинга

Все остальные запросы передаются на обработку оставшейся части конвейера

Возвращает Task, даже несмотря на то, что явным образом ничего не возвращается, поскольку метод асинхронен

Проверка состояния также может быть асинхронной, так что приходится ждать ее результата

Теперь у нас есть элемент промежуточного ПО, реализующий обе конечные точки для мониторинга и не зависящий от специфики микросервиса `Shopping Cart`. Реализация функции проверки состояния, конечно, отражает специфику `Shopping Cart`, но мы предусмотрительно вынесли ее за пределы этого промежуточного ПО.

Создав промежуточное ПО, приступим к реализации проверки состояния. После этого добавим элемент `MonitoringMiddleware` в конвейер OWIN микросервиса.

Как уже упоминалось, функция проверки состояния запрашивает в базе данных микросервиса Shopping Cart количество корзин заказов и сравнивает его с пороговым значением. Напомню, что микросервис использует SQL базу данных, для выполнения запроса к которой мы применяем Dapper. Следующий код (листинг 9.4) с помощью Dapper выполняет проверку состояния.

Листинг 9.4. Проверка состояния на базе количества корзин заказов

```
private const string connectionString =
    @"Data Source=.\SQLEXPRESS;
    Initial Catalog=ShoppingCart;
    Integrated Security=True";

private readonly int threshold = 1000;

public async Task<bool> HealthCheck()
{
    using (var conn = new SqlConnection(connectionString))
    {
        var count =
            (await conn.QueryAsync<int>("select count(ID) from ShoppingCart"))
            .Single();
        return count > this.threshold;
    }
}
```

Строка соединения с базой данных микросервиса ShoppingCart

Запрос количества корзин заказов из базы данных

Выясняем, что показал запрос: нормально функционирует микросервис или нет

В этом коде заключена определенная, относящаяся именно к микросервису Shopping Cart функциональность: простая операция для проверки того, нормально ли функционирует микросервис. Эта операция проверяет наличие у микросервиса Shopping Cart доступа к его базе данных, а также наличие в этой базе данных корзин заказов. Она не охватывает все возможные варианты отказов, но служит довольно хорошим показателем состояния микросервиса.

Добавление промежуточного ПО для мониторинга в конвейер OWIN

Нам осталось всего лишь внедрить проверку состояния в `MonitoringMiddleware` и добавить это промежуточное ПО в конвейер OWIN. И то и другое реализуем в классе `Startup`.

До сих пор мы использовали класс `Startup` только для добавления в конвейер фреймворка Nancy. Теперь вставим в него метод для проверки состояния и расширим конвейер, в результате чего получим листинг 9.5.

Листинг 9.5. Добавление промежуточного ПО `MonitoringMiddleware` в конвейер OWIN в классе `Startup`

```
namespace ShoppingCart
{
    using System.Data.SqlClient;
    using System.Threading.Tasks;
```

```

using Microsoft.AspNet.Builder;
using System.Linq;
using Dapper;
using Nancy.Owin;
using global::ShoppingCart.Infrastructure;

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseOwin(buildFunc =>
        {
            buildFunc(next =>
                new MonitoringMiddleware(next, HealthCheck).Invoke();
            buildFunc.UseNancy();
        });
    }

    private const string connectionString =
        @"Data Source=.\SQLEXPRESS;
        Initial Catalog=ShoppingCart;
        Integrated Security=True";
    private readonly int threshold = 1000;

    public async Task<bool> HealthCheck()
    {
        using (var conn = new SqlConnection(connectionString))
        {
            var count =
                (await conn.QueryAsync<int>("select count(ID) from ShoppingCart"))
                .Single();
            return count > this.threshold;
        }
    }
}

```

Добавляем MonitoringMiddleware в конвейер OWIN

Та же проверка состояния, которую вы встречали ранее

После этого можно запустить микросервис Shopping Cart с помощью утилиты `dotnet` и протестировать конечные точки мониторинга. Запрос к конечной точке `/_monitor/shallow` выглядит следующим образом:

```

GET /_monitor/shallow HTTP/1.1
Host: localhost:5000
Accept: application/json

```

Ответ об успешной проверке — вот так:

```
HTTP/1.1 204 No Content
```

А вот запрос к конечной точке `/_monitor/deep`:

```

GET /_monitor/deep HTTP/1.1
Host: localhost:5000
Accept: application/json

```

Ответ об успешной проверке:

HTTP/1.1 204 No Content

И наконец, ответ о неудачной проверке:

HTTP/1.1 503 Service Unavailable

На этом реализация промежуточного ПО для мониторинга завершена. В силу того что только проверка состояния, реализованная вне промежуточного ПО, связана с особенностями микросервиса Shopping Cart, само промежуточное ПО хорошо подходит для новаторного использования в других микросервисах.

9.4. Реализация промежуточного ПО для журналирования

Далее реализуем три элемента промежуточного ПО, связанных с журналированием:

- ❑ обеспечивающий включение во все журнальные сообщения маркеров корреляции. Наряду с этим позаботимся, чтобы маркеры корреляции содержались во всех исходящих HTTP-запросах;
- ❑ журналирующий все запросы и ответы;
- ❑ измеряющий длительность обработки всех запросов с журналированием результатов.

Как упоминалось ранее, для записи журнальных сообщений мы воспользуемся библиотекой Serilog. Мы настроим ее так, чтобы журнальные сообщения выводились в консоль, но она поддерживает записи во множество различных мест, включая поисковую систему Elasticsearch, которую мы будем использовать для отправки сообщений микросервису Log.

В этом и следующем разделах вы узнаете, как выполнить неречисленные далее действия.

1. Установить и настроить библиотеку Serilog, а также создать регистратор.
2. Написать элемент промежуточного ПО для включения во все журнальные сообщения маркеров корреляции. Если входящий запрос содержит в заголовке маркер корреляции, промежуточное ПО будет использовать его, если нет, то создаст новый маркер и воспользуется им.
3. Написать необходимый конфигурационный и фабричный код, чтобы гарантировать наличие маркеров корреляции в исходящих запросах.
4. Написать элемент промежуточного ПО для журналирования всех запросов и ответов.
5. Написать элемент промежуточного ПО для измерения и журналирования длительности обработки запросов.
6. Добавить все три элемента промежуточного ПО в конвейер OWIN в микросервисе Shopping Cart.

Прежде всего нужно установить пакеты Serilog и Serilog.Sinks.ColoredConsole системы управления пакетами NuGet. Для этого достаточно добавить их в файл `project.json`:

```
"dependencies": {
  "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
  "Microsoft.AspNetCore.Owin": "1.0.0",
  "Nancy": "2.0.0-barneyrubble",
  "Polly": "4.2.1",
  "Dapper": "1.50.0-rc2a",
  "LibOwin": "1.0.0",
  "Serilog": "2.0.0-rc-600",
  "Serilog.Sinks.ColoredConsole": "2.0.0-beta-1001"
}
```

Теперь перейдите в файл `Startup.cs` для настройки Serilog и создания объекта-регистратора. Объект-регистратор представляет собой вонлощение `ILogger` — интерфейса из библиотеки Serilog — и используется для отправки журнальных сообщений. Добавьте этот метод в класс `Startup` (листинг 9.6).

Листинг 9.6. Настройка и создание регистратора Serilog

```
private ILogger ConfigureLogger()
{
    return new LoggerConfiguration()
        .Enrich.FromLogContext()
        .WriteTo.ColoredConsole(
            LogEventLevel.Verbose,
            "{NewLine}{Timestamp:HH:mm:ss} [{Level}]
            ➔ ({CorrelationToken}) {Message}{NewLine}{Exception}")
        .CreateLogger();
}
```

К контексту журнала будет добавлен маркер корреляции. Это приведет к пополнению им всех журнальных сообщений

Настройка журналирования в консоль в формате, включающем маркер корреляции

Тип `LoggerConfiguration` — точка входа в текущий API для конфигурации Serilog. В нем мы настроили Serilog на вывод в консоль и включение в выводимую информацию маркера корреляции. Библиотека Serilog предоставляет возможность одновременно выводить журнальные сообщения в различные места, для этого необходимо добавить дополнительные блоки `WriteTo`. Serilog также дает возможность пополнять *информацией* (*enrich*) журнальные сообщения, то есть придавать им дополнительные свойства. Пополнение можно использовать для добавления имени сервера, названия среды (например, QA или производственной эксплуатации), роли иницировавшего запрос пользователя или любой другой информации, которую вы хотели бы вставить в журнальные сообщения, недоступной в том месте кода, где происходит запись сообщений. В данном случае мы используем пополнение для добавления маркера корреляции. Добавляем маркер в *контекст журнала*, представляющий собой функцию библиотеки Serilog, предназначенную для добавления

журнальных свойств ко всем элементам определенного контекста. Чуть позже мы обсудим подробнее, что это значит.

Мы вызываем метод `ConfigureLogger` в методе `Configure` класса `Startup`. После создания всех трех элементов промежуточного ПО разместим регистратор в нужный. Помните, что платформа ASP.NET Core вызывает метод `Configure` при загрузке. Пока он выглядит следующим образом (листинг 9.7).

Листинг 9.7. Создание регистратора в методе `Configure` путем вызова `ConfigureLogger`

```
public void Configure(IApplicationBuilder app)
{
    var log = ConfigureLogger();

    app.UseOwin(buildFunc =>
    {
        buildFunc(next => new MonitoringMiddleware(next, HealthCheck).Invoke);
        buildFunc.UseNancy();
    });
}
```

Теперь, создав и настроив регистратор, добавим в контекст журнала маркер корреляции.

Добавление маркеров корреляции во все журнальные сообщения

Как вы помните из главы 6, ценность журнальных сообщений повышается, если они позволяют отслеживать движение запроса пользователя по системе микросервисов. Пользовательский запрос, вероятно, будет обрабатываться не одним микросервисом. Скорее всего, для его выполнения понадобится совместная работа нескольких микросервисов. Возможность отслеживать движение пользовательских запросов по всем этим взаимодействующим микросервисам приносит немалую пользу, и обеспечить ее может как раз маркер корреляции. Маркер корреляции представляет собой GUID, присоединяемый к запросу, а затем передаваемый всем взаимодействующим во время его обработки микросервисам при их вызове. Маркер корреляции передается от одного микросервиса другому в пользовательском заголовке `Correlation-Token` HTTP-запроса:

```
GET /shoppingcart/42 HTTP/1.1
```

```
Host: localhost:5000
```

```
Accept: application/json
```

```
Correlation-Token: 600580e6-90b6-47e7-a054-1f5d2731135f
```

Содержит
маркер
корреляции

Элемент промежуточного ПО, который мы собираемся написать, будет искать заголовок `Correlation-Token` запроса и в случае обнаружения добавлять его в контекст журнала. В противном случае он будет создавать новый маркер и добавлять в контекст журнала его. Взглянем на это промежуточное ПО, а затем обсудим, что представляет собой контекст журнала.

Добавьте новый файл `LoggingMiddleware.cs` в микросервис `Shopping Cart`. Мы реализуем в нем все три элемента промежуточного ПО, начиная со следующего, который добавляет маркер корреляции в контекст журнала `Serilog` (листинг 9.8).

Листинг 9.8. Промежуточное ПО OWIN, реализованное в виде лямбды

```
namespace ShoppingCart.Infrastructure
{
    using System;
    using LibOwin;
    using Serilog;
    using Serilog.Context;

    public class CorrelationToken
    {
        {
            public static AppFunc Middleware(AppFunc next)
            {
                return async env =>
                {
                    Guid correlationToken;
                    var owinContext = new OwinContext(env);
                    if (!owinContext.Request.Headers["Correlation-Token"] != null
                        && Guid.TryParse(owinContext.Request.Headers["Correlation-
                            Token"],
                            out correlationToken))
                    {
                        correlationToken = Guid.NewGuid();
                        owinContext.Set("correlationToken", correlationToken.ToString());
                        using (LogContext.PushProperty("CorrelationToken", correlationToken))
                        {
                            await next(env);
                        }
                    }
                };
            }
        }
    }
}
```

Сохраняем маркер корреляции для дальнейшего использования

Пытаемся найти маркер корреляции в заголовке запроса

Добавляем маркер корреляции в контекст журнала

Это промежуточное ПО OWIN, реализованное в лямбда-стиле. Оно выполняет попытку чтения заголовка `Correlation-Token` запроса из окружения OWIN. При нахождении такого со значением GUID оно использует этот GUID в качестве маркера корреляции. Данное промежуточное ПО добавляет маркер корреляции в контекст журнала библиотеки `Serilog`. Созданный регистратор можно настроить так, чтобы он использовал контекст журнала для пополнения всех журнальных сообщений информацией. Именно так мы и настраивали регистратор, а значит, все отправляемые нами журнальные сообщения будут пополнены свойствами контекста. Маркер корреляции содержится в контексте журнала для всего выполняемого после оператора `using` кода, то есть всего, что выполняется в строке `await next(env)`, включая оставшуюся часть конвейера OWIN, любое промежуточное ПО, добавленное ниже промежуточного ПО маркера корреляции, а также `Nancy`. В общем, это соответствует всем журнальным сообщениям, отправляемым в этих элементах промежуточного ПО, модулях `Nancy` и еще глубже внутри кода микросервиса `Shopping Cart`.

Промежуточное ПО маркера корреляции проверяет заголовок `Correlation-Token` до создания нового маркера корреляции. После создания маркера корреляции его необходимо присоединить ко всем запросам к другим микросервисам, выполняемым в результате обработки текущего. Благодаря этому появляется возможность отслеживать операции по всем участвующим в обработке микросервисам. Именно поэтому промежуточное ПО добавляет маркер корреляции в окружение OWIN. Далее необходимо убедиться, что во всех исходящих запросах задан маркер корреляции, для чего мы извлечем маркер корреляции обратно из окружения OWIN.

Добавление маркера корреляции во все исходящие HTTP-запросы

Микросервис `Shopping Cart` иногда выполняет запросы к микросервису `Product Catalog`. При этом в заголовке `Correlation-Token` запроса должен содержаться маркер корреляции. Вспоминаем, что для выполнения HTTP-запросов мы используем объект `HttpClient` и что до сих пор при необходимости выполнения HTTP-запроса создавали новый экземпляр класса `HttpClient`. Немного изменим это поведение: воспользуемся фабрикой для создания объектов `HttpClient`, в которой сделаем так, чтобы `HttpClient` вставлял во все запросы заголовок `Correlation-Token`.

Для создания, настройки и использования фабрики `HttpClient` выполним следующее.

1. Напишем класс `HttpClientFactory`, который умел бы создавать объекты `HttpClient`, сконфигурированные на отправку вместе с каждым запросом соответствующего заголовка `Correlation-Token`.
2. Создадим загрузчик `Nancy`, извлекающий маркер корреляции из окружения OWIN, создающий объект `HttpClientFactory` и регистрирующий его в контейнере внедрения зависимости `Nancy`.
3. Передаем ссылку на фабрику `HttpClientFactory` в ту часть микросервиса `Shopping Cart`, которая отвечает за выполнение HTTP-запроса. Фреймворк `Nancy` сам позаботится о внедрении экземпляра `HttpClientFactory`, созданного нами в загрузчике.

Прежде всего добавьте в микросервис `Shopping Cart` файл `HttpClientFactory.cs`, содержащий следующий код (листинг 9.9).

Листинг 9.9. Создание объектов `HttpClient` с целью отправки маркеров корреляции

```
namespace ShoppingCart
{
    using System;
    using System.Net.Http;

    public interface IHttpClientFactory
    {
        HttpClient Create(Uri uri);
    }

    public class HttpClientFactory : IHttpClientFactory
    {
```

```
private readonly string correlationToken;

public HttpClientFactory(string correlationToken)
{
    this.correlationToken = correlationToken;
}

public HttpClient Create(Uri uri)
{
    var client = new HttpClient() { BaseAddress = uri };
    client
        .DefaultRequestHeaders
        .Add("Correlation-Token", this.correlationToken);
    return client;
}
}
```

Обеспечиваем внедрение маркера корреляции

Добавляем маркер корреляции в каждый запрос

Мы добавили как реализацию, так и интерфейс, так что код, которому нужен `HttpClient`, будет зависеть только от интерфейса.

В качестве следующего шага нам не понадобятся `HttpClientFactory` и регистратор в загрузчике `Nancy`, который будет регистрировать их в контейнере внедрения зависимостей. Добавьте следующий код (листинг 9.10) загрузчика в новый файл `Bootstrapper.cs`.

Листинг 9.10. Загрузчик `Nancy`

```
namespace ShoppingCart
{
    using Nancy;
    using Nancy.Bootstrapper;
    using Nancy.TinyIoc;
    using Serilog;

    public class Bootstrapper : DefaultNancyBootstrapper
    {
        private readonly ILogger log;

        public Bootstrapper(ILogger log)
        {
            this.log = log;
        }

        protected override void ApplicationStartup(
            TinyIoCContainer container, IPipelines pipelines)
        {
            base.ApplicationStartup(container, pipelines);
            container.Register(this.log);
        }

        protected override void RequestStartup(
            TinyIoCContainer container,
            IPipelines pipelines,
            NancyContext context)
        {

```

Внедряет регистратор Serilog

Отмечает регистратор в контейнере Nancy

```

base.RequestStartup(container, pipelines, context);
var correlationToken =
    context.GetOwinEnvironment()["correlationToken"] as string;
container.Register<IHttpClientFactory>(
    new HttpClientFactory(correlationToken));
}
}
}

```

Получаем маркер корреляции из окружения OWIN

Внедряем маркер корреляции в HttpClientFactory, после чего регистрируем эту фабрику в контейнере Nancy

Загрузчик использует метод `RequestStartup` для создания и регистрации объекта `HttpClientFactory` для каждого запроса. Для каждого запроса должен быть свой объект, поскольку у каждого запроса свой маркер корреляции. В промежуточном ПО маркера корреляции выполняется его сохранение в окружении OWIN. Здесь же мы извлекаем его из окружения и передаем `HttpClientFactory`. Мы берем маркер корреляции непосредственно из словаря окружения OWIN, а не получаем его путем создания объекта `OwinContext` с помощью библиотеки `LibOwin`, поскольку `correlationToken` — наш собственный пользовательский ключ, о котором `LibOwin` ничего не знает.

Если запустить микросервис `Shopping Cart` сейчас, то произойдет фатальный сбой, поскольку у загрузчика нет конструктора по умолчанию. Мы создали его с конструктором, принимающим в качестве параметра объект типа `ILogger`, поскольку нам нужно, чтобы можно было использовать везде один и тот же созданный в классе `Startup` объект типа `ILogger`. Именно поэтому мы внедряем его в загрузчик и регистрируем в контейнере `Nancy`. Фреймворк `Nancy` внедрит нужный экземпляр `ILogger` везде, где классу передается ссылка на `ILogger`.

Чтобы микросервис снова заработал, немного измените метод `Configure` в классе `Startup`: создайте загрузчик и передайте его `Nancy` (листинг 9.11).

Листинг 9.11. Передаем `Nancy` загрузчик

```

public void Configure(IApplicationBuilder app)
{
    var log = ConfigureLogger();

    app.UseOwin(buildFunc =>
    {
        buildFunc(next => GlobalErrorLogging.Middleware(next, log));
        buildFunc(next => CorrelationToken.Middleware(next));
        buildFunc(next => RequestLogging.Middleware(next, log));
        buildFunc(next => PerformanceLogging.Middleware(next, log));
        buildFunc(next => new MonitoringMiddleware(next, HealthCheck).Invoke);
        buildFunc.UseNancy(opt => opt.Bootstrapper = new Bootstrapper(log));
    });
}

```

Создаем загрузчик и передаем его Nancy

Теперь микросервис будет запускаться без проблем, но мы по-прежнему не используем `HttpClientFactory`. Как вы помните из главы 5, в микросервисе `Shopping Cart` есть класс `ProductCatalogClient`, отвечающий за отправку запросов микросер-

вису Product Catalog. Модифицируем этот класс так, чтобы он получал в качестве параметра ссылку на `HttpClientFactory` и при каждом запросе создавался объект `HttpClientFactory` в загрузчике, внедряемом в создаваемые Nancy объекты (листинг 9.12).

Листинг 9.12. Создаем объект `HttpClientFactory` для каждого запроса

```
private readonly IHttpClientFactory httpClientFactory;

public ProductCatalogClient(
    ICache cache,
    IHttpClientFactory httpClientFactory) ←
{
    this.cache = cache;
    this.httpClientFactory = httpClientFactory;
}
```

Nancy внедряет `HttpClientFactory`, регистрируемый в загрузчике для каждого запроса

Далее для создания объектов `HttpClient` мы начнем использовать `HttpClientFactory` вместо того, чтобы создавать их с помощью оператора `new`. В методе `RequestProductFromProductCatalogue` есть следующая строка кода:

```
var httpClient = new HttpClient(productCatalogBaseUrl);
```

Замените ее на такую:

```
var httpClient =
    this.httpClientFactory.Create(new Uri(productCatalogBaseUrl));
```

Теперь выполняемые из `ProductCatalogClient` запросы будут включать заголовок `Correlation-Token`, содержащий соответствующий маркер корреляции. Если микросервис Product Catalog работает правильно: читает маркер корреляции из заголовка и использует его во всех журнальных сообщениях, как и должен, то мы сможем отслеживать движение запросов пользователей на добавление товаров в их корзины заказов через микросервис Shopping Cart в микросервис Product Catalog. Это очень удобно при отладке проблем, возникающих при производственной эксплуатации.

Далее реализуем два элемента промежуточного ПО для журналирования запросов и показателей их производительности.

Журналирование запросов и показателей их производительности

Нам осталось написать два элемента промежуточного ПО. Первый из них будет журналировать все входящие запросы и исходящие ответы. Поскольку в этих журнальных сообщениях содержится маркер корреляции, мы сможем соотнести запрос и ответ. Добавьте следующий код промежуточного ПО для журналирования запросов в файл `LoggingMiddleware.cs` (листинг 9.13).

Листинг 9.13. Промежуточное ПО для журналирования запросов и ответов

```
public class RequestLogging
{
    public static AppFunc Middleware(AppFunc next, ILogger log)
    {
        return async env =>
        {
            var owinContext = new OwinContext(env);
            log.Information(
                "Incoming request: {@Method}, {@Path}, {@Headers}",
                owinContext.Request.Method,
                owinContext.Request.Path,
                owinContext.Request.Headers);
            await next(env);
            log.Information(
                "Outgoing response: {@StatusCode}, {@Headers}",
                owinContext.Response.StatusCode,
                owinContext.Response.Headers);
        };
    }
}
```

Отправляет журнальное сообщение, содержащее метод запроса, путь и заголовки

Отправляет запрос в остальную часть конвейера

Отправляет журнальное сообщение, содержащее код состояния и заголовки ответа

Это промежуточное ПО OWIN, написанное в лямбда-стиле, выбирает из запроса и ответа наиболее важную информацию и отправляет в журнал сообщения с этой информацией.

Далее добавим в файл `LoggingMiddleware.cs` промежуточное ПО для журналирования производительности запросов. Оно использует класс `Stopwatch`, так что добавим его с помощью директивы `using` вверху файла:

```
using System.Diagnostics;
```

Само промежуточное ПО выглядит следующим образом (листинг 9.14).

Листинг 9.14. Промежуточное ПО, измеряющее длительность выполнения запроса и заносящее результат в журнал

```
public class PerformanceLogging
{
    public static AppFunc Middleware(AppFunc next, ILogger log)
    {
        return async env =>
        {
            var stopwatch = new Stopwatch();
            stopwatch.Start();
            await next(env);
            stopwatch.Stop();
            var owinContext = new OwinContext(env);
            log.Information(
                "Request: {@Method} {@Path} executed in {RequestTime:000} ms",
                owinContext.Request.Method, owinContext.Request.Path,
                stopwatch.ElapsedMilliseconds);
        };
    }
}
```

Запускаем таймер до выполнения остальной части конвейера и останавливаем его после завершения выполнения

Отправляем журнальное сообщение с информацией о запросе и длительности его выполнения

Этот элемент промежуточного ПО измеряет время, которое занимает выполнение оставшейся части конвейера. Если номестить промежуточное ПО в начало конвейера, это значение будет соответствовать времени, требующемуся для обработки запроса. По завершении выполнения оставшейся части конвейера промежуточное ПО заносит время выполнения в журнал.

Конфигурация конвейера OWIN с промежуточным ПО для маркеров корреляции и журналирования

Мы написали все три части промежуточного ПО и теперь можем включить их в конвейер OWIN микросервиса Shopping Cart. Создадим конвейер с промежуточным ПО для маркеров корреляции в начале, за которым следует промежуточное ПО для журналирования запросов, промежуточное ПО для журналирования показателей производительности, промежуточное ПО мониторинга и, наконец, Nancy. Конвейер OWIN комонуется в методе `Configure` класса `Startup`. После добавления всего промежуточного ПО метод `Configure` выглядит так (листинг 9.15).

Листинг 9.15. Конвейер OWIN с промежуточным ПО и Nancy

```
public void Configure(IApplicationBuilder app)
{
    var log = ConfigureLogger();

    app.UseOwin(buildFunc =>
    {
        buildFunc(next => CorrelationToken.Middleware(next));
        buildFunc(next => RequestLogging.Middleware(next, log));
        buildFunc(next => PerformanceLogging.Middleware(next, log));
        buildFunc(next => new MonitoringMiddleware(next, HealthCheck).Invoke);
        buildFunc.UseNancy();
    });
}
```

После создания конвейера при запуске микросервиса Shopping Cart и отнравке ему запросов вы увидите в консоли журнальные сообщения о запросах и показателях производительности. Например, запустите этот микросервис с помощью утилиты `dotnet` и отнравьте ему следующий запрос для мониторинга запросов:

```
GET /_monitor/shallow HTTP/1.1
Host: localhost:5000
Accept: application/json
```

Вы увидите в консоли следующий вывод:

```
PS> dotnet run
Hosting environment: Production
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

```
22:24:04 [Information] (1717b4c2-9734-4425-933f-7cf18f9b0bca) Incoming
➡ request: "GET", PathSt
```

```

ring { Value: "/_monitor/shallow", HasValue: True }, [KeyValuePair`2 {
  ➤ Key: "Cache-Control",
  Value: ["no-cache"] }, KeyValuePair`2 { Key: "Connection", Value:
  ➤ ["keep-alive"] }, KeyValuePair
air`2 { Key: "Content-Type", Value: ["applic 0.2 .. ation/json"] },
  ➤ KeyValuePair`2 { Key: "Accept", V
  alue: ["application/json"] }, KeyValuePair`2 { Key: "Accept-Encoding",
  ➤ Value: ["gzip, deflate
  , sdch"] }, KeyValuePair`2 { Key: "Accept-Language", Value:
  ➤ ["en-US,en;q=0.8,da;q=0.6,nb;q=0.
  4,sv;q=0.2"] }, KeyValuePair`2 { Key: "Cookie", Value:
  ➤ ["__atuvc=4%7C37; todoUser=chr_horsdal
  ; _nc=xoP1b4Vx70TPRwvKVfanfyaAckK8L8pkaVJehq4pns%3di5jbH1%2fJPKfbgy99
  ➤ ZaaFUhSoZtmNTYZxuHm3NPr
  3mIdyVccjbWCrKBtYCYZFbaknlwUTSbgMnEwj18HkLi%2fI4XTsa4hzaEngPZ5wp8waBcO
  ➤ 7sOHeARlfeMhbMF6x11iwkT
  HebV67sBTW8B7S8u2jCgVtSlrps%2bXx6s3W4MH1NGJjJK6wy4804xFmauLJWBQ%2bvISV
  ➤ XpROXKu%2fm5QdeKwTNw%3d
  %3d"] }, KeyValuePair`2 { Key: "Host", Value: ["localhost:5000"] },
  ➤ KeyValuePair`2 { Key: "Us
  er-Agent", Value: ["Mozilla/5.0 (Windows NT 10.0; WOW64)
  ➤ AppleWebKit/537.36 (KHTML, like Geck
  o) Chrome/47.0.2526.111 Safari/537.36"] }, KeyValuePair`2 { Key:
  ➤ "Location", Value: ["http://
  www.google.com"] }, KeyValuePair`2 { Key: "Postman-Token", Value:
  ➤ ["e180f522-dea5-0c56-0e35-9
  c2be0626112"] }}

```

```

22:24:04 [Information] (1717b4c2-9734-4425-933f-7cf18f9b0bca)
  ➤ Request: "GET" PathString { Val
  ue: "/_monitor/shallow", HasValue: True } executed in 001 ms

```

```

22:24:04 [Information] (1717b4c2-9734-4425-933f-7cf18f9b0bca)
  ➤ Outgoing response: 200, [KeyVal
  uePair`2 { Key: "Date", Value: ["Mon, 25 Jan 2016 21:24:04 GMT"] },
  ➤ KeyValuePair`2 { Key: "Server", Value: ["Kestrel"] }]

```

В этом выводе содержится три журнальных сообщения: одно с запросом, второе с временем выполнения запроса и третье — с ответом, как мы и ожидали. Вы могли обратить внимание на следующее обстоятельство: поскольку мы используем структурное журналирование, в журнальное сообщение можно с легкостью включить множество данных. Сообщения в консоли на первый взгляд могут ошеломить, но при отправке их микросервису Log, у которого имеются возможности поиска и удобный графический интерфейс — например, в качестве хранилища может использоваться Elasticsearch, а в качестве GUI Kibana, — лишние данные не представляют никаких проблем. Напротив, это ценная информация.

В этой главе мы создали четыре элемента промежуточного ПО OWIN в микросервисе Shopping Cart, но ни один из них не использует никакой бизнес-логики этого микросервиса. Все четыре могли с такой же легкостью находиться в любом другом микросервисе. В главе 11 мы извлечем эти элементы промежуточного ПО из микросервиса Shopping Cart и скомбинируем их в пакеты NuGet, готовые для повторного использования в других микросервисах.

9.5. Резюме

- ❑ Для системы микросервисов необходимы мониторинг и журналирование, как и для любой другой серверной системы.
- ❑ Необходимо контролировать и журналировать функционирование всех микросервисов.
- ❑ В силу большого числа микросервисов в системе наладка мониторинга и журналирования должна быть простой и удобной.
- ❑ Маркеры корреляции облегчают отслеживание движения запроса по цепочке микросервисов.
- ❑ Можно использовать промежуточное ПО OWIN для создания двух конечных точек для мониторинга: одной, которая будет отвечать на все запросы **204 No Content** (204 Нет содержимого), и второй, которая перед этим будет сначала проверять состояние микросервиса.
- ❑ Структурное журналирование — удобный способ включения в журнальные сообщения ценной информации и обеспечения возможности поиска по этой информации.
- ❑ Serilog — библиотека для реализации структурного журналирования.
- ❑ Для добавления маркеров корреляции в журнальные сообщения можно воспользоваться промежуточным ПО OWIN и контекстом журнала библиотеки Serilog.
- ❑ Для реализации журналирования запросов и ответов, а также времени выполнения запросов также можно воспользоваться промежуточным ПО OWIN.
- ❑ Для передачи уникального для каждого запроса маркера корреляции из промежуточного ПО в загрузчик Nancy можно воспользоваться окружением OWIN.
- ❑ Можно воспользоваться `HttpClient`, чтобы включить заголовок `Correlation-Token` во все исходящие HTTP-запросы.

10 Обеспечение безопасности взаимодействия микросервисов

В этой главе:

- ❑ поиск в системе микросервисов подходящего места для аутентификации и авторизации пользователей;
- ❑ выбор уровня доверия в системе микросервисов;
- ❑ применение IdentityServer для аутентификации пользователей;
- ❑ авторизация запросов от микросервиса к микросервису.

До этого момента мы игнорировали безопасность, но для большинства систем это очень важный вопрос, требующий пристального внимания. В этой главе мы обсудим способы решения проблем безопасности в системах микросервисов. В монолитной системе за аутентификацию и авторизацию пользователей отвечает монолит — в конце концов, кроме монолита в ней ничего нет. В системе микросервисов же, чтобы ответить на большинство запросов пользователей, приходится вовлекать несколько микросервисов. Вопрос в следующем: какие из них должны отвечать за аутентификацию, а какие — за авторизацию? Не помешает также задать себе вопрос: насколько микросервисы могут доверять друг другу? Если один микросервис аутентифицирует пользователя, могут ли другие микросервисы доверять этому пользователю? Все ли микросервисы должны иметь право обращаться друг к другу?

Ответы на эти вопросы сильно меняются от системы к системе. В начале главы обсудим пути решения этих вопросов, а затем углубимся в реализацию одного из возможных решений.

10.1. Безопасность микросервисов

Безопасность — важная тема практически для любой серверной системы. Но эта тема очень широка, и большая часть ее выходит за рамки данной книги. Мы сосредоточимся на двух вопросах безопасности, имеющих отношение к разработке систем

микросервисов: *аутентификации (authentication)* и *авторизации (authorization)*, а также на безопасных взаимодействиях между микросервисами.

В большинстве систем имеется функциональность, доступная только для пользователей, выполнивших вход в приложение. Вернемся к обсуждавшейся в предыдущих главах POS-системе. В главах 3 и 4 мы говорили о добавлении в нее программы лояльности, которая дает возможность зарегистрированным пользователям получать специальные предложения по электронной почте в зависимости от их интересов. Если пользователей интересует гольф, их будут оповещать о специальных предложениях, касающихся шариков для гольфа. Если пользователь хочет отредактировать настройки своих интересов, например отказаться от гольфа и заняться шитьем, ему придется выполнить вход в систему. В противном случае один пользователь мог бы отредактировать настройки другого, в результате чего система оповещала бы их не о тех предложениях. Задача *аутентификации* — убедиться, что пользователь именно тот, за кого себя выдает. А определить, что пользователь имеет право делать — например, редактировать только свои настройки, а не чужие, — задача *авторизации*.

Аутентификация и авторизация — проблемы, с которыми ваша система, вероятно, столкнется независимо от того, будет ли она основана на микросервисах. Отличие состоит в том, что гранулярная структура системы микросервисов ставит вопрос: какие из микросервисов должны отвечать за аутентификацию и авторизацию? В следующих двух разделах мы ответим на него.

Передаваемые и хранимые данные

Системы обрабатывают данные. Они почти всегда имеют большое значение для системы, а часто — и для пользователей. Эти данные могут быть конфиденциальными — таковы, например, домашние адреса пользователей, номера кредитных карт или истории болезней. Даже если данные неконфиденциальны, они все равно могут быть ценными для системы — каталог товаров интернет-магазина не является конфиденциальным, но он весьма ценен для фирмы-владельца веб-сайта. Обрабатываемые системой данные важны и должны обрабатываться безопасно. В целом обрабатываемые данные можно разделить на две категории.

- *Передаваемые данные (data in motion)*. Данные, перемещаемые из одной части системы в другую, называют передаваемыми данными. Например, при обмене данными между взаимодействующими микросервисами посредством команд, запросов или событий данные, которыми обмениваются, являются передаваемыми.
- *Хранимые данные (data at rest)*. Данные, которые сохраняются для дальнейшего использования, называются хранимыми. Например, элемент, который микросервис — владелец элемента данных хранит в своей базе данных, является хранимым.

Обеспечить безопасность данных важно в обоих случаях. В этой главе мы сосредоточимся на передаваемых данных, поскольку именно в этой области система микросервисов отличается от систем с другими архитектурами. Методы обеспечения безопасности хранимых данных в системе микросервисов не отличаются от таковых в системе с монолитной или сервисно-ориентированной архитектурой.

Аутентификация пользователей на периферии

Аутентификация — это проверка того, что пользователи — те, за кого себя выдают. В контексте системы микросервисов это означает проверку того, что запросы выполняются от имени тех пользователей, от которых заявлены. Рассмотрим пример (рис. 10.1), демонстрирующий часть POS-системы из глав 3 и 4 — часть, примыкающую к микросервису Loyalty Program.

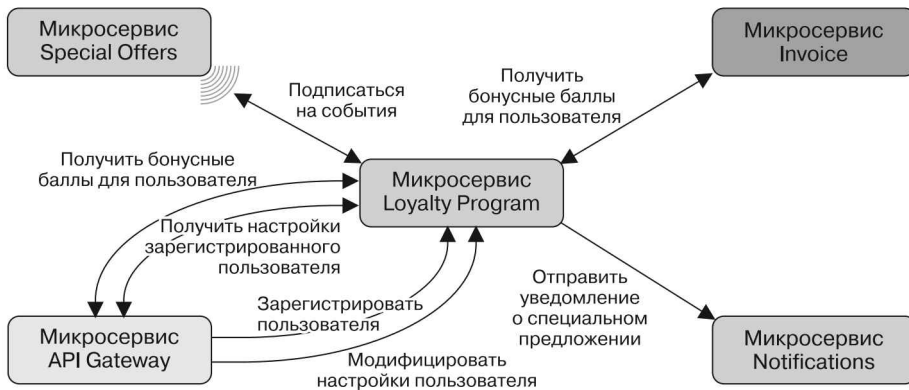


Рис. 10.1. Микросервис Loyalty Program в POS-системе

Если добавить сюда клиентскую часть с веб-интерфейсом, в котором зарегистрированные пользователи могли бы редактировать свои настройки, например исключить из них свой интерес к гольфу и добавить вместо него шитье, можно получить что-то вроде рис. 10.2. Необходимо, чтобы пользователи могли редактировать только свои настройки, но не настройки других пользователей. Очевидно, это можно делать в API Gateway — микросервисе, получающем от клиента запрос на обновление настроек пользователя. Этот запрос выполняется от имени какого-то пользователя, значит, необходимо убедиться, что клиент имеет право выполнять запросы от имени пользователей. Для этого нужно проверить, что соответствующий пользователь вошел в систему.

На рис. 10.3 показано добавление в систему микросервиса Login. Этот новый микросервис отвечает за процесс входа в систему, но шлюз API по-прежнему отвечает за то, чтобы принимались только запросы от вошедших в систему пользователей. В микросервисе API Gateway аутентификация реализуется элементом промежуточного ПО, который перенаправляет пользователей в микросервис Login, если они еще не вошли в систему. Микросервис Login определяет, как именно пользователь может войти в систему, и проводит его через процесс входа. Пользователи могут входить в систему множеством различных способов, в том числе:

- ☐ с помощью имени пользователя и пароля;
- ☐ через двухфакторный механизм входа;
- ☐ через внешнюю систему, например Active Directory;
- ☐ через поставщиков идентификационной информации в социальных сетях, таких как Facebook, Twitter, Google и т. д.

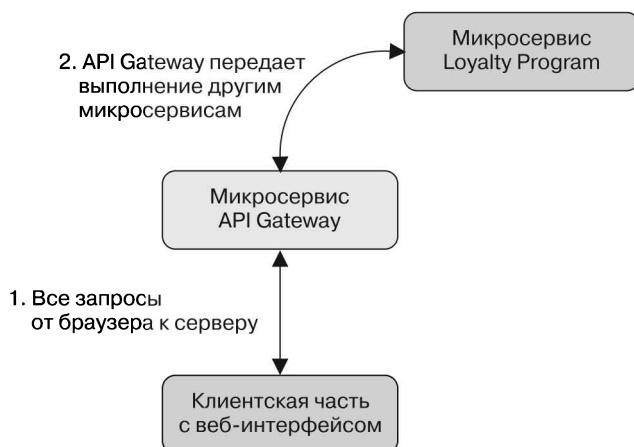


Рис. 10.2. Клиентская часть с веб-интерфейсом для пользователей программы лояльности взаимодействует только с микросервисом API Gateway

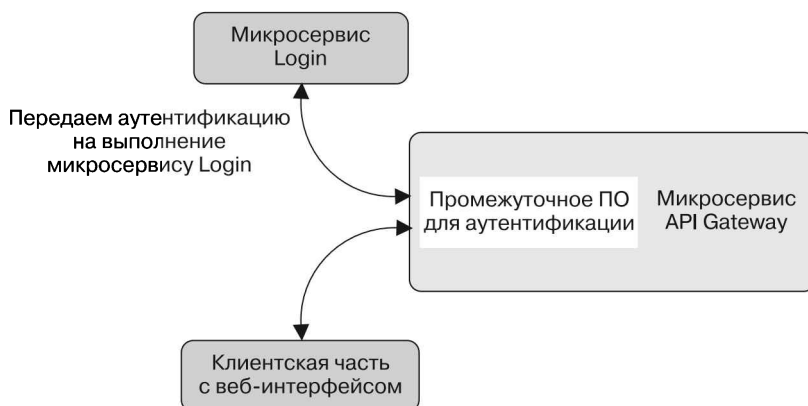


Рис. 10.3. Добавление микросервиса Login в POS-систему. Микросервис Login отвечает за аутентификацию, а элемент промежуточного ПО в микросервисе API Gateway перенаправляет в него неаутентифицированных пользователей

Вне зависимости от механизма входа в систему микросервис Login осуществляет процедуру входа и предоставляет микросервису API Gateway подтверждение личности пользователя в таком виде, который дает API Gateway возможность проверить подлинность. Для этого существуют стандартизированные протоколы. Реализации во второй половине этой главы используют для этого протокол OpenId Connect.

Обратите внимание на то, что аутентификация пользователей — периферия системы, то есть она выполняется микросервисом, получающим запрос непосредственно от клиента. Это общий принцип: аутентификация пользователей выполняется на периферии системы.

Авторизация пользователей в микросервисах

Мы выяснили, что аутентификация должна выполняться на периферии системы микросервисом, непосредственно получающим запрос от клиента. После аутентификации запроса мы знаем, кто такой клиент, но не знаем, имеет ли он право на выполнение данного запроса. Это задача авторизации. Пользователи имеют право на модификацию только своих настроек. Система должна отклонять запросы клиентского приложения, отправленные от имени пользователя А и пытающиеся изменить настройки пользователя В.

В системе микросервисов находящийся на периферии микросервис — тот, который инициирует аутентификацию, зачастую отличается от микросервиса, выполняющего требуемое запросом действие. На рис. 10.4 снова приведена схема программы лояльности. Запросы от пользователя получает микросервис API Gateway, но за отслеживание настроек интересов пользователя отвечает микросервис Loyalty Program. Следовательно, обновлением настроек интересов пользователя занимается микросервис Loyalty Program.



Рис. 10.4. Авторизация должна быть частью бизнес-логики; микросервис Loyalty Program отвечает за авторизацию обновлений настроек пользователей

Как вы помните из главы 3, область действия микросервисов в основном определяется бизнес-возможностями: бизнес-возможность реализуется одним микросервисом. Это включает авторизацию, поскольку она является частью бизнес-правил подобной функциональной возможности, и вполне согласуется с тем, что микросервис Loyalty Program определяет, имеет ли право пользователь обновлять настройки, относящиеся к его интересам. Чтобы это работало, необходимо передавать идентификационную информацию пользователя от микросервиса к микросервису. В программе лояльности микросервис API Gateway должен включать в отправляемые микросервису Loyalty Program запросы идентификаторы пользователей.

Насколько микросервисы должны доверять друг другу

Микросервисы должны взаимодействовать, чтобы обеспечить необходимые конечным пользователям функциональные возможности, а значит, можно предположить, что все микросервисы будут именно к этому и стремиться. Но как можно быть в этом уверенными? Может ли злоумышленник получить контроль над микросервисом и изменить его поведение? Да, это вполне возможно. Тогда вопрос преобразуется в следующий: может ли один микросервис доверять другому? Ответ на этот вопрос: увы, когда как. Это зависит, например, от угроз, с которыми сталкивается система,

и от возможных последствий успешной атаки. Зависит и от других факторов, таких как организационная структура и соответствие нормативным требованиям.

При максимальном уровне доверия между микросервисами все они полностью доверяют всем запросам и ответам от любого другого микросервиса. Именно такой уровень доверия подразумевался в созданных нами до сих пор в этой книге микросервисах.

Принцип *эшелонированной защиты* (*defense in depth*) гласит: вышеприведенное — далеко не лучшая идея. Если злоумышленник сумеет взломать хотя бы один микросервис и заставить его выполнять запросы к другим, он получит доступ ко всей системе. Для конкретной системы подобная ситуация может быть как приемлемой, так и неприемлемой. Во втором случае имеет смысл ограничить то, какие микросервисы могут взаимодействовать между собой.

Например, обратимся к рис. 10.1. Микросервисы API Gateway, Special Offers и Invoice могут обращаться к микросервису Loyalty Program, а Notifications — нет. В то же время только Loyalty Program может обращаться к Notifications, другие микросервисы — нет. Это ограничивает возможные действия злоумышленника в случае взлома одного из микросервисов.

Можно пойти дальше и ограничить не только то, какие микросервисы могут обращаться друг к другу, но и к каким конечным точкам они могут обращаться. Например, Special Offers может обращаться только к конечной точке ленты событий микросервиса Loyalty Program, но не к конечным точкам администрирования пользователей. Аналогично микросервис API Gateway может обращаться к конечным точкам администрирования пользователей, но не к ленте событий.

Немного терминологии

Определим несколько терминов.

- *Область действия* — идентификатор одной или нескольких конечных точек, которые нам нужно защитить от злоумышленников. Микросервис, которому требуется обратиться к любой из них, должен получить на это разрешение. Оно представляет собой маркер, содержащий область действия. Вы можете рассматривать области действия как имена групп защищаемых конечных точек. Например, все конечные точки, которые потенциально могут изменить информацию о товаре, должны иметь область действия `product_information_write`.
- *Маркер доступа* — подписанный объект, используемый для предоставления доступа к ресурсам. Микросервис может запросить маркер доступа к определенной области действия у микросервиса Login. Если Login разрешает микросервису доступ к этой области действия, то возвращает маркер доступа, который можно передавать вместе с запросами к конечным точкам, соответствующим данной области действия.
- *OAuth и OpenId Connect* — открытые стандарты, используемые совместно для аутентификации и авторизации как конечных пользователей, так и микросервисов. Мы будем использовать эти два протокола для аутентификации и авторизации, но углубляться в подробности я не стану, так как их реализацию обеспечит нам фреймворк IdentityServer.

Чтобы обеспечить соблюдение ограничений по взаимодействию микросервисов, мы создадим в микросервисе Log *области действия*. При необходимости вызова другого микросервиса каждый микросервис должен будет запрашивать у микросервиса

Login разрешение в виде маркера доступа к конкретной области действия. Микросервис Login должен обеспечить аутентификацию пользователя, например потребовав, чтобы каждый сервис включал в свои запросы уникальный секретный ключ, а затем решить, предоставлять ли пользователю маркер доступа на основе заданных настроек безопасности. Получающий запрос микросервис может затем просмотреть маркер в соответствующем элементе промежуточного ПО и проверить, что в нем присутствует нужная область действия.

Например, в микросервис Loyalty Program можно добавить элемент промежуточного ПО, которому для всех запросов к конечным точкам, регистрирующим пользователей или изменяющим их данные, требуется область действия `loyalty_program_write`. Микросервису API Gateway при этом придется запрашивать у микросервиса Login маркер для области действия `loyalty_program_write` до выполнения запросов к Loyalty Program. А он будет отправлять этот маркер, только если настроить его так, чтобы он разрешал обращения от API Gateway к Loyalty Program. Эта схема гарантирует, что за определение того, какие микросервисы могут взаимодействовать, будет отвечать микросервис Login.

Даже если ограничить то, какие микросервисы и как могут взаимодействовать, все равно остается неявный элемент веры в подлинность любой отправляемой одним микросервисом другому информации. Если воспользоваться для взаимодействия между микросервисами протоколом HTTPS вместо HTTP, можно обеспечить шифрование транспортного уровня, а следовательно, определенную степень защиты от фальсификации запросов злоумышленниками при выполнении запроса от одного микросервиса к другому. Но даже в этом случае остается вопрос: можно ли доверять поступающим с запросом данным? Особенно важна тут личность пользователя, поскольку на ее основе мы выполняем авторизацию.

Эшелонированная защита

Эшелонированная защита (defense in depth) — подход к безопасности, использующий сочетание нескольких механизмов защиты. Идея заключается в реализации многослойной стратегии защиты: если злоумышленник сумеет пробиться через первую линию обороны, он столкнется со следующей. Например, даже если микросервис на периферии системы аутентифицировал и авторизовал все входящие запросы, у него не должно быть прав администратора на сервере, где он выполняется. Даже если злоумышленник обойдет авторизацию и обманом заставит микросервис выполнить загруженный им код, у него все равно не будет полного контроля над сервером — он ограничен теми правами, которые предоставляет микросервису операционная система.

В предыдущем разделе мы видели, что шлюз API должен передавать идентификационные данные пользователя вместе с запросами к микросервису Loyalty Program. Если нам нужно, чтобы получающий в запросе идентификационные данные пользователя микросервис мог их проверить, необходимо передавать эти данные в виде зашифрованного маркера, а не простого текста.

Как видите, уровень доверия между микросервисами меняется от системы к системе. Вы можете выбирать различные уровни доверия. В следующих разделах мы воплотим уровень доверия, соответствующий описанным ранее в данной главе принципам безопасности, но это далеко не единственный способ их реализации.

10.2. Реализация безопасного обмена сообщениями между микросервисами

Далее займемся реализацией вопросов безопасности, относящихся к программе лояльности. Требования к безопасности следующие.

- ❑ Аутентифицировать пользователей в микросервисе API Gateway.
- ❑ Ограничивать допустимые взаимодействия между микросервисами.
- ❑ Обеспечить возможность проверки микросервисами пользовательских идентификационных данных, передаваемых от других микросервисов.

При воплощении этих требований будем основываться на нескольких стандартах:

- ❑ OpenId Connect для аутентификации пользователей;
- ❑ OAuth для ограничения того, какие микросервисы могут взаимодействовать между собой;
- ❑ JSON Web Tokens (JWT) для пользовательских идентификационных данных.

Мы также воспользуемся продуктом с открытым исходным кодом IdentityServer, который будет играть роль микросервиса Login. Я познакомлю вас с IdentityServer в следующем разделе, и мы настроим его и запустим.

Мы реализуем в микросервисе API Gateway показанную на рис. 10.5 схему. Конвейер в микросервисе API Gateway содержит промежуточное ПО аутентификации и Nancy. Промежуточное ПО аутентификации проверяет, аутентифицированы ли пользователи, и, если нет, перенаправляет их в микросервис Login.

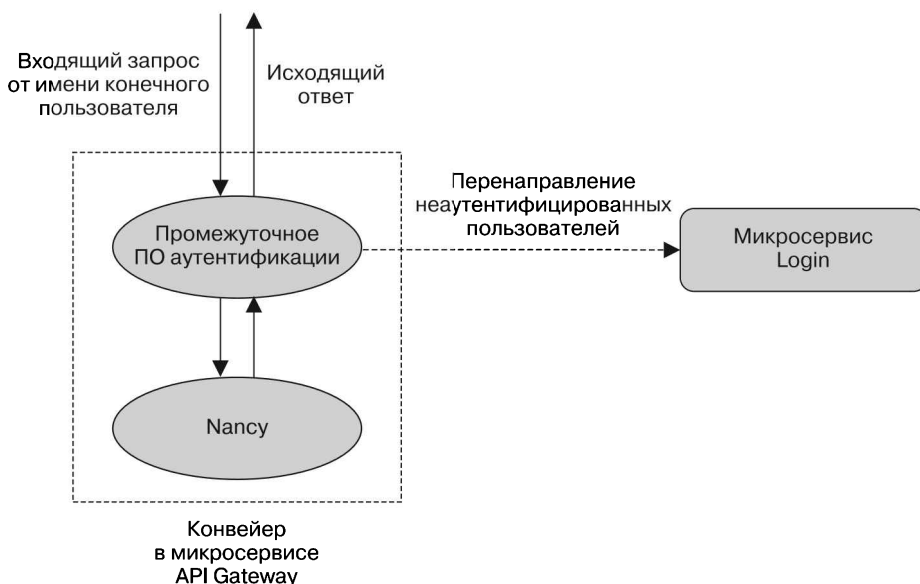


Рис. 10.5. Применение промежуточного ПО аутентификации для перенаправления неаутентифицированных пользователей в микросервис Login

Еще немного терминологии

Прежде чем заняться реализацией, нужно определить еще несколько терминов.

- *JSON Web Token (веб-маркер JSON, JWT)* — стандартный формат для маркеров доступа. JWT можно подписать криптографически, а значит, у микросервиса, получающего JWT от другого микросервиса, есть возможность проверить его подлинность. Если маркер подлинный, его содержимому можно доверять, а значит, можно с уверенностью положиться на JWT при авторизации.
- *Утверждения (claims)* и тип `ClaimsPrincipal` (Утверждения) — это пары «ключ — значение», используемые для предоставления информации об аутентифицированном конечном пользователе или микросервисе. Если речь идет о конечном пользователе, утверждения могут включать его идентификатор, имя, адрес электронной почты и т. д., а также имеющиеся у данного пользователя права, например доступ на запись к определенным данным. В случае микросервиса утверждение может включать область действия, к которой он имеет доступ. У платформы .NET есть встроенная поддержка утверждений с помощью типов из пространства имен `System.Security.Claims`. В частности, мы будем использовать тип `ClaimsPrincipal`. Он представляет аутентифицированного пользователя и позволяет получить доступ для просмотра всех его утверждений.
- *Заголовок HTTP-запроса Authorization* — стандартный заголовок HTTP-запроса, который мы будем использовать для передачи маркеров доступа вместе с запросами.

В микросервисе *Loyalty Program* мы реализуем показанный на рис. 10.6 конвейер. Перед *Nancy* располагаются два элемента промежуточного ПО: один обеспечивает наличие во всех входящих запросах маркеров доступа, необходимых для взаимодействия с микросервисом *Loyalty Program*, а второй выполняет чтение идентификационных маркеров инициировавших запрос конечных пользователей.

В следующих разделах мы реализуем вот что.

- ❑ Микросервис *Login* для POS-системы. Он будет целиком основан на *IdentityServer*.
- ❑ Промежуточное ПО в микросервисе *API Gateway*, инициирующее аутентификацию пользователя при входящих запросах. Если пользователь не выполнил вход в систему, он будет перенаправлен в микросервис *Login*, где сможет это сделать. Микросервис *API Gateway* будет использовать *Login* для проверки выполнения пользователями входа в систему при последующих запросах.
- ❑ Безопасное взаимодействие между микросервисами *API Gateway* и *Loyalty Program*. Оно состоит из трех частей:
 - задание области действия в микросервисе *Login* и передача микросервису *API Gateway* разрешения на вызов *Loyalty Program*;
 - в микросервисе *API Gateway* запрос маркера у микросервиса *Login* перед каждым выполнением запроса к *Loyalty Program*. Этот маркер мы будем помещать в заголовок *Authorization* всех запросов к микросервису *Loyalty Program*;
 - в микросервисе *Loyalty Program* реализация промежуточного ПО, требующего маркера с соответствующей областью действия. Если нужная область действия отсутствует, промежуточное ПО будет прерывать выполнение конвейера и отправлять вызывающей стороне ответ с кодом состояния **403 Forbidden** (403 Запрещено). Если же нужная область действия присутствует, выполнение запроса будет продолжено стандартным образом.

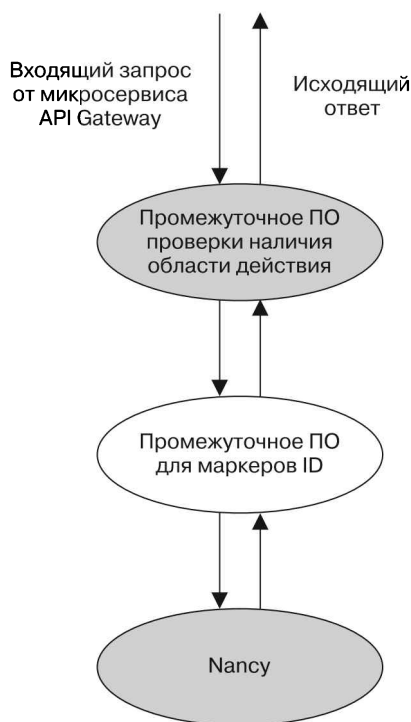


Рис. 10.6. Использование промежуточного ПО для авторизации запросов и идентификации конечного пользователя

- Передачу идентификатора пользователя от микросервиса API Gateway микросервису Loyalty Program и использование его для выполнения авторизации в модулях Nancy. Для этого мы:
- добавим идентификатор нользователя в заголовки всех занросов от микросервиса API Gateway к микросервису Loyalty Program;
 - будем читать идентификатор пользователя в элементе промежуточного ПО в микросервисе Loyalty Program и присваивать его свойству контекста Nancy;
 - используем это свойство из контекста Nancy в модулях Nancy для авторизации.

Реализовав все это, мы выполним заявленные в начале данного раздела требования безопасности.

Познакомьтесь: IdentityServer

IdentityServer — продукт с открытым исходным кодом, облегчающий реализацию единого входа в систему и управление доступом в веб-приложениях и API для протокола HTTP. В его основе лежит нлатформа .NET, это гибкий но своей сути фреймворк, что позволяет настраивать его под свои потребности. В данном примере настроим IdentityServer так, чтобы он играл роль микросервиса Login в POS-системе. Мы будем использовать жестко зашитые базы данных как для пользователей, так

и для областей действия, но IdentityServer поддерживает и многие другие варианты, например использование для пользователей реляционных баз данных или делегирование входа в систему иному сервису идентификации — Twitter, Google, Active Directory или одному из многих других.

ПРИМЕЧАНИЕ

Больше информации по этому вопросу, включая детальную техническую документацию, иллюстрирующую установку и использование IdentityServer, можно найти на сайте <https://identityserver.github.io/Documentation>. В этом разделе мы не станем углубляться в детали и покажем только реализацию простейшей схемы его применения.

Реализуем простой микросервис Login на основе IdentityServer. Необходимо сделать следующее.

1. Создать пустой проект ASP.NET Core, точно такой же, как и для любого другого микросервиса.
2. Добавить с помощью системы управления пакетами NuGet фреймворк IdentityServer в этот новый микросервис.
3. Выполнить настройки IdentityServer в классе `Startup`.
4. Задать несколько жестко зашитых в код пользователей и область действия для микросервиса Loyalty Program.

По окончании получим микросервис Login с показанными на рис. 10.7 компонентами. Отмечу, что почти весь микросервис реализован IdentityServer.

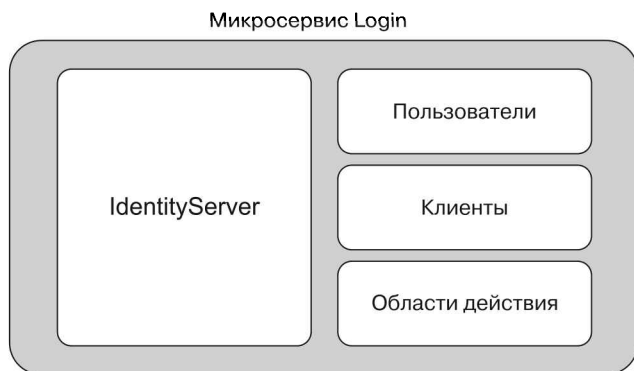


Рис. 10.7. Основанный на IdentityServer микросервис Login

Пройдем по озвученным шагам. Во-первых, создайте новый пустой проект ASP.NET Core с помощью Visual Studio или Yeoman и назовите его `Login`. Вы уже создали множество таких проектов при чтении данной книги, так что я не стану повторять все подробности.

Далее добавьте пакет IdentityServer системы управления пакетами NuGet путем добавления `IdentityServer4` в файл `project.json` в новом микросервисе вместе

с пакетом Serilog и несколькими пакетами журналирования ASP.NET Core. Раздел зависимостей в файле `project.json` должен выглядеть следующим образом:

```
"dependencies": {
  "Microsoft.NETCore.App": {
    "version": "1.0.0",
    "type": "platform"
  },
  "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
  "Microsoft.AspNetCore.StaticFiles": "1.0.0",
  "Microsoft.AspNetCore.Mvc": "1.0.0",
  "Microsoft.Extensions.Logging.Console": "1.0.0",
  "Microsoft.Extensions.Logging.Debug": "1.0.0",
  "IdentityServer4": "1.0.0-beta5",
  "Serilog": "2.0.0-rc-600"
},
```

После выполнения команды `dotnet restore` фреймворк IdentityServer будет добавлен в проект.

Слегка подправим настройки нового проекта, чтобы микросервис Login работал на отдельном порте. Измените файл `program.cs` так, как показано в листинге 10.1, чтобы этот микросервис работал на порте 5001.

Листинг 10.1. Задаем порт, на котором будет работать микросервис Login

```
namespace Login
{
    using System.IO;
    using Microsoft.AspNetCore.Hosting;

    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .UseUrls("http://localhost:5001") ← Настройка ASP.NET Core
                .Build();                                     на прослушивание
                                                         на порте 5001

            host.Run();
        }
    }
}
```

Третий шаг — настройка IdentityServer. Ради простоты воспользуемся хранящими данные в оперативной памяти версиями всего, что только нужно IdentityServer. На практике, вероятнее всего, мы бы использовали другую реализацию входа пользователя в систему и, возможно, другие реализации остальных частей. Конфигурация IdentityServer выполняется в коде с помощью API IdentityServer в классе `Startup`. В листинге 10.2 используются три класса: `Users`, `Claims` и `Scoped`, которые мы вскоре напомним.

Листинг 10.2. Класс Startup с простейшими настройками IdentityServer

```

namespace Login
{
    using System.IO;
    using Microsoft.AspNetCore.Builder;
    using Microsoft.AspNetCore.Hosting;
    using System.Security.Cryptography.X509Certificates;
    using Microsoft.AspNetCore.Http;
    using Microsoft.Extensions.Logging;
    using Microsoft.Extensions.DependencyInjection;

    using Configuration;

    public class Startup
    {
        private readonly IHostingEnvironment environment;

        public Startup(IHostingEnvironment env)
        {
            this.environment = env;
        }

        public void ConfigureServices(IServiceCollection services)
        {
            var cert =
                new X509Certificate2(
                    Path.Combine(
                        this.environment.ContentRootPath,
                        "idsrv3test.pfx"),
                    "idsrv3test");

            services.AddSingleton<IHttpContextAccessor,
                ➤ HttpContextAccessor>();
            var builder = services
                .AddIdentityServer()
                .SetSigningCredential(cert);
            builder.AddInMemoryClients(Clients.Get());
            builder.AddInMemoryScopes(Scopes.Get());
            builder.AddInMemoryUsers(Users.Get());
            services.AddMvc();

            public void Configure(IApplicationBuilder app, ILoggerFactory
                ➤ loggerFactory)
            {
                loggerFactory.AddConsole(LogLevel.Trace);
                loggerFactory.AddDebug(LogLevel.Trace);

                app.UseCookieAuthentication(new CookieAuthenticationOptions
                {
                    AuthenticationScheme = "Temp",
                    AutomaticAuthenticate = false,
                    AutomaticChallenge = false
                });

                app.UseIdentityServer();

                app.UseStaticFiles();
            }
        }
    }
}

```

Используем сертификат для подписывания маркеров

ASP.NET Core может внедрять зависимости в Startup

Конфигурация ASP.NET Core, от которой зависит IdentityServer

Настраиваем IdentityServer

Добавляем хранящие данные в оперативной памяти версии всего, что нужно IdentityServer

Добавляем промежуточное ПО аутентификации, используемое IdentityServer

Занываем IdentityServer

```

    app.UseMvcWithDefaultRoute();
  }
}

```

Добавляем необходимую IdentityServer маршрутизацию

Теперь осталось всего лишь добавить простые реализации классов `Users`, `Clients` и `Scopes`. Начнем с класса `Users` (листинг 10.3).

Листинг 10.3. Зашитые в код описания пользователей

```

namespace IdentityServer.Configuration
{
    using System.Collections.Generic;
    using System.Security.Claims;
    using IdentityModel;
    using IdentityServer4.Services.InMemory;

    static class Users
    {
        public static List<InMemoryUser> Get()
        =>
            new List<InMemoryUser>
            {
                new InMemoryUser{
                    Subject = "818727", Username = "alice", Password = "alice",
                    Claims = new[]
                    {
                        new Claim(JwtClaimTypes.Name, "Alice Smith"),
                        new Claim(JwtClaimTypes.GivenName, "Alice"),
                        new Claim(JwtClaimTypes.FamilyName, "Smith"),
                        new Claim(JwtClaimTypes.Email, "AliceSmith@email.com"),
                        new Claim(JwtClaimTypes.EmailVerified,
                            "true", ClaimValueTypes.Boolean),
                        new Claim(JwtClaimTypes.Role, "User"),
                        new Claim(JwtClaimTypes.Id, "1", ClaimValueTypes.Integer64)
                    }
                },
                ...
            };
    }
}

```

Описание пользователя "alice" IdentityServer

Список утверждений для пользователя "alice"

Здесь будут остальные описания пользователей

Мы добавили нескольких нользователей, под именем которых сможем нозднее войти в систему. Это немного нереалистично: сама идея нрограммы лояльности заключается в как можно большем количестве зарегистрированных нользователей, так что нет никакого смысла зашивать их в код микросервиса Login. Гораздо более разумно было бы нредоставить возможность войти через разнообразные социальные медиа: Facebook, Twitter, Google и т. н. — и позволить аутентифицироваться через них тем, кто хочет зарегистрироваться или поменять свои настройки нользователей. Как уже упоминалось, IdentityServer поддерживает подобные сценарии работы, вся необходимая для их реализации информация есть в документации IdentityServer.

Далее мы опишем область действия для микросервиса Loyalty Program. Необходимо модифицировать Loyalty Program так, чтобы он нринимал только обращения, содержащие маркер с этой областью действия (листинг 10.4).

Листинг 10.4. Защиты в код описания областей действия

```

namespace IdentityServer.Configuration
{
    using System.Collections.Generic;
    using IdentityServer4.Core.Models;

    public class Scopes
    {
        public static IEnumerable<Scope> Get() =>
            new[]
            {
                // Стандартные области действия OpenIDConnect
                StandardScopes.OpenId,
                StandardScopes.ProfileAlwaysInclude,
                StandardScopes.EmailAlwaysInclude,
                new Scope
                {
                    Name = "loyalty_program_write",
                    DisplayName = "Loyalty Program write access",
                    Type = ScopeType.Resource,
                },
            };
    }
}

```

Три стандартные области действия для конечных пользователей, определяющих, что включается в идентификационные маркеры

Описание области действия для конечных точек в микросервисе LoyaltyProgram

В отличие от жестко зашитых пользователей жестко зашитые в коде области действия вполне реалистичны, по крайней мере до тех пор, пока система не достигнет определенных размеров. Ничего неправильного в том, чтобы начать реализацию с них, нет. Недостаток такого подхода состоит в следующем: при появлении нового микросервиса приходится вносить изменения в код микросервиса Login и задавать там области действия для этого микросервиса. До поры до времени это допустимо, но когда количество микросервисов достигнет определенного значения, ситуация начнет выходить из-под контроля.

Накопец, нам нужно задать *клиентов* (листинг 10.5). Клиенты в нашем случае — это микросервисы, которым необходимо обращаться к другим микросервисам. Конфигурация клиента указывает IdentityServer, какие области действия можно разрешить при запросе микросервисом маркера доступа.

Листинг 10.5. Защиты в код описания клиентов

```

namespace IdentityServer.Configuration
{
    using System.Collections.Generic;
    using IdentityServer4.Models;

    public class Clients
    {
        public static IEnumerable<Client> Get() =>
            new List<Client>
            {
                new Client
                {
                    ClientName = "API Gateway",

```

Задаем идентификатор и секретный ключ, необходимые микросервису API Gateway для запроса маркера

```

        ClientId = "api_gateway",
        ClientSecrets = new List<Secret>
        {
            new Secret("secret".Sha256())
        },
        AllowedScopes = new List<string>
        {
            "loyalty_program_write", ←
        },
        AllowedGrantTypes = GrantTypes.ClientCredentials
    },
    new Client
    {
        ClientName = "Web Client", ←
        ClientId = "web",
        RedirectUri = new List<string>
        {
            "http://localhost:5003/signin-oidc",
        },
        PostLogoutRedirectUri = new List<string>
        {
            "http://localhost:5003/",
        },
        AllowedScopes = new List<string>
        {
            "openid",
            "email",
            "profile",
        }
    }
}
};
}
}
}

```

Области действия, включаемые в маркеры доступа для микросервиса API Gateway

Клиент, предоставляющий пользователям возможность входить в систему через клиентскую часть с веб-интерфейсом

Это дает возможность микросервису API Gateway получать доступ к области действия `loyalty_program_write`, что на практике означает: он может обращаться к конечным точкам микросервиса Loyalty Program. Если микросервис API Gateway запросит у микросервиса Login разрешение на обращение к Loyalty Program, это разрешение будет ему предоставлено.

На этом создание микросервиса Login завершено. Вы можете запустить его обычным способом с помощью утилиты `dotnet`.

Реализация аутентификации с помощью промежуточного ПО IdentityServer

В этом разделе мы обеспечим пользователям возможность входить в систему. Будем считать, что микросервис API Gateway предоставляет пользователям какой-то веб-интерфейс. Им может быть, например, использующее конечные точки API Gateway приложение JavaScript, изначально загружаемое из API Gateway. Чтобы с ним работать, пользователи должны будут сначала войти в систему.

Чтобы заставить API Gateway требовать от пользователей войти в систему, необходимо внести некоторые изменения на уровне платформы ASP.NET Core, чтобы

приложение запускало аутентификацию посредством основополагающего на IdentityServer микросервиса Login. Прежде чем написать код для этого, необходимо добавить в микросервис API Gateway NuGet-пакеты `Microsoft.AspNetCore.Authentication.Cookies` и `Microsoft.AspNetCore.Authentication.OpenIdConnect`. Раздел зависимостей в файле `project.json` проекта API Gateway должен выглядеть следующим образом:

```
"dependencies": {
  "Microsoft.AspNetCore.IISPlatformHandler": "1.0.0",
  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
  "Microsoft.AspNetCore.Diagnostics": "1.0.0",
  "Microsoft.AspNetCore.Mvc": "6.0.0",
  "Microsoft.AspNetCore.Authentication.Cookies": "1.0.0",
  "Microsoft.AspNetCore.Authentication.OpenIdConnect": "1.0.0"
},
```

После добавления пакетов, поменяйте код класса `Startup` в микросервисе API Gateway на следующий (листинг 10.6).

Листинг 10.6. Аутентификация через микросервис Login

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();
        app.UseCookieAuthentication(new CookieAuthenticationOptions
        {
            AuthenticationScheme = "Cookies",
            AutomaticAuthenticate = true
        });
        var oidcOptions = new OpenIdConnectOptions
        {
            AuthenticationScheme = "oidc",
            SignInScheme = "Cookies",
            Authority = "http://localhost:5001",
            RequireHttpsMetadata = false,
            ClientId = "web",
            ResponseType = "id_token token",
            GetClaimsFromUserInfoEndpoint = true,
            SaveTokens = true
        };
        oidcOptions.Scope.Clear();
        oidcOptions.Scope.Add("openid");
        oidcOptions.Scope.Add("profile");
        oidcOptions.Scope.Add("api1");

        app.UseOwin(buildFunc => buildFunc.UseNancy());
    }
}
```

Меняем настройки платформы ASP.NET Core так, чтобы использовался протокол OpenIdConnect →
 Идентифицируем данный веб-клиент для микросервиса Login →
 Указываем ссылку на микросервис Login как центр аутентификации →
 Меняем настройки платформы ASP.NET Core для чтения и записи cookie-файла входа в систему →

Вот и все, что требуется. Теперь еще не вошедших в систему пользователей перенаправляют на страницу входа в микросервисе Login, предоставляемую в готовом виде IdentityServer. Вы можете войти в него под именами введенных ранее в код

пользователей, например *alice*. Все выполняемые после входа пользователя в систему запросы будут включать cookie-файл входа, проверяемый микросервисом API Gateway. Пока этот cookie-файл действителен, микросервис API Gateway будет принимать запросы, а когда он перестанет быть действительным, пользователь вновь будет перенаправлен на микросервис Login.

Реализация авторизации при взаимодействии между микросервисами с помощью IdentityServer и промежуточного ПО

Следующее, что нужно сделать после реализации аутентификации пользователей, — обезопасить взаимодействие между микросервисами. Мы воспользуемся микросервисом Login, чтобы гарантировать возможность взаимодействия только тех микросервисов, которым оно разрешено. Для этого нужно будет реализовать показанную на рис. 10.8 схему взаимодействия: микросервис API Gateway сначала запрашивает у микросервиса Login маркер доступа, после чего включает его в запрос к микросервису Loyalty Program.

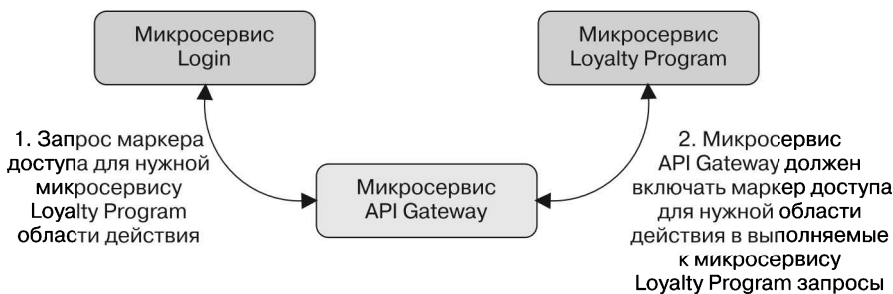


Рис. 10.8. Использование маркеров доступа для авторизации запросов между микросервисами

Микросервис Loyalty Program проверяет наличие в каждом запросе от микросервиса API Gateway маркера доступа, а также наличие в этом маркере доступа соответствующей области действия. Для реализации этой возможности нужно выполнить следующее.

1. Модифицировать микросервис API Gateway, чтобы он запрашивал маркер у микросервиса Login перед каждым запросом к Loyalty Program и помещал маркер в заголовок **Authorization** всех запросов к Loyalty Program.
2. Модифицировать микросервис Loyalty Program, чтобы он требовал маркер с соответствующей областью действия при каждом входящем запросе. Мы сделаем это в промежуточном ПО. Если область действия отсутствует, промежуточное ПО будет прерывать выполнение копвейера и отправлять вызывающей стороне ответ с кодом состояния **403 Forbidden** (403 Запрещено). Если же область действия указана, запрос будет обрабатываться обычным образом.

Вначале необходимо запрашивать маркер у микросервиса Login. Мы будем запрашивать его в создаваемой в главе 8 фабрике `HttpClientFactory`, поскольку хотим вставлять его в каждый запрос к микросервису `Loyalty Program`. Напомню, что при стандартных настройках `HttpClientFactory` — маленький класс, способный создавать готовые к использованию для выполнения запросов объекты `HttpClient`. В главе 8 мы использовали `HttpClientFactory` для добавления во все запросы маркеров корреляции, теперь же расширим его, чтобы добавлять во все запросы маркеры доступа (листинг 10.7).

Листинг 10.7. Добавление в запросы маркеров доступа и маркеров корреляции

```
public interface IHttpHttpClientFactory
{
    Task<HttpClient> Create(Uri uri, string scope);
}

public class HttpClientFactory : IHttpHttpClientFactory
{
    private readonly TokenClient tokenClient;
    private readonly string correlationToken;

    public HttpClientFactory(string correlationToken)
    {
        this.correlationToken = correlationToken;
        this.tokenClient = new TokenClient(
            "http://localhost:5001/connect/token",
            "api_gateway",
            "secret");
    }

    public async Task<HttpClient> Create(Uri uri, string scope)
    {
        var response = await
            this.tokenClient
                .RequestClientCredentialsAsync(scope)
                .ConfigureAwait(false);
        var client = new HttpClient() {BaseAddress = uri};
        client
            .DefaultRequestHeaders
            .Authorization =
                new AuthenticationHeaderValue("Bearer", response.AccessToken);
        client
            .DefaultRequestHeaders
            .Add("Correlation-Token", this.correlationToken);
        return client;
    }
}
```

Возвращает Task, чтобы обеспечить асинхронность реализации

Подготовка к запросу маркера доступа у микросервиса Login

Запрашиваем маркер доступа у микросервиса Login

Добавляем маркер доступа в исходящие запросы

После этого расширения `HttpClientFactory` во всех исходящих запросах будут присутствовать маркеры доступа, если не забывать использовать `HttpClientFactory` для создания объектов `HttpClient`. В главе 8 мы также добавили `HttpClientFactory` в контейнер DI фреймворка Nancу, так что любой код приложения сможет легко получить ссылку на `HttpClientFactory` и воспользоваться ею.

Теперь обратим внимание на микросервис *Loyalty Program*, который должен принимать только запросы с действительным маркером доступа. Для этого необходимо только добавить в *Loyalty Program* библиотеку *LibOwin*, как описывалось в главе 8, и NuGet-пакет *Microsoft.AspNetCore.Authentication.JwtBearer*. Далее нужно будет добавить пемного копфигурационного кода в класс *Startup* микросервиса *Loyalty Program* и элемент промежуточного ПО — в копвейер OWIN. Добавление копфигурационного кода меняет класс *Startup* следующим образом (листинг 10.8).

Листинг 10.8. Чтение маркера носителя из входящих запросов

```
namespace LoyaltyProgram
{
    using System.Collections.Generic;
    using System.IdentityModel.Tokens.Jwt;
    using System.Threading.Tasks;
    using LibOwin;
    using Microsoft.AspNetCore.Builder;
    using Nancy.Owin;

    public class Startup
    {
        public void Configure(IApplicationBuilder app)
        {
            JwtSecurityTokenHandler.DefaultInboundClaimTypeMap =
                new Dictionary<string, string>();

            app.UseJwtBearerAuthentication(options =>
            {
                options.Authority = "http://localhost:5001";
                options.RequireHttpsMetadata = false;
                options.Audience = "http://localhost:5001/resources";
                options.AutomaticAuthenticate = true;
            });

            app.UseOwin(buildFunc => buildFunc.UseNancy());
        }
    }
}
```

Меняем настройки ASP.NET Core для чтения маркера носителя из входящих запросов и использования микросервиса Login для его проверки

Этот код заставляет ASP.NET Core читать маркер посетителя из заголовка *Authorization* входящих запросов, что идеально соответствует пастройкам *HttpClientFactory* в микросервисе *API Gateway*. ASP.NET Core будет использовать этот маркер для создания объекта *ClaimsPrincipal*, к которому мы сможем обращаться через окружение OWIN. Утверждения из этого *ClaimsPrincipal* представляют собой утверждения, которые микросервис *Login* задает при запросе микросервисом *API Gateway* маркера доступа, включающего допустимые для *API Gateway* области действия.

Последнее, что нужно сделать, чтобы микросервис *Loyalty Program* припимал запросы только с правильной областью действия, — добавить элемент промежуточного ПО для проверки субъекта па паличие требуемой области действия. С помощью промежуточного ПО OWIN, написанного в лямбда-стиле, расширим копвейер OWIN еще одним элементом промежуточного ПО (листинг 10.9).

Листинг 10.9. Конвейер OWIN с промежуточным ПО для проверки области действия

```

Проверяет наличие
требуемой области действия

app.UseOwin(buildFunc =>
{
    buildFunc(next => env =>
    {
        var ctx = new OwinContext(env);
        var principal = ctx.Request.User;
        if (principal.HasClaim("scope", "loyalty_program_write"))
            return next(env);
        ctx.Response.StatusCode = 403;
        return Task.FromResult(0);
    });
    buildFunc.UseNancy();
});

```

Новый элемент промежуточного ПО

Если область действия есть, продолжаем выполнение конвейера

Если области действия нет, возвращаем ответ с кодом состояния 403 Forbidden (403 Запрещено)

Теперь микросервис Loyalty Program будет возвращать ответ с кодом состояния 403 Forbidden (403 Запрещено), если маркер доступа в поступившем запросе не содержит пужпой области действия. Реализация подобной схемы во всех микросервисах дает возможность управлять тем, какие микросервисы могут взаимодействовать между собой.

Реализация авторизации пользователей в модулях Nancy

После обеспечения пользователям возможности входа в систему и реализации управления тем, какие микросервисы могут взаимодействовать друг с другом, нам осталось реализовать только одно — безопасную отправку идентификационных данных пользователя от одного микросервиса другому для выполнения авторизации в пужном микросервисе. План ее реализации выглядит следующим образом.

1. В микросервисе API Gateway воспользоваться `HttpClientFactory` для добавления еще одного заголовка под пазванием `pos-end-user` во все исходящие запросы. Он будет содержать идентификатор пользователя.
2. В микросервисе Loyalty Program воспользоваться элементом промежуточного ПО для чтения идентификатора пользователя из заголовка `pos-end-user`.
3. В микросервисе Loyalty Program воспользоваться загрузчиком Nancy для присваивания идентификатора пользователя свойству `CurrentUser` контекста Nancy.

Мы реализуем эти три шага в следующих разделах.

Добавление идентификатора пользователя в запросы

Сначала нужно добиться того, чтобы `HttpClientFactory` добавлял еще один заголовок. Мы пазовем его `pos-end-user`, и он будет содержать идентификатор конечного пользователя в виде маркера. Этот маркер возвращается микросервисом Login после выполнения аутентификации и присутствует в виде утверждения `id_token`. Мы уже

создали экземпляр `HttpClientFactory` для каждого запроса в загрузчике, но теперь нужно передавать в него маркер ID вместе с уже передаваемым туда маркером корреляции (листинг 10.10).

Листинг 10.10. Чтение маркера ID в загрузчике

```
using LibOwin;

public class Bootstrapper : DefaultNancyBootstrapper
{
    ...

    protected override void RequestStartup(
        TinyIoCContainer container,
        IPipelines pipelines,
        NancyContext context)
    {
        base.RequestStartup(container, pipelines, context);
        var correlationToken =
            context.GetOwinEnvironment()["correlationToken"] as string;
        var principal =
            context.GetOwinEnvironment()[OwinConstants.RequestUser]
            as ClaimsPrincipal;
        var idToken = principal.FindFirst("id_token");
        container.Register<IHttpClientFactory>(
            new HttpClientFactory(idToken, correlationToken));
    }
}
```

Читаем пользователя из окружения OWIN

Читаем маркер ID

Передаем маркер ID в фабрику

Из-за этого придется внести изменения в `HttpClientFactory`, поскольку теперь она должна принимать маркер ID в виде аргумента конструктора. Модифицированный класс `HttpClientFactory` приведен в листинге 10.11.

Листинг 10.11. Класс `HttpClientFactory`, принимающий маркер ID в виде аргумента конструктора

```
public class HttpClientFactory : IHttpClientFactory
{
    private readonly TokenClient tokenClient;
    private readonly string correlationToken;
    private readonly string idToken;

    public HttpClientFactory(
        string tokenUrl,
        string correlationToken,
        string idToken)
    {
        this.tokenClient = new TokenClient(tokenUrl, clientName, clientSecret);
        this.correlationToken = correlationToken;
        this.idToken = idToken;
        this.tokenClient = new TokenClient(
            "http://localhost:5001/connect/token",
            "api_gateway",
```

```

        "secret"));
    }

    public async Task<HttpClient> Create(Uri uri)
    {
        var response = await
            this.tokenClient
                .RequestClientCredentialsAsync("loyalty_program_write")
                .ConfigureAwait(false);
        var client = new HttpClient() { BaseAddress = uri };
        client
            .DefaultRequestHeaders
            .Authorization =
                new AuthenticationHeaderValue("Bearer", response.AccessToken);
        client
            .DefaultRequestHeaders
            .Add("Correlation-Token", this.correlationToken);
        client
            .DefaultRequestHeaders
            .Add("pos-end-user", this.idToken);
        return client;
    }
}

```

Добавляем маркер ID в заголовок

Теперь во всех исходящих из микросервиса API Gateway запросах есть заголовок `pos-end-user`, содержащий маркер ID.

Чтение идентификатора пользователя из запросов

Следующий шаг — чтение микросервисом Loyalty Program маркера ID из заголовка, которое мы реализуем с помощью добавленного в копвейер OWIN микросервиса Loyalty Program элемента промежуточного ПО (листинг 10.12).

Листинг 10.12. Чтение маркера ID в элементе промежуточного ПО

```

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        ...

        app.UseOwin(buildFunc =>
        {
            ...
            buildFunc(next => env =>
            {
                var ctx = new OwinContext(env);
                if (ctx.Request.Headers.ContainsKey("pos-end-user"))
                {
                    var tokenHandler = new JwtSecurityTokenHandler();
                    SecurityToken token;
                    var userPrincipal =
                        tokenHandler.ValidateToken(
                            ctx.Request.Headers["pos-end-user"],

```

Проверяем, имеется ли заголовок `pos-end-user`

Читаем и проверяем маркер ID

```

        new TokenValidationParameters(),
        out token);
    ctx.Set("pos-end-user", userPrincipal);
}
return next(env);
});
});
}
}

```

Создаем пользователя на основе маркера ID и добавляем его в окружение OWIN

Этот код читает маркер ID из заголовка запроса, проверяет его, создавая в качестве побочного результата объект `ClaimsPrincipal`, содержащий все утверждения из маркера ID, то есть все утверждения, которые возвращает микросервис Login при входе пользователя в систему.

Присваиваем значение идентификатора пользователя свойству `CurrentUser` Nancy

После того как объект `ClaimsPrincipal` окажется в окружении OWIN, можно будет получить его в загрузчике микросервиса `Loyalty Program` и присвоить его свойству `CurrentUser` из `NancyContext` (листинг 10.13).

Листинг 10.13. Присваивание созданного на основе маркера ID идентификатора пользователя свойству из `NancyContext`

```

public class Bootstrapper : DefaultNancyBootstrapper
{
    ...
    protected override void RequestStartup(
        TinyIoCContainer container,
        IPipelines pipelines,
        NancyContext context)
    {
        base.RequestStartup(container, pipelines, context);
        context.CurrentUser =
            context.GetOwinEnvironment()["pos-end-user"] as ClaimsPrincipal;
    }
}

```

Nancy вызывает его при каждом входящем запросе

Присваиваем идентификатор пользователя из маркера ID в пользовательском HTTP-заголовке свойству «текущий пользователь» из контекста Nancy

Фреймворк Nancy предоставляет несколько удобных методов, которыми можно воспользоваться для выполнения авторизации в модулях Nancy. Допустим, нам нужно, чтобы доступ к конечным точкам в модуле предоставлялся только пользователям с определенным утверждением. Для этого можно применить метод `RequiresClaims` вот так:

```

public class UsersModule : NancyModule
{
    public SecuredModule() : base("/secret-stuff")
    {

```

```

    this.RequiresClaims("my claim");

    Get("/", ...);
    Post("/", ...);
}
}

```

Метод `RequiresClaims` можно использовать также на уровне обработчиков конечных точек путем его вызова в обработчике. В этом случае не нужно будет требовать наличия в модуле какого-то конкретного утверждения пользователя, но потребуется обеспечить редактирование пользователями только их собственных настроек, относящихся к интересам. Так что мы будем сверять в обработчиках `POST` и `PUT` в `UserModule` идентификатор пользователя с идентификатором, приведенным в URL запроса (листинг 10.14).

Листинг 10.14. Проверяем утверждения, чтобы обеспечить редактирование пользователями только их собственных настроек, относящихся к интересам

```

public class UsersModule : NancyModule
{
    public SecuredModule() : base("/secret-stuff")
    {
        ...
        Put("/{userId:int}", parameters =>
        {
            int loggedInUserId;
            int.TryParse(
                this.Context
                    .CurrentUser
                    .Claims.FirstOrDefault(c => c.Type.StartsWith("id"))
                    ?.Value.Split(':').Last() ?? "",
                out loggedInUserId);
            int userId = parameters.userId;
            if (loggedInUserId != userId)
                return HttpStatusCode.Forbidden;
            var updatedUser = this.Bind<LoyaltyProgramUser>();
            registeredUsers[userId] = updatedUser;
            return updatedUser;
        });
    }
}

```

Пытаемся прочесть утверждение id из идентификатора пользователя

Сравниваем идентификатор выполнившего вход в систему пользователя с ID пользователя из URL

Теперь пользователи могут менять только свои собственные настройки, относящиеся к интересам.

На этом реализацию требований безопасности можно считать завершенной. Мы выполнили эти требования с помощью `IdentityServer`, которым воспользовались для реализации микросервиса `Login`. Микросервис `API Gateway` использует микросервис `Login` для аутентификации, причем именно `Login` выполняет как аутентификацию, так и авторизацию обращений `API Gateway` к микросервису `Loyalty Program`. Наконец, мы передаем маркер `ID` от микросервиса к микросервису, чтобы они оба могли использовать идентификационные данные выполнившего вход в систему пользователя.

10.3. Резюме

- ❑ Пользователей необходимо аутентифицировать на периферии системы. Инициировать аутентификацию должен микросервис, который первым получает запрос пользователя.
- ❑ Авторизация в системе микросервисов должна происходить в микросервисе, которому принадлежат запрашиваемые данные или который осуществляет требуемое для запроса действие. Авторизация — часть бизнес-правил, относящихся к определенной бизнес-возможности. Поскольку микросервисы, как сказано в главе 3, проектируются в соответствии с бизнес-возможностями, то отвечающий за какую-либо бизнес-возможность микросервис должен отвечать и за любую относящуюся к этой бизнес-возможности авторизацию.
- ❑ Принцип эшелонированной безопасности требует обдумывания уровня доверия между микросервисами. В некоторых системах вполне допустимо, чтобы микросервисы доверяли запросам от других микросервисов, а в других системах — нет.
- ❑ Должен соблюдаться компромисс между удобством высокого уровня доверия в микросервисах и безопасностью более низкого уровня доверия.
- ❑ Можно ввести в систему микросервис Login, который отвечал бы за всю аутентификацию, включая аутентификацию конечных пользователей и обращений одних микросервисов к другим.
- ❑ Микросервис Login используется для контроля за тем, какие микросервисы могут взаимодействовать друг с другом, посредством настройки областей действия и того, какие микросервисы какие области действия получают.
- ❑ Реализовать микросервис Login можно с помощью IdentityServer.
- ❑ Можно настроить любой микросервис так, чтобы он использовал микросервис Login для аутентификации конечных пользователей.
- ❑ Для выполнения безопасных обращений между микросервисами можно включать в каждый запрос получаемый от микросервиса Login маркер и выполнять (в промежуточном ПО получающего запрос микросервиса) проверку наличия в этом маркере требуемой области действия.
- ❑ Полученный от микросервиса Login маркер ID можно использовать для более безопасной передачи идентификатора пользователя, чем простое включение идентификатора пользователя в URL запроса.
- ❑ Можно читать маркер ID из входящих запросов и присваивать его свойству контекста Nansu, тем самым делая его доступным модулям Nansu в качестве текущего пользователя. В дальнейшем его можно использовать для авторизации.

11

Создание платформы многоразового кода для микросервисов

В этой главе:

- ❑ ускорение создания микросервисов с помощью платформы многоразового кода;
- ❑ компоненты платформы многоразового кода;
- ❑ упаковка повторно используемого промежуточного ПО с помощью NuGet;
- ❑ создание платформы многоразового кода из нескольких пакетов NuGet.

Система микросервисов может включать множество микросервисов. Вы будете часто создавать новые микросервисы, добавляя новые возможности в систему или заменяя существующие микросервисы. Хотелось бы создавать их быстро, но при этом включать в них весь тот код, благодаря которому они хорошо работают в производственной среде, то есть написанный в нескольких предыдущих главах инфраструктурный код. В этой главе мы создадим платформу, состоящую из нескольких пакетов NuGet, которая обеспечит возможность быстрого создания новых, надежно работающих микросервисов.

11.1. Создание нового микросервиса должно быть быстрым и легким

В главе 1 я перечислил несколько отличительных признаков микросервисов, в том числе следующий: *микросервис отвечает за конкретную функциональную возможность*. Я объяснял, что этот отличительный признак — разовидность принципа единственной обязанности. Серьезный подход к этому вопросу приводит к тому, что в системе получается большое число микросервисов. И по мере развития системы вы будете создавать новые микросервисы очень часто, например, когда системе понадобится новая функциональность конечного пользователя, а также когда вы станете лучше понимать предметную область.

Как говорилось в главе 3, скорее всего, с первого раза у вас не получится правильно определить области действия всех микросервисов, и при возникновении обобщающих сомнений следует создавать микросервисы с чуть большей, чем нужно, областью действия. Это также приводит к созданию новых микросервисов по ходу дела: когда вы начинаете понимать предметную область лучше, сферы ответствен-

ности различных функциональных возможностей становятся более ясными и иногда оказывается, что какой-то микросервис следует разбить на два.

Еще один отличительный признак микросервисов гласит: *микросервис можно легко заменить*. Смысл в том, что микросервис можно быстро полностью переписать, если его реализация становится непригодной. Код может выйти из-под контроля, или припавшие вами в самом начале проектные и технологические решения оказываются не подходящими для растущей нагрузки на данный микросервис. Какова бы ни была причина, иногда приходится заменять существующий микросервис новым.

Основной вывод: при работе с системой микросервисов вам часто придется создавать новые микросервисы. Если вы не станете этого делать, сервисы вашей системы медленно, но верно вырастут, их границы станут менее четкими, и в результате вы утратите основные достоинства системы микросервисов — гибкость и быстроту разработки. Нам необходим способ сделать разработку новых микросервисов быстрой и удобной, чтобы вы не старались подсознательно без этого обойтись.

11.2. Создание платформы многоразового кода для микросервисов

В главах 8 и 9 подчеркивалось, что микросервисы должны быть «законопослушными гражданами» в производственной среде. Они должны предоставлять сведения о своем состоянии путем журналирования, давать возможность выполнять мониторинг, а также соответствовать принятым стандартам безопасности системы, например позволять авторизацию запросов между микросервисами посредством распределения областей действия микросервисом входа в систему, как обсуждалось и было реализовано в главе 10. Следовательно, для добавления нового микросервиса недостаточно просто создать новый проект и добавить в него NuGet Nancy.

Именно поэтому я рекомендую создать стандартную платформу для микросервисов, которую можно было бы использовать в системе для всех микросервисов, основанных на платформе .NET. Как показано на рис. 11.1, платформа встраивается во все микросервисы. Во второй половине главы мы создадим подобную платформу, включающую:

- ❑ промежуточное ПО для мониторинга и журналирования;
- ❑ промежуточное ПО для обеспечения безопасности;
- ❑ компоненты, поддерживающие отправку HTTP-запросов в соответствии со стандартами вашей системы микросервисов.

Таковы сферы платформы микросервисов, которые я выбрал для рассмотрения в данной книге, но в вашей собственной платформе микросервисов может попадаться и другое. Я постарался включить наиболее распространенные компоненты. В частности, считаю, что подобные платформы должны включать мониторинг, маркеры корреляции, журналирование запросов и ответов, а также элементы журналирования показателей производительности. Они не обязательно должны выглядеть именно так, как показано в нашей книге, — вы можете, например, предпочесть другой формат маркеров корреляции или фреймворк журналирования. Ваше промежуточное ПО авторизации будет привязано к выбранному вами подходу

к безопасности, но если вы решите внедрить меры безопасности для взаимодействия между микросервисами, то необходимые для этого вспомогательные библиотеки следует включить в платформу микросервисов.



Рис. 11.1. Платформа для микросервисов — набор пакетов, реализующих техническую функциональность, сквозную для всех микросервисов. Платформа встраивается во все микросервисы и становится их частью

Обратная сторона этой медали: следует ли включать все больше и больше в платформу для микросервисов? Однозначного ответа для всех случаев не существует, но добавлять новые элементы следует с осторожностью: всякий раз, когда вы что-то добавляете в платформу, она становится еще чуть-чуть больше и тяжелее. Более того, что происходит с работающими на различных версиях платформы микросервисами при добавлении того, что все они, как предполагается, должны включать? Смогут ли они справиться с этим? Сможет ли справиться с этим сопутствующая системе микросервисов инфраструктура? Если платформа вырастет в нечто требующее единообразия по всем микросервисам, то при частых изменениях она вместо подспорья превратится в пастоящее узкое место. Во-первых, платформу придется обновлять в базах кода всех микросервисов, после чего пужпо будет выполнить повторное развертывание каждого из них. Если микросервисов много, это потребует немалых усилий, причем с каждым новым микросервисом все больше.

В идеале платформа микросервисов должна включать только технические возможности, действительно сквозные для микросервисов, которые можно обновлять пошагово, то есть в одном микросервисе за раз на протяжении определенного отрезка времени. Это приводит к значительному снижению плапки создания нового микросервиса: пужпо гораздо меньше трудозатрат и меньше подробной информации о сквозной технической функциональности.

11.3. Упаковка и совместное использование промежуточного ПО с помощью NuGet

Используем NuGet для нашей платформы

NuGet — не только набор утилит для установки пакетов, но и формат этих пакетов. До сих пор мы устанавливали только существующие пакеты NuGet, свободно доступные в ленте пакетов на сайте nuget.org. Но возможности системы управления пакетами NuGet не ограничиваются установкой общедоступных пакетов. Как вы увидите в подразделе «Создание пакета с REST-фабрикой клиентов» текущего раздела, можно устанавливать пакеты и из своей собственной ленты пакетов. Можно также создавать собственные пакеты NuGet, как вы также увидите в подразделе «Создание пакета с промежуточным ПО журналирования и мониторинга» этого раздела, и размещать их в своей ленте NuGet.

Теперь обратим внимание на код, необходимый для создания платформы, которую можно будет многократно использовать в микросервисах. Мы уже создали всю необходимую для платформы функциональность, по каждый раз делали это только в каком-то одном микросервисе. Теперь извлечем соответствующий код из микросервисов, в которых его создали, и поместим его в пакеты, которые можно было бы легко установить и применять в новых микросервисах. Мы будем использовать систему управления пакетами NuGet, поскольку с ее помощью можно легко создавать пакеты, подходящие для установки и применения нашими микросервисами. Вы уже неоднократно использовали пакеты NuGet в своих микросервисах, так что это должно было стать частью процесса разработки, и пакеты для платформы многоразового кода для микросервисов отлично впишутся в него.

Мы создадим платформу из следующих частей:

- ❑ промежуточного ПО мониторинга из главы 9;
- ❑ промежуточного ПО маркеров корреляции из главы 9;
- ❑ промежуточного ПО журналирования запросов и ответов из главы 9;
- ❑ промежуточного ПО журналирования показателей производительности из главы 9;
- ❑ промежуточного ПО авторизации из главы 10, которое проверяет маркер запроса на наличие пужпой области действия;
- ❑ промежуточного ПО из главы 10, читающего идентификационные маркеры пользователей из входящих запросов;
- ❑ фабрики `HttpClientFactory` из глав 9 и 10;
- ❑ автоматической регистрации фабрики `HttpClientFactory` в контейнере Nancy, которую мы реализуем в подразделе «Создание пакета с REST-фабрикой клиентов».

Как показано на рис. 11.2, платформа будет состоять из следующих пакетов NuGet:

- ❑ `MicroserviceNET.Logging` — промежуточного ПО мониторинга, промежуточного ПО маркеров корреляции, промежуточного ПО журналирования запросов и ответов и промежуточного ПО журналирования показателей производительности;

- ❑ `MicroserviceNET.Auth` — промежуточного ПО авторизации и промежуточного ПО для чтения идентификаторов конечных пользователей;
- ❑ `MicroserviceNET.Platform` — фабрики `HttpClientFactory` и автоматической регистрации `HttpClientFactory` в контейнере Nancy. Этот пакет зависит от двух остальных.



Рис. 11.2. Платформа микросервисов состоит из трех пакетов, каждый из которых содержит код для нескольких элементов сквозной технической функциональности

В ваши микросервисы необходимо будет всего лишь добавить пакет `MicroserviceNET.Platform` и немного кода инициализации для использования и настройки платформы микросервисов и всей ее функциональности. После установки и настройки платформы конвейеры микросервисов будут выглядеть так, как показано на рис. 11.3.

В следующих разделах мы создадим по очереди три пакета NuGet.

Создание пакета с промежуточным ПО для журналирования и мониторинга

В этом разделе мы для создания NuGet-пакета `MicroserviceNET.Logging` выполним следующие действия.

1. Извлечем созданное в главе 9 промежуточное ПО мониторинга, маркеров корреляции, журналирования запросов/ответов и журналирования показателей производительности из микросервиса `Shopping Cart` в библиотеку классов под названием `MicroserviceNET.Logging`.
2. Добавим удобный метод, упрощающий добавление в конвейер OWIN промежуточного ПО мониторинга и журналирования.
3. Создадим пакет NuGet на основе библиотеки `MicroserviceNET.Logging`.

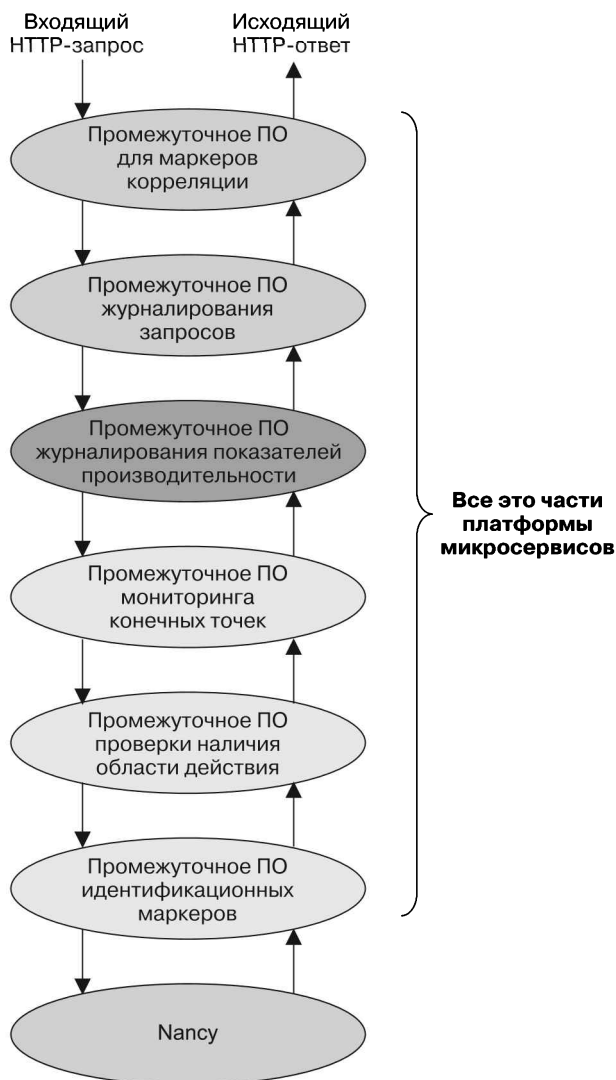


Рис. 11.3. После установки и настройки в микросервисе платформы микросервисов формируется конвейер, обеспечивающий мониторинг этого микросервиса, должное журналирование и выполнение тех проверок безопасности, которые вы сочтете необходимыми

На первом шаге создайте библиотеку классов .NET Core и назовите ее `MicroserviceNET.Logging`. Сделать это можно с помощью вашего IDE или утилиты Yeoman. Далее добавьте элементы промежуточного ПО для мониторинга и журналирования, разработанные в главе 9, в библиотеку `MicroserviceNET.Logging`. Я продублирую здесь код промежуточного ПО, но за подробностями обратитесь к главе 9.

Добавьте в проект `MicroserviceNET.Logging` файл `MonitoringMiddleware.cs` и поместите в него следующий код (листинг 11.1). Он реагирует на запросы к конечным точкам `/_monitor/shallow` и `/_monitor/deep`.

Листинг 11.1. Промежуточное ПО для мониторинга, разработанное в главе 9

```

namespace MicroserviceNET.Logging
{
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using LibOwin;
    using AppFunc =  Сигнатура OWIN AppFunc
        System.Func<
            System.Collections.Generic.IDictionary<string, object>,
            System.Threading.Tasks.Task>;

    public class MonitoringMiddleware
    {
        private AppFunc next;
        private Func<Task<bool>> healthCheck;

        private static readonly PathString monitorPath = 
            new PathString("/_monitor");
        private static readonly PathString monitorShallowPath =
            new PathString("/_monitor/shallow");
        private static readonly PathString monitorDeepPath =
            new PathString("/_monitor/deep");

        public MonitoringMiddleware(
            AppFunc next,
            Func<Task<bool>> healthCheck)
        {
            this.next = next;
            this.healthCheck = healthCheck;
        }

        public Task Invoke(IDictionary<string, object> env) 
        {
            var context = new OwinContext(env);
            if (context.Request.Path.StartsWithSegments(monitorPath)) 
                return HandleMonitorEndpoint(context);
            else
                return this.next(env);
        }

        private Task HandleMonitorEndpoint(OwinContext context)
        {
            if (context.Request.Path.StartsWithSegments(monitorShallowPath))
                return ShallowEndpoint(context);
            else if (context.Request.Path.StartsWithSegments(monitorDeepPath))
                return DeepEndpoint(context);
            return Task.FromResult(0);
        }

        private async Task DeepEndpoint(OwinContext context)
        {
            if (await this.healthCheck()) 
                context.Response.StatusCode = 204;
            else

```

Пути, на обращение к которым должно реагировать это промежуточное ПО

Реализация AppFunc промежуточного ПО мониторинга, которую можно добавить в конвейер OWIN

Проверяем, предназначен ли входящий запрос для конечной точки мониторинга

Выполняем учитывающую специфику микросервиса проверку его состояния в конечной точке /_monitor/deep

```

        context.Response.StatusCode = 503;
    }

    private Task ShallowEndpoint(OwinContext context)
    {
        context.Response.StatusCode = 204;
        return Task.FromResult(0);
    }
}

```

Всегда возвращаем ответ об успехе в конечной точке /_monitor/shallow

Далее добавьте в проект файл `LoggingMiddleware.cs`. Вставьте в него следующее промежуточное ПО журналирования из главы 9 (листинг 11.2): журналирования запросов и ответов, показателей производительности, глобальных ошибок, а также создания и чтения маркеров корреляции.

Листинг 11.2. Промежуточное ПО журналирования из главы 9

```

namespace MicroserviceNET.Logging
{
    using System;
    using System.Diagnostics;
    using LibOwin;
    using Serilog;
    using Serilog.Context;

    using AppFunc =  Сигнатура OWIN AppFunc
        System.Func<
            System.Collections.Generic.IDictionary<string, object>,
            System.Threading.Tasks.Task>;

    public class RequestLogging
    {
        public static AppFunc Middleware(AppFunc next, ILogger log) 
        {
            return async env =>
            {
                var owinContext = new OwinContext(env);
                log.Information(
                    "Incoming request: {@Method}, {@Path}, {@Headers}",
                    owinContext.Request.Method,
                    owinContext.Request.Path,
                    owinContext.Request.Headers);
                await next(env);
                log.Information(
                    "Outgoing response: {@StatusCode}, {@Headers}",
                    owinContext.Response.StatusCode,
                    owinContext.Response.Headers);
            };
        }
    }

    public class PerformanceLogging
    {
        public static AppFunc Middleware(AppFunc next, ILogger log) 

```

Промежуточное ПО журналирования запросов и ответов, реализованное в лямбда-стиле

Журналируем длительность всех запросов в элементе промежуточного ПО, реализованном в лямбда-стиле

```

{
    return async env =>
    {
        var stopWatch = new Stopwatch();
        stopWatch.Start();
        await next(env);
        stopWatch.Stop();
        var owinContext = new OwinContext(env);
        log.Information(
            "Request: {@Method} {@Path} executed in {RequestTime:000} ms",
            owinContext.Request.Method, owinContext.Request.Path,
            stopWatch.ElapsedMilliseconds);
    };
}
}

public class CorrelationToken ← Промежуточное ПО добавления маркера
                               корреляции в контекст журнала
                               для всех запросов
{
    public static AppFunc Middleware(AppFunc next)
    {
        return async env =>
        {
            Guid correlationToken;
            var owinContext = new OwinContext(env);
            if (!(owinContext.Request.Headers["Correlation-Token"] != null
                && Guid.TryParse(owinContext.Request.Headers["Correlation-
                ➤ Token"],
                    out correlationToken)))
                correlationToken = Guid.NewGuid();

            owinContext.Set("correlationToken", correlationToken.ToString());
            using (LogContext.PushProperty("CorrelationToken", correlationToken))
                await next(env);
        };
    }
}

public class GlobalErrorLogging ← Промежуточное ПО для перехвата
                                  и журналирования всех исключений,
                                  которые в противном случае
                                  оказались бы необработанными
{
    public static AppFunc Middleware(AppFunc next, ILogger log)
    {
        return async env =>
        {
            try
            {
                await next(env);
            }
            catch (Exception ex)
            {
                log.Error(ex, "Unhandled exception");
            }
        };
    }
}
}
}

```

Мы извлекли весь код промежуточного ПО мониторинга и журналирования из микросервиса Shopping Cart и поместили его в библиотеку классов. После упаковки его можно будет с легкостью использовать в микросервисах: достаточно установить этот пакет и подключить все элементы промежуточного ПО в конвейер OWIN. Мы уже делали это в микросервисе Shopping Cart, где это выглядело следующим образом (листинг 11.3).

Листинг 11.3. Формирование конвейера OWIN в микросервисе Shopping Cart

```
app.UseOwin(buildFunc =>
{
    buildFunc(next => GlobalErrorLogging.Middleware(next, log));
    buildFunc(next => CorrelationToken.Middleware(next));
    buildFunc(next => RequestLogging.Middleware(next, log));
    buildFunc(next => PerformanceLogging.Middleware(next, log));
    buildFunc(next => new MonitoringMiddleware(next, HealthCheck).Invoke);
    buildFunc.UseNancy(opt => opt.Bootstrapper = new Bootstrapper(log));
});
```

Как можете видеть, для реализации всего журналирования и мониторинга нужно добавить пять элементов промежуточного ПО в конвейер OWIN, причем сделать это в правильной последовательности. Для упрощения этого процесса мы создадим вспомогательный метод, который будет делать это вместо нас. Добавьте в проект `MicroserviceNET.Logging` файл `BuildFuncExtensions.cs` и поместите в него следующий код (листинг 11.4).

Листинг 11.4. Добавление промежуточного ПО мониторинга и журналирования в конвейер

```
namespace MicroserviceNET.Logging
{
    using System;
    using System.Threading.Tasks;
    using Serilog;

    using BuildFunc = System.Action<System.Func<
        System.Func<
            System.Collections.Generic.IDictionary<string, object>,
            System.Threading.Tasks.Task>,
            System.Func<
                System.Collections.Generic.IDictionary<string, object>,
                System.Threading.Tasks.Task>
            >>>;

    public static class BuildFuncExtensions
    {
        public static BuildFunc UseMonitoringAndLogging(
            this BuildFunc buildFunc,
            ILogger log,
            Func<Task<bool>> healthCheck)
        {
            buildFunc(next => GlobalErrorLogging.Middleware(next, log));
            buildFunc(next => CorrelationToken.Middleware(next));
        }
    }
}
```

Несколько устрашающая BuildFunc OWIN

Метод расширения BuildFunc

```

    buildFunc(next => RequestLogging.Middleware(next, log));
    buildFunc(next => PerformanceLogging.Middleware(next, log));
    buildFunc(next => new MonitoringMiddleware(next, healthCheck).Invoke);
    return buildFunc;
  }
}

```

Возвращаем BuildFunc, чтобы можно было организовывать вызовы расширений BuildFunc цепочкой

Мы добавили метод расширения, в котором все элементы промежуточного ПО добавляются в копвейер OWIN с помощью BuildFunc. Этот метод принимает па входе два аргумента, которые должен передавать использующий его микросервис. Теперь для установки промежуточного ПО мониторинга и журналирования можно выполнить одип-единственный вызов метода расширения UseMonitoringAndLogging в классе Startup микросервиса, как показано в листинге 11.5.

Листинг 11.5. Использование метода UseMonitoringAndLogging в классе Startup

```

public void Configure(IApplicationBuilder app)
{
    app.UseOwin(buildFunc =>
    {
        var log = ConfigureLogger();
        buildFunc.UseMonitoringAndLogging(log, HealthCheck);
        buildFunc.UseNancy();
    }
}

private ILogger ConfigureLogger() {...}
public async Task<bool> HealthCheck() { ... }

```

Добавляем промежуточное ПО мониторинга и журналирования в правильной последовательности

Вот и весь код, который должен быть в проекте MicroserviceNET.Logging. Единственное, что нам осталось, — сформировать из библиотеки пакет NuGet. Это можно сделать с помощью утилиты dotnet. Перейдите в каталог проекта — тот, где располагается файл project.json, — в командной оболочке PowerShell и выполните следующую команду:

```
PS> dotnet pack --configuration Release
```

Эта команда создаст в каталоге bin/Release пакет NuGet под названием MicroserviceNET.Logging.1.0.0.nupkg. На рис. 11.4 показан MicroserviceNET.Logging.1.0.0.nupkg, открытый в утилите Package Explorer («Обозреватель пакетов») системы управления пакетами NuGet. Этот пакет содержит весь код проекта MicroserviceNET.Logging, скомпилированный в DLL, а также XML-файл документации с комментариями из этого кода. Как видите, этот пакет требует библиотеки Serilog, а значит, при установке его в проект устанавливается и Serilog.

Данный пакет готов к установке и использованию в таком количестве микросервисов, как вам заблагорассудится. В конце главы мы вернемся к вопросу установки его с вашей локальной машины при создании нового микросервиса на основе созданной нами платформы микросервисов.

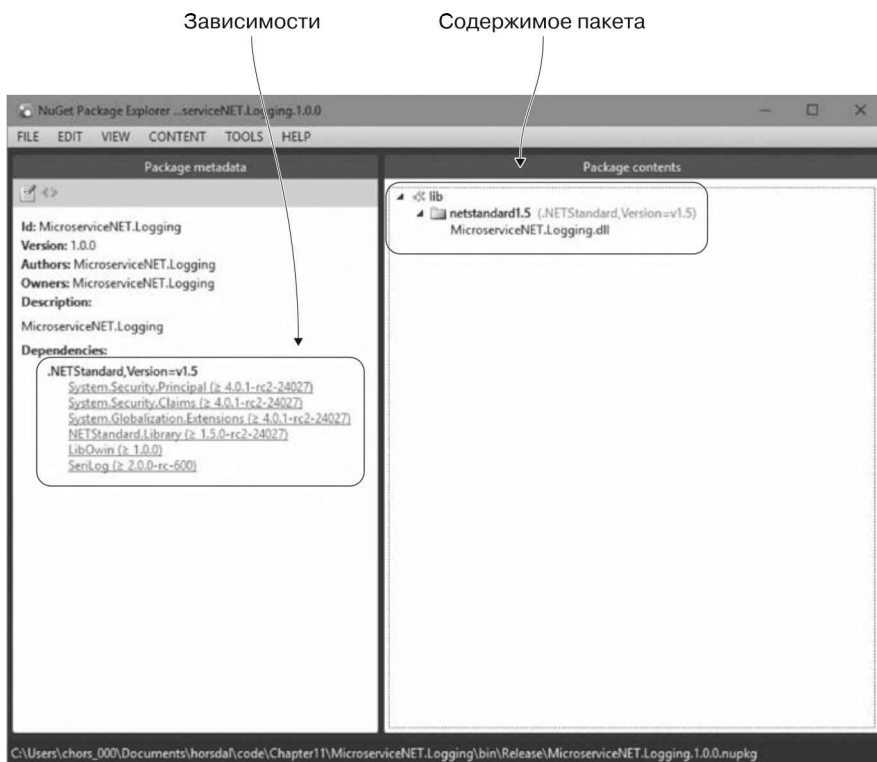


Рис. 11.4. Заглядываем внутрь пакета `MicroserviceNET.Logging.nupkg` с помощью обозревателя пакетов системы NuGet. Пакет содержит код мониторинга и журналирования, скомпилированный в DLL. В пакете также можно увидеть зависимость от библиотеки `Serilog`

Создание пакета с промежуточным ПО для авторизации

В этом разделе создадим NuGet-пакет `MicroserviceNET.Auth`, содержащий промежуточное ПО авторизации, созданное нами в главе 10. Шаги по его созданию аналогичны последовательности создания пакета `MicroserviceNET.Logging`.

1. Извлекаем созданное в главе 10 промежуточное ПО авторизации из микросервиса `Loyalty Program` в библиотеку классов `MicroserviceNET.Auth`.
2. На основе библиотеки `MicroserviceNET.Auth` создаем пакет NuGet, включающий весь код библиотеки.

Как и в предыдущем разделе, начнем с создания новой библиотеки классов. Назовем ее `MicroserviceNET.Auth`. Далее добавим в нее файл `AuthorizationMiddleware.cs` и вставим в него следующий код промежуточного ПО авторизации из главы 10 (листинг 11.6).

Листинг 11.6. Промежуточное ПО авторизации из микросервиса Loyalty Program

```

namespace MicroserviceNET.Auth
{
    using System.Threading.Tasks;
    using LibOwin;

    using AppFunc =
        System.Func<
            System.Collections.Generic.IDictionary<string, object>,
            System.Threading.Tasks.Task>;

    public class Authorization
    {
        {
            public AppFunc Middleware(AppFunc next, string requiredScope)
            {
                return env =>
                {
                    var ctx = new OwinContext(env);
                    var principal = ctx.Request.User;
                    if (principal.HasClaim("scope", requiredScope))
                        return next(env);
                    ctx.Response.StatusCode = 403;
                    return Task.FromResult(0);
                }
            }
        }
    }
}

```

Промежуточное ПО авторизации создано в лямбда-стиле

Вызываем следующий элемент конвейера только в случае наличия у запроса требуемой области действия

Если у запроса нет требуемой области действия, возвращаем ответ с кодом состояния 403 Forbidden (403 Запрещено)

Данное промежуточное ПО проверяет все входящие запросы на наличие требуемой области действия. Напомню, что в главе 10 мы реализовали такую схему взаимодействия между микросервисами, что микросервис Login выдавал микросервисам области действия, разрешающие им взаимодействовать. Два микросервиса могли взаимодействовать только в том случае, если это разрешил микросервис Login. Область действия — подтверждение того, что запрос разрешен микросервисом Login. Поэтому микросервисы должны проверять входящие запросы на любые требуемые области действия.

В главе 10 мы также создали промежуточное ПО для чтения идентификатора конечного пользователя из заголовка выполняемого между микросервисами запроса. Извлеките это промежуточное ПО, назовите его `IdToken` и поместите в библиотеку `MicroserviceNET.Auth` (листинг 11.7).

Листинг 11.7. Промежуточное ПО для идентификаторов пользователей

```

using LibOwin;
using System.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;

using AppFunc =
    System.Func<
        System.Collections.Generic.IDictionary<string, object>,
        System.Threading.Tasks.Task>;

public class IdToken
{

```

```

public static AppFunc Middleware(AppFunc next)
{
    return env =>
    {
        var ctx = new OwinContext(env);
        if (ctx.Request.Headers.ContainsKey("microservice.NET-end-user"))
        {
            var tokenHandler = new JwtSecurityTokenHandler();
            SecurityToken token;
            var userPrincipal =
                tokenHandler.ValidateToken(
                    ctx.Request.Headers["microservice.NET-end-user"],
                    new TokenValidationParameters(), out token);
            ctx.Set("microservice.NET-end-user", new User(userPrincipal));
        }
        return next(env);
    };
}

```

Проверяем на наличие заголовка, который должен содержать идентификатор конечного пользователя

Читаем и проверяем идентификатор конечного пользователя

Создаем объект пользователя на основе утверждений из идентификатора конечного пользователя и добавляем его в контекст OWIN

В главе 10 мы писали код в загрузчике Nancy для чтения объекта пользователя из окружения OWIN и передачи его в Nancy, чтобы у Nancy была информация о пользователе. Мы добавим эту функциональность в пакет, по немного иначе — воспользовавшись интерфейсом Nancy `IRequestStartup`, с которым еще не сталкивались. Аналогично тому, как Nancy автоматически подхватывает загрузчики и модули Nancy, он подхватывает и реализации интерфейса `IRequestStartup`. Nancy находит все такие реализации, включая падающие в пакетах NuGet, во время загрузки приложения и подключает их к конвейеру запроса. Реализация интерфейса `IRequestStartup` читает пользователя из окружения OWIN и передает его Nancy:

```

public class SetUser : IRequestStartup
{
    public void Initialize(IPipelines pipelines, NancyContext context) =>
    {
        context.CurrentUser =
            context.GetOwinEnvironment()["microservice.NET-end-user"]
                as ClaimsPrincipal;
    }
}

```

Вызывается для каждого запроса

Присваиваем пользователя пользователю из контекста Nancy

Этот маленький класс упрощает использование пакета `MicroserviceNET.Auth`, поскольку даппый небольшой кусочек функциональности подключается автоматически. К сожалению, элементы промежуточного ПО OWIN автоматически не подключаются. Чтобы упростить и это, создайте удобный метод для добавления промежуточного ПО в конвейер OWIN (листинг 11.8).

Листинг 11.8. Метод расширения для добавления промежуточного ПО авторизации

```

namespace MicroserviceNET.Auth
{
    using BuildFunc = System.Action<System.Func<
        System.Func<
            System.Collections.Generic.IDictionary<string, object>,
            System.Threading.Tasks.Task>,
            System.Func<
                System.Collections.Generic.IDictionary<string, object>,

```

Сигнатура OWIN BuildFunc

```

    System.Threading.Tasks.Task>
    >>;

    public static class BuildFuncExtensions
    {
        public static BuildFunc UseAuthPlatform(
            this BuildFunc buildFunc, string requiredScope)
        {
            buildFunc(next => Authorization.Middleware(next, requiredScope));
            buildFunc(next => IdToken.Middleware(next));
            return buildFunc;
        }
    }
}

```

Вспомогательный метод для добавления промежуточного ПО авторизации и промежуточного ПО идентификационных маркеров

Вот и весь код, который нужно добавить в пакет `MicroserviceNET.Auth`. Создайте пакет NuGet с помощью утилиты `dotnet`:

```
PS> dotnet pack --configuration Release
```

Новый пакет получит название `MicroserviceNET.Auth.1.0.0.nupkg` и находится в каталоге `bin/Release`. Теперь у нас есть уже два из трех составляющих платформу микросервисов пакетов NuGet. В следующем разделе создадим оставшийся пакет.

Создание пакета с REST-фабрикой клиентов

Последний пакет платформы микросервисов будет называться `MicroserviceNET.Platform` и содержать созданную нами в главах 9 и 10 фабрику `HttpClientFactory`. Он будет зависеть от двух остальных пакетов, а значит, пользователю достаточно установить `MicroserviceNET.Platform`, чтобы получить всю платформу.

Этапы создания пакета `MicroserviceNET.Platform` выглядят следующим образом.

1. Создать новую библиотеку классов `MicroserviceNET.Platform`.
2. Добавить созданные ранее NuGet-пакеты `MicroserviceNET.Logging` и `MicroserviceNET.Auth`.
3. Добавить NuGet-пакеты `RestSharp`, `IdentityModel` и `Nancy`.
4. Добавить файл `LibOwin.cs` с сайта <http://mng.bz/8pRq>, как описано в главе 8.
5. Добавить в `MicroserviceNET.Platform` код фабрики `HttpClientFactory`.
6. Добавить в `MicroserviceNET.Platform` удобный метод для упрощения правильной настройки фабрики `HttpClientFactory` и регистрации ее в контейнере внедрения зависимостей (DI) `Nancy`.

Первые три шага аналогичны тому, что вы уже делали неоднократно. После создания нового проекта `MicroserviceNET.Platform` и добавления необходимых NuGet-пакетов раздел внедрения зависимостей в файле `project.json` будет выглядеть следующим образом:

```

"dependencies": {
    "NETStandard.Library": "1.6.0",
    "MicroserviceNET.Auth": {

```

```
        "target": "project",
        "version": "1.0.0"
    },
    "MicroserviceNET.Logging": {
        "target": "project",
        "version": "1.0.0"
    },
    "Nancy": "2.0.0-barneyrubble",
    "IdentityModel": "2.0.0-beta5"
}
```

Создание и использование своей собственной ленты пакетов NuGet

Создаваемые нами для платформы микросервисов пакеты системы управления пакетами NuGet предназначены для использования только в нашей системе микросервисов. А значит, их место не в общедоступной ленте пакетов NuGet на сайте <http://www.nuget.org/>. К счастью, система управления пакетами NuGet не требует многого для ленты — достаточно папки на локальном диске или общей папки на сетевом диске. Необходимо только настроить NuGet, чтобы она искала пакеты в этой папке, а не только на сайте <http://www.nuget.org/>.

Сначала выберите папку, в которой будут находиться пакеты NuGet. В данном случае это будет папка `c:\nuget-packages\`. Далее скопируйте в нее свои пакеты NuGet. Пакеты, создаваемые с помощью утилиты `dotnet`, попадают в подкаталог `bin\Release` каталога проекта. Скопируйте пакеты, то есть файлы `.nupkg`, в папку `c:\nuget-packages\`. Наконец, настройте NuGet на поиск пакетов в этой папке путем добавления ее в список лент пакетов в файле конфигурации NuGet, расположенном в вашем профиле пользователя в подкаталоге `~AppData\Roaming\NuGet\NuGet.Config`. Например, мой файл `NuGet.Config` выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="nuget.org" value="https://www.nuget.org/api/v2/" />
    <add key="test" value="c:\nuget-packages\" />
  </packageSources>
  <disabledPackageSources>
    <add key="NancyAsync" value="true" />
  </disabledPackageSources>
  <activePackageSource>
    <add key="nuget.org" value="https://www.nuget.org/api/v2/" />
  </activePackageSource>
</configuration>
```

Теперь при вызове команды `dotnet restore` поиск пакетов будет производиться также в каталоге `c:\nuget-packages\`, благодаря чему можно будет установить свои пакеты из этого каталога в различные проекты.

Следующий этап — добавление в проект нового файла `HttpClientFactory.cs` и вставка в него кода фабрики `HttpClientFactory`, который мы написали в главах 9 и 10 (листинг 11.9)

Листинг 11.9. HttpClientFactory

```
namespace MicroserviceNET.Platform
```

```
{
```

```
    using System;
    using System.Net.Http;
    using System.Net.Http.Headers;
    using System.Threading.Tasks;
    using IdentityModel.Client;
```

```
    public interface IHttpHttpClientFactory
```

```
    {
```

```
        Task<HttpClient> Create(Uri uri, string requestScope);
```

```
    }
```

```
    public class HttpClientFactory : IHttpHttpClientFactory
```

```
    {
```

```
        private readonly TokenClient tokenClient;
        private readonly string correlationToken;
        private readonly string idToken;
```

```
        public HttpClientFactory(
```

```
            string tokenUrl,
```

```
            string clientName,
```

```
            string clientSecret,
```

```
            string correlationToken,
```

```
            string idToken)
```

```
        {
```

```
            this.tokenClient =
```

```
                new TokenClient(tokenUrl, clientName, clientSecret);
```

```
            this.correlationToken = correlationToken;
```

```
            this.idToken = idToken;
```

```
        }
```

```
        public async Task<HttpClient>
```

```
            Create(Uri uri, string requestScope)
```

```
        {
```

```
            var response = await
```

```
                this.tokenClient
```

```
                    .RequestClientCredentialsAsync(requestScope)
```

```
                    .ConfigureAwait(false);
```

```
            var client = new HttpClient() { BaseAddress = uri }; ←
```

```
            client.DefaultRequestHeaders.Authorization =
```

```
                new AuthenticationHeaderValue("Bearer", response.AccessToken); ←
```

```
            client
```

```
                .DefaultRequestHeaders
```

```
                    .Add("Correlation-Token", this.correlationToken);
```

```
            if (!string.IsNullOrEmpty(this.idToken)) ←
```

```
                client
```

```
                    .DefaultRequestHeaders
```

```
                        .Add("microservice.NET-end-user", this.idToken);
```

```
            return client;
```

```
        }
```

```
    }
```

```
}
```

Название клиента и секретный ключ, используемые при получении маркера доступа от конечной точки маркеров

URL конечной точки маркеров в микросервисе Login

Маркер корреляции (свой для каждого запроса), поступающий от элемента промежуточного ПО

Маркер с идентификационными данными конечного пользователя

Запрос у микросервиса Login маркера авторизации, который разрешил бы обращения, требующие указанной в requestScope области действия

Добавляем маркер корреляции в заголовок запроса

Добавляем маркер авторизации в заголовок запроса

Подготовка клиента к выполнению запросов к URI

Добавляем идентификационные данные конечного пользователя в заголовок запроса

Фабрика `HttpClientFactory` отвечает за создание объектов `HttpClient`, используемых для выполнения запросов к другим микросервисам. Фабрика гарантирует, что объекты `HttpClient` смогут выполнять только соответствующие правилам системы микросервисов запросы. `HttpClientFactory` гарантирует, что выполняемые с помощью созданных ею объектов `HttpClient` запросы содержат следующее.

- ❑ *Маркер авторизации для какой-либо области действия.* Используя объект `HttpClient` код должен указывать, какая область действия необходима ему для этих запросов, по `HttpClientFactory` гарантирует получение и помещение в запросы маркеров авторизации.
- ❑ *Маркер корреляции.* Может быть использован для отслеживания цепочки выполнения запросов в системе микросервисов. Один из элементов промежуточного ПО из пакета `Microsoft.NET.Logging` обеспечивает чтение маркеров корреляции входящих запросов. Необходимо передавать эти маркеры `HttpClientFactory`, обеспечивающей вставку маркеров корреляции также во все исходящие запросы.
- ❑ *Маркер, содержащий идентификационные данные конечного пользователя, если запрос порождается запросом конечного пользователя.* Элемент промежуточного ПО из пакета `Microsoft.AspNetCore.Authentication` выполняет чтение идентификатора конечного пользователя из входящих запросов. Необходимо передать этот маркер `HttpClientFactory`, которая обеспечивает его передачу со всеми исходящими запросами.

Как видно из листинга 11.9, конструктор `HttpClientFactory` принимает на входе не менее пяти аргументов. Два из них — маркер корреляции и идентификационный маркер — поступают из промежуточного ПО в других пакетах платформы микросервисов. Эти элементы промежуточного ПО добавляют маркеры корреляции и идентификационные маркеры в окружение OWIN.

Автоматическая регистрация фабрики `HttpClientFactory` в контейнере Nancy

Хотелось бы, чтобы нашу платформу было удобно использовать для создания микросервисов. Для этого мы добавим в пакет `Microsoft.AspNetCore.Platform` несколько методов для упрощения настройки `HttpClientFactory`. В этих методах предполагается, что микросервисы добавили промежуточное ПО из двух других пакетов платформы и в окружении OWIN присутствуют маркер корреляции и идентификационный маркер.

Добавьте в проект `Microsoft.AspNetCore.Platform` файл `MicrosoftPlatformHelper.cs`. Мы вставим в этот файл два метода. Первый будет вызываться микросервисами из класса `Startup` и служить для запоминания определенной статической конфигурации: URL маркера, названия клиента и секретного ключа клиента (листинг 11.10).

Листинг 11.10. Сохранение необходимой фабрике `HttpClientFactory` статической конфигурации

```
namespace Microsoft.AspNetCore.Platform
{
    using System.Security.Claims;
    using Nancy;
    using Nancy.Owin;
```

```

using Nancy.TinyIoc;
using LibOwin;

public static class MicroservicePlatform
{
    private static string TokenUrl;
    private static string ClientName;
    private static string ClientSecret;

    public static void Configure(
        string tokenUrl,
        string clientName,
        string clientSecret)
    {
        TokenUrl = tokenUrl;
        ClientName = clientName;
        ClientSecret = clientSecret;
    }
}

```

Второй удобный метод (листинг 11.11) предназначен для использования в загрузчике Nancy, где он будет вызываться при каждом запросе. Он создает на основе этой конфигурации и информации из окружения OWIN фабрику `HttpClientFactory`, после чего регистрирует ее в контейнере внедрения зависимостей Nancy, чтобы у кода приложения (модулей Nancy и т. н.) была возможность получать ссылку на `HttpClientFactory` и автоматически внедрять ее.

Листинг 11.11. Удобный метод для регистрации `HttpClientFactory`

Читаем маркер корреляции
из окружения OWIN

```

public static TinyIoCContainer UseHttpClientFactory(
    this TinyIoCContainer self,
    NancyContext context)
{
    var correlationToken =
    context.GetOwinEnvironment()?["correlationToken"] as string;
    object key = null;
    context
        .GetOwinEnvironment()
        ?.TryGetValue(OwinConstants.RequestUser, out key);
    var principal = key as ClaimsPrincipal;
    var idToken = principal?.FindFirst("id_token");
    self.Register<IHttpClientFactory>(
        new HttpClientFactory(
            TokenUrl, ClientName,
            ClientSecret,
            correlationToken ?? "",
            idToken?.Value));
    return self;
}

```

Читаем конечного
пользователя
из окружения OWIN

Создаем `HttpClientFactory`
со всей нужной информацией

Получаем из объекта пользователя
идентификационный маркер
конечного пользователя

Регистрируем `HttpClientFactory` в качестве индивидуальной
зависимости для каждого запроса в контейнере Nancy

Вот и все, что нужно для пакета `MicroserviceNET.Platform`. Осталось только упаковать NuGet:

```
PS> dotnet pack --configuration Release
```

Пакет `MicroserviceNET.Platform` создается в виде файла `bin/Release/MicroserviceNET.Platform.1.0.0.nupkg`. На рис. 11.5 показано содержимое этого пакета NuGet, представляющего собой DLL. Из этого рисунка также видно, что `MicroserviceNET.Platform` зависит от двух других пакетов платформы — `MicroserviceNET.Logging` и `MicroserviceNET.Auth`.

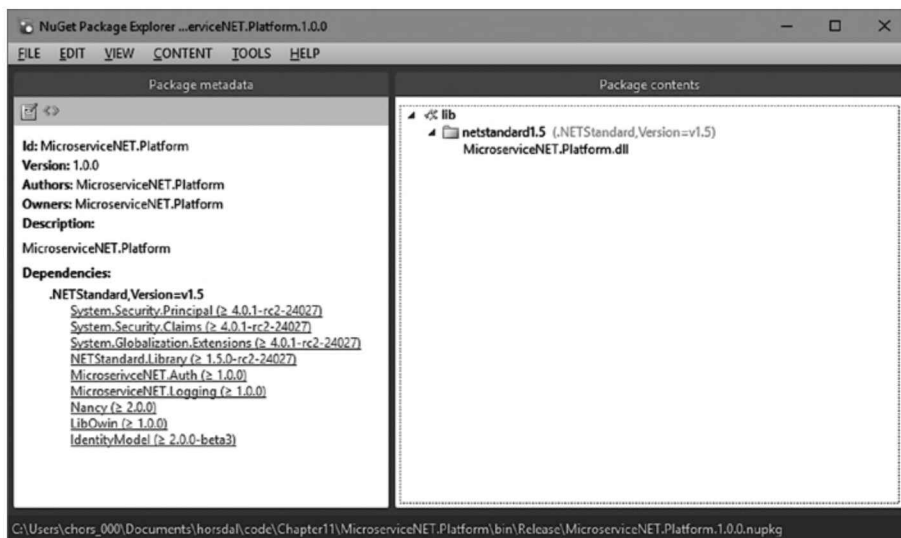


Рис. 11.5. Заглянув внутрь пакета `MicroserviceNET.Platform.1.0.0.nupkg`, видим, что он зависит от двух других пакетов платформы — `MicroserviceNET.Logging` и `MicroserviceNET.Auth`, а также от нескольких внешних пакетов

На этом создание платформы микросервисов завершено. В следующем разделе посмотрим на нее в действии.

Использование платформы микросервисов

Мы создали платформу, облегчающую создание новых микросервисов, работающих должным образом в производственной среде. В данном разделе создадим с помощью этой платформы и фреймворка Nancy небольшой микросервис а-ля Hello World. Поскольку микросервис будет основан на нашей платформе микросервисов, он будет обладать предоставляемыми ею возможностями мониторинга, журналирования и обеспечения безопасности.

Создание микросервиса не потребует больших усилий. Нужно сделать всего лишь следующее.

1. Создать пустое веб-приложение `HelloMicroservicePlatform`.
2. Добавить в него NuGet-пакет `MicroserviceNET.Platform`.

3. Выполнить настройки платформы микросервисов в классе `Startup`.
4. Создать небольшой загрузчик `Nancy` и настроить регистрацию `HttpClientFactory`.
5. Добавить маленький модуль `Nancy`.

Начнем с создания пустого веб-приложения с помощью утилиты `yo` или `Visual Studio`. Добавьте `NuGet`-накет `MicroserviceNET.Platform`, указав его в зависимостях в файле `project.json` и выполнив команду `dotnet restore`. Добавление пользовательского накета `NuGet` ничем не отличается от добавления любого другого, так что зависимости будут выглядеть вот так:

```
"dependencies": {
  "Microsoft.NETCore.App": {
    "version": "1.0.0-rc2-3002702",
    "type": "platform"
  },
  "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0-rc2-final",
  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0-rc2-final",
  "Microsoft.AspNetCore.Owin": "1.0.0-rc2-final",
  "Serilog": "2.0.0-rc-600",
  "MicroserviceNET.Platform": "1.0.0",
  "Serilog.Sinks.ColoredConsole": "2.0.0-beta-700"
},
```

Пакет `MicroserviceNET.Platform` импортирует два других `NuGet`-накета платформы микросервисов — `MicroserviceNET.Logging` и `MicroserviceNET.Auth`, а также библиотеку `Serilog` и фреймворк `Nancy`. Другими словами, `MicroserviceNET.Platform` импортирует все, что нужно для создания микросервиса.

Следующий шаг — настройка платформы микросервисов. Сделать это можно в классе `Startup`, вызвав несколько удобных методов, установив `Serilog` и создав простейшую проверку состояния (листинг 11.12).

Листинг 11.12. Настройка платформы микросервисов в классе `Startup`

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseOwin()
            .UseMonitoringAndLogging(ConfigureLogger(), HealthCheck)
            .UseAuthPlatform("test-scope")
            .UseNancy();
    }

    private ILogger ConfigureLogger()
    {
        ...
    }

    private static Task<bool> HealthCheck()
    {
        ...
    }
}
```

Настройка относящейся к авторизации части платформы →

Настройка относящейся к мониторингу и журналированию части платформы

Теперь у нашего микросервиса есть конечные точки мониторинга, журналирование запросов и журналирование показателей производительности, и мы проверяем

маркеры доступа входящих запросов. Оставшиеся части платформы относятся к обработке исходящих запросов. Для обеспечения их работы добавьте в микросервис следующий загрузчик Nancy (листинг 11.13).

Листинг 11.13. Регистрация фабрики HttpClientFactory в контейнере Nancy

```
public class Bootstrapper : DefaultNancyBootstrapper
{
    protected override void RequestStartup(
        TinyIoCContainer container,
        IPipelines pipelines,
        NancyContext context)
    {
        base.RequestStartup(container, pipelines, context);
        container.UseHttpClientFactory(context);
    }
}
```

Вызывается фреймворком Nancy для каждого запроса

Удобный метод, регистрирующий HttpClientFactory в контейнере Nancy

Платформа микросервисов настроена и готова к использованию. Вот и все, что нужно для создания нового микросервиса. Остается только добавить поведение. Последний этап создания микросервиса Hello World — добавление модуля Nancy с одной конечной точкой, выполняющего запрос к другому микросервису и отправляющего ответ обратно вызывающей стороне (листинг 11.14). Это не такое уж сложное поведение для микросервиса, но оно позволяет проиллюстрировать работу платформы микросервисов с исходящими запросами: в запросе к другому микросервису содержатся добавленные платформой микросервисов как маркер корреляции, так и маркер доступа.

Листинг 11.14. Использование фабрики HttpClientFactory для выполнения «правильных» запросов

```
public class Hello : NancyModule
{
    public Hello(IHttpClientFactory clientFactory)
    {
        Get("/", async (_, __) =>
        {
            var client = await
                clientFactory.Create(
                    new Uri("http://otherservice/"),
                    "scope_for_other_microservice");
            var resp = await
                client.GetAsync("/some/path").ConfigureAwait(false);
            return resp.StatusCode;
        });
    }
}
```

Создаем объект HttpClient для выполнения запросов к другим микросервисам

Отправляем запросы к другим микросервисам, включая в них маркеры корреляции и маркеры доступа, добавляемые платформой микросервисов

Микросервис Hello World готов к использованию. Занузить его можно точно так же, как и любой другой, — с помощью утилиты dotnet. Если по URL `http://otherservice` у вас занушен другой микросервис, то микросервис Hello World будет принимать запросы только с действительным маркером доступа для области действия `test-scope`, журналировать все запросы и ответы и возвращать код состояния получаемых от этого другого микросервиса ответов. У микросервиса Hello World также имеются точки мониторинга, и он использует маркеры корреляции.

11.4. Резюме

- ❑ Поскольку в системе микросервисов нам часто понадобится создавать новые микросервисы, необходима возможность быстрого и удобного создания микросервисов с нуля.
- ❑ Чтобы удовлетворить сквозные потребности в мониторинге, журналировании и безопасности, все микросервисы в системе должны выполнять некоторые операции. Какие именно, зависит от системы.
- ❑ Необходимо разработать платформу многоразового кода для вашей системы микросервисов. С помощью такой платформы можно с легкостью создавать новые микросервисы, работающие должным образом в смысле мониторинга, журналирования и безопасности.
- ❑ Платформа многоразового кода для микросервисов должна быть нацелена на реализацию только сквозной технической функциональности, такой как мониторинг, журналирование и обеспечение безопасности.
- ❑ Платформа многоразового кода для микросервисов не должна решать задачи логики предметной области, поскольку эта логика различна для разных микросервисов.
- ❑ NuGet — отличный формат для распространения платформы микросервисов.
- ❑ Создание локальной ленты пакетов NuGet не представляет сложностей.
- ❑ Для создания пакета NuGet можно использовать команду `dotnet pack`.
- ❑ Можно создать пакеты NuGet для решения следующих задач:
 - добавления в микросервисы конечных точек мониторинга;
 - добавления в микросервисы журналирования запросов;
 - добавления в микросервисы журналирования показателей производительности;
 - добавления маркеров корреляции во все журнальные сообщения;
 - добавления маркеров корреляции во все исходящие запросы;
 - разрешения выполнения только входящих запросов, содержащих маркер доступа для требуемой области действия;
 - добавления маркеров доступа для нужных областей действия во все исходящие запросы.
- ❑ Пакеты NuGet можно создавать на основе библиотек.
- ❑ Можно использовать пользовательские пакеты NuGet в своих микросервисах.

Часть IV

Создание приложений

В этой части книги мы нанесем на общую картину завершающие штрихи — узнаем, как создать приложение для конечных пользователей. Вы уже знаете, как разбить систему на микросервисы и как создавать эти микросервисы. В главе 12 вы создадите графический интерфейс на базе ваших микросервисов, чтобы конечные пользователи смогли воспользоваться всей их функциональностью.

12 Создание приложений на основе микросервисов

В этой главе:

- ❑ сборка приложений на основе системы микросервисов;
- ❑ шаблоны проектирования «Составное приложение», «Шлюз API» и «Бэкенд для фронтенда»;
- ❑ использование отрисовки на стороне клиента и на стороне сервера.

До этого момента мы рассматривали реализацию микросервисов для бизнеса и обеспечение доступа к ним с помощью HTTP API. Но конечные пользователи не работают с HTTP API — они работают с веб-приложениями, мобильными приложениями, настольными приложениями, умными телевизорами, очками виртуальной реальности и другими приложениями на устройствах, интерфейсы которых предназначены для людей. Для того чтобы конечные пользователи получили возможность работать с микросервисами, нам нужно реализовать приложения на базе этих микросервисов. Этой теме и посвящена данная глава: мы не откажемся от разработки отдельных микросервисов на объединение микросервисов в архитектуру, которая поддерживает создание приложений для конечных пользователей.

Начнем с широкой дискуссии без технических терминов о том, как создавать приложения на основе системы микросервисов. Затем рассмотрим три конкретных архитектурных шаблона, необходимых для реализации приложений: «Составное приложение», «Шлюз API» и «Бэкенд для фронтенда».

12.1. Приложения для систем микросервисов: одно или несколько приложений?

Существует несколько способов построения приложений, ориентированных на пользователей, как с точки зрения пользователя, так и с технической точки зрения. Мы начнем с точки зрения пользователей и рассмотрим несколько способов предоставления конечным пользователям функциональности микросервисов, от универсальных приложений, предоставляющих всю функциональность системы, до коллекции небольших специализированных приложений, каждое из которых предлагает лишь несколько функций. Эти два способа выделения функциональности представляют разные концы спектра, как показано на рис. 12.1.



Рис. 12.1. На основе системы микросервисов может быть построено множество типов приложений, начиная с универсальных и заканчивая специализированными

Между этими крайностями лежит множество других вариантов построения приложений, которые захватывают большие или меньшие части функциональности системы микросервисов. Положение конкретной системы в спектре приложений зависит от ее контекста — конечных пользователей, функциональности и т. д.

Оба края спектра имеют свои плюсы. В следующих двух разделах мы их рассмотрим, чтобы вы смогли увидеть количество способов, с помощью которых ваша система микросервисов может предоставлять свою функциональность. Положение системы в спектре влияет на способ построения вашего приложения (приложений), а также на используемые шаблоны проектирования.

Универсальные приложения

Система микросервисов может предоставлять множество разных функций. Рассмотрим, например, бизнес-систему для страховой компании. Система управляет бизнес-процессами, включая продажу полисов, установку цен и обработку исков, сделанных пользователями. Ею пользуются:

- ❑ продавцы, которые звонят потенциальным покупателям и предлагают свои услуги;
- ❑ актуарии, устанавливающие цены на полисы и оценивающие бизнес-риски, основываясь на оценке будущих исков и дохода;
- ❑ оценщики, определяющие стоимость товаров, которые клиенты хотят застраховать и для которых клиенты создают иски;
- ❑ лица, которые занимаются расследованием и урегулированием клиентских исков;
- ❑ IT-персонал, наблюдающий за пользователями и правами.

Вы реализуете все функции, которыми должна обладать система страхования, в разных микросервисах, но также можете решить реализовать распространённое приложение, которое содержит всю функциональность и используется всеми пользователями. Такое приложение можно назвать универсальным. Оно может иметь любую форму — например, веб-приложения или настольного приложения. Приложение для Facebook для iOS или Android является примером универсального приложения: оно позволяет пользователям задействовать все функции Facebook, например читать свою новостную ленту, отправлять сообщения, писать на стенах, управлять настройками, загружать фотографии в альбомы и многое другое.

Вы можете захотеть создать универсальное приложение, которое объединяет в себе всю функциональность системы, но несколькими причинами. Например, некоторым пользователям системы страхования может понадобиться более одной функции, например лицам, урегулирующим иски и одновременно выполняющим

оценку. Также может существовать пересечение между видами функций, необходимыми разным группам пользователей.

Управляя правами, вы можете ограничивать возможности каждого пользователя во время работы с универсальными приложениями. Но все они используют одно и то же приложение, которое предоставляет всю функциональность системы.

Специализированные приложения

Еще одним вариантом построения приложений на основе системы микросервисов является создание множества небольших специализированных приложений. В какой-то степени этот подход противоположен созданию универсального приложения: у вас есть несколько узкоспециализированных приложений, каждое из которых охватывает лишь небольшую часть функциональности всей системы. Продолжая рассматривать пример с бизнес-приложением для страховой компании, вы можете создать отдельные приложения для следующих целей:

- ☐ просмотра каталога полисов, предлагаемых компанией;
- ☐ создания предложений для потенциальных клиентов;
- ☐ создания страхового полиса для клиента;
- ☐ создания отчетов о полисах, действительных в данный момент;
- ☐ создания прогнозов о стоимости будущих исков;
- ☐ регистрации и расследования исков;
- ☐ оценки застрахованных товаров;
- ☐ урегулирования исков.

В рамках этого подхода пользователи должны использовать более одного приложения. Поначалу это может показаться неправильным, но так ли это на самом деле? Если вы станете реализовывать эти приложения как веб-приложения, они могут содержать ссылки друг на друга, создавая связанность, несмотря на то что функциональность разбита на множество небольших приложений. Философия, которая заключается в том, что одно приложение должно выполнять одну задачу, и выполнять ее хорошо, может привести к тому, что для каждого фрагмента функциональности могут быть написаны очень хорошие приложения. Например, Facebook имеет приложение для Android, которое называется Selfies и является частью системы обмена сообщениями. Оно предназначено для выполнения всего одной задачи: создания селфи и отправки их с помощью Facebook Messenger. Это приложение упрощает выполнение данной задачи.

Несмотря на то что я упомянул лишь универсальные и специализированные приложения, ваш выбор не ограничивается этими вариантами. Существует целый спектр вариантов (рис. 12.2), который начинается с одного большого приложения, которое предоставляет всю функциональность системы, и заканчивается множеством специализированных приложений. Другие варианты выглядят так.

- ☐ Разбиение одного всеобъемлющего приложения на два, одно из которых управляет административными функциями, а другое — всеми остальными.

- Объединение нескольких специализированных приложений в более крупные приложения, которые охватывают всю функциональность, необходимую одному типу пользователей.



Рис. 12.2. Повторю: так выглядит широкий спектр типов приложений, которые можно создать на основе микросервисов

В случае со страховой компанией одним из компромиссных подходов станет создание приложений для каждого типа пользователей. Люди, занимающиеся продажами, получают приложение, содержащее всю необходимую им функциональность, от поиска новых клиентов до подписания сделок. Аналогично, актуарии, оценщики, лица, урегулирующие иски, и IT-персонал получают приложения, подготовленные специально для них.

Несмотря на то что этот подход позволит избежать сложностей, возникающих при создании крупного универсального приложения, а также большого количества специализированных приложений, он имеет ряд собственных проблем: не исключено, что функциональность будет дублироваться между приложениями, некоторые пользователи имеют более одной роли и т. д. Правильного подхода не существует: вам нужно решить, над какую часть спектра поднадаете вы, основываясь на собственных требованиях, а не на технических решениях.

Теперь обратимся к технической стороне вопроса и взглянем на три шаблона, используемых при построении приложений на основе микросервисов.

12.2. Шаблоны для построения приложений на основе микросервисов

В этом разделе рассматриваются шаблоны проектирования «Составное приложение», «Шлюз API» и «Бэкенд для фронтенда».

Составные приложения: интеграция на фронтенде

Первым шаблоном для построения приложений на основе микросервисов является шаблон «Составное приложение». Такое приложение состоит из функциональности, взятой из нескольких мест — в нашем случае из нескольких микросервисов, — с каждым из которых оно связывается непосредственно. Микросервисы могут общаться друг с другом для выполнения своих задач — для составного приложения это важно.

На рис. 12.3 мы возвращаемся к примеру со страховкой — у нас есть универсальное приложение, которое включает в себя всю функциональность системы. Система для работы со страховками построена с использованием микросерви-

сов, но поэтому для того, чтобы предоставить всю функциональность системы, приложение должно получать бизнес-возможности от нескольких микросервисов. Микросервисов в системе больше, чем показано на рисунке, и приложение не будет непосредственно наследовать функциональность от всех них. Такое приложение сосредоточивает всю функциональность в одном месте, отсюда и возник термин «*составное приложение*».

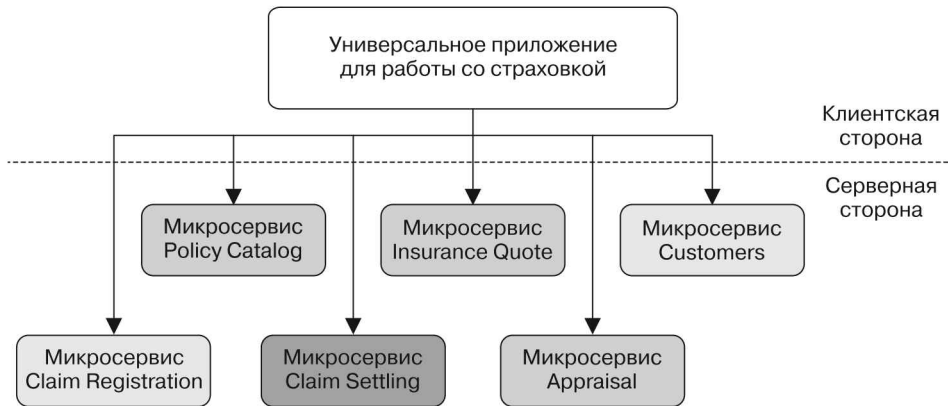


Рис. 12.3. Составное универсальное приложение использует несколько разных микросервисов, каждый из которых предоставляет функциональность и интерфейс для ее использования

Когда вы создаете составное универсальное приложение на основе системы микросервисов, эти микросервисы предоставляют функциональность и интерфейс. Как следствие, интерфейс приложения является объединением меньших интерфейсов, полученных из разных микросервисов. На рис. 12.4 показан пример структуры интерфейса страхового приложения: он состоит из четырех разделов, каждый из которых берется из отдельного микросервиса, предоставляющего функциональность и интерфейс.

Реализация объединения GUI выполняется в зависимости от технологии, использованной при построении клиента. Работая с настольным приложением Windows Presentation Foundation (WPF) (<http://mng.bz/0YfW>), вы можете, к примеру, использовать фреймворк Managed Extensibility Framework (MEF, <http://mng.bz/6NKA>) для того, чтобы динамически загрузить в приложение компоненты, каждый из которых будет иметь собственный интерфейс. При использовании веб-приложения интерфейс может быть создан загрузкой фрагментов HTML и накетов JavaScript из микросервиса в главное приложение и добавлением их в модель DOM с помощью JavaScript. В обоих случаях микросервисы предоставляют функциональность и интерфейс.

Не все составные приложения являются универсальными приложениями, они могут быть и меньшего размера. Например, если страховое приложение имеет приложения для каждого типа пользователей, каждое приложение должно предоставлять функциональность, которая принадлежит разным бизнес-возможностям

и, соответственно, разным микросервисам. Оно соответствует принципу, который гласит, что каждое приложение для определенного типа пользователей может быть создано как составное.

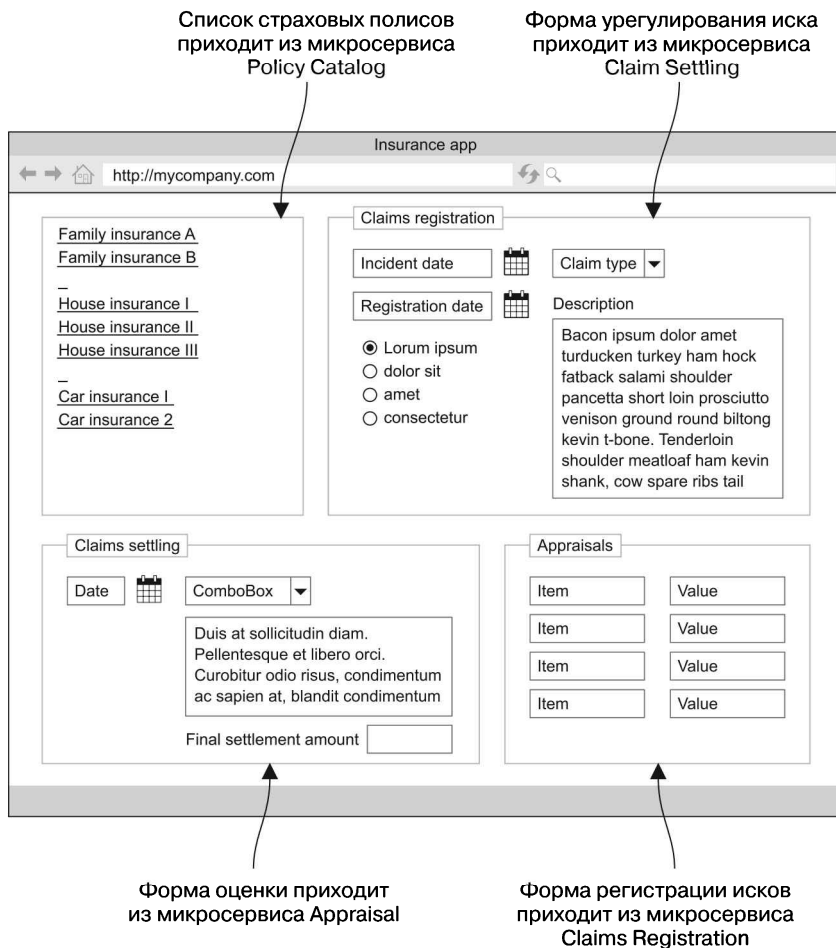


Рис. 12.4. Составное приложение получает компоненты интерфейса из разных микросервисов и использует их для того, чтобы сформировать связанный, составной интерфейс

Преимущества. Когда вы создаете составные приложения, интерфейс разбивается на меньшие части в соответствии с бизнес-возможностями точно так же, как функциональность распределяется между микросервисами. Это означает, что интерфейс для каждой бизнес-возможности реализуется и разворачивается вместе с соответствующей бизнес-логикой. Поскольку составное приложение получает интерфейс для бизнес-возможности от микросервиса, приложение обновляется при каждом обновлении микросервиса. Это означает, что гибкость, которой вы до-

стигаете разбиением системы на небольшие специализированные микросервисы, применима и к интерфейсу приложения.

Недостатки. Составное приложение отвечает на интегрирование всей функциональности, реализованной в системе микросервисов. Эта задача может оказаться сложной: в системе микросервисов может находиться множество бизнес-возможностей, а интерфейс приложения может быть разбит по другому принципу, что приводит к появлению страниц, включающих интерфейсы для нескольких разных микросервисов, которые не выглядят для конечного пользователя как единая страница.

Такой вид сложности может означать, что составное приложение точно знает, как работают микросервисы и их интерфейсы. Если составное приложение начинает делать слишком много предположений об интерфейсах микросервисов, оно становится чувствительным к изменениям в каждом микросервисе и поэтому может перестать работать целиком в результате изменения в каком-то одном микросервисе. Если вы оказались в подобной ситуации, это означает, что вы потеряли гибкость, которая является одним из основных преимуществ использования составного приложения.

В заключение хотелось бы отметить, что составные приложения могут работать очень хорошо, но только если вы сможете избежать реализации сложных взаимодействий.

Шлюз API

Вторым шаблоном для построения приложений на основе микросервисов является «Шлюз API». Шлюз API — это микросервис с открытым HTTP API, который охватывает всю функциональность системы, но не реализует никакой функциональности сам по себе. Вместо этого шлюз API делегирует всю работу другим микросервисам. По сути, шлюз API действует как адантер между приложениями и системой микросервисов.

Когда вы строите приложение на основе системы микросервисов, которая использует шлюз API, приложения не знают, как функциональность системы разбита между микросервисами, — они даже не знают, что система использует микросервисы. Приложению нужно знать только об одном микросервисе — о шлюзе API.

На протяжении этой книги вы видели пример реализации корзины для интернет-магазина, который содержит шлюз API. На рис. 12.5 показан запрос на добавление элемента в корзину пользователя, поступающий из приложения в шлюз API, который делегирует его другим микросервисам для обслуживания запроса. Роль шлюза API в этом случае заключается в том, что приложение имеет единую точку входа, что упрощает интерфейс системы, поэтому приложениям не нужно взаимодействовать с несколькими микросервисами непосредственно.

Используя шлюз API, вы можете построить любое приложение, от универсального, которое пользуется всей функциональностью шлюза API, до специализированных приложений, которые применяют лишь часть шлюза API, а также любое промежуточное приложение.

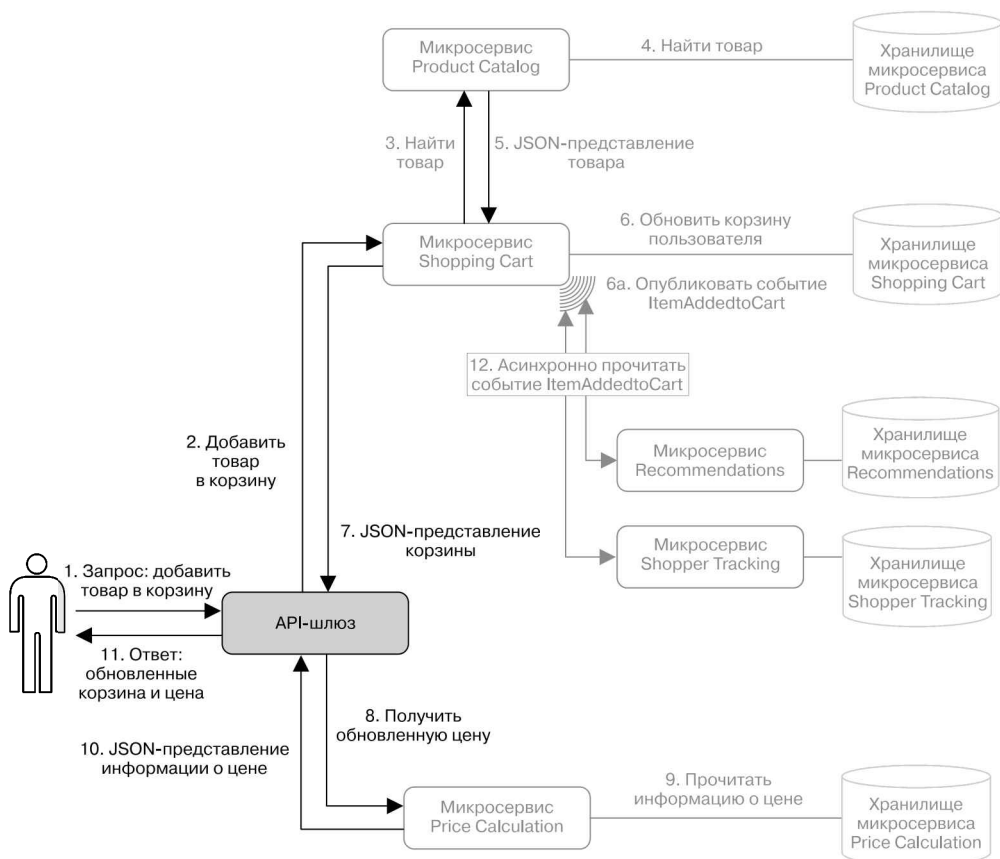


Рис. 12.5. Шлюз API — это единая входная точка для приложений. Любой запрос из приложения попадает в шлюз API, который делегирует его остальной части системы микросервисов

Преимущества. Основное преимущество шаблона «Шлюз API» заключается в том, что приложения отвязываются от структуры микросервисов. Шлюз API полностью скрывает эту структуру от приложений.

Когда функциональность нескольких приложений нересекается или некоторые приложения созданы третьей стороной, использование шаблона «Шлюз API» поддерживает:

- ❑ низкий входной барьер для наостроения приложений;
- ❑ стабильность открытого API;
- ❑ обратную совместимость открытого API.

Использование шлюза API означает, что разработчикам приложений нужно рассмотреть только один API для того, чтобы начать работу. Вы можете сконцентрироваться на поддержании стабильности и обратной совместимости API по мере развития других микросервисов.

Недостатки. Основным недостатком шаблона «Шлюз API» является тот факт, что шлюз API сам по себе может разрастись до размера крупной базы кода и, как следствие, приобрести все недостатки монолитной системы. Это особенно верно, если вы поддадитесь искушению реализовать бизнес-логику в шлюзе API. Шлюз API может получать функциональность из нескольких других микросервисов для того, чтобы выполнить один запрос. Поскольку он и так объединяет данные из нескольких микросервисов, может возникнуть соблазн применить к этим данным какие-нибудь бизнес-правила. В ближайшей перспективе это можно сделать быстро, но это подталкивает шлюз API в сторону монолитности.

В заключение хотелось бы сказать, что шаблон «Шлюз API» очень полезен и зачастую является верным вариантом развития приложения. Но вам следует тщательно следить за размером шлюза API и быть готовыми отреагировать, если он станет настолько тяжел, что с ним будет трудно работать.

Бэкенд для фронтенда

Третий и последний шаблон для построения приложений на основе микросервисов, который мы рассмотрим в этой книге, — это «Бэкенд для фронтенда» (*backend for frontend, BFF*). Шаблон BFF полезен, когда вам нужно построить больше одного приложения для системы микросервисов. Например, система страхования может иметь веб-приложение, содержащее наиболее часто используемую функциональность, приложение для iOS, которое эксперты-оценщики могут использовать вне офиса, а также специализированное приложение для ПК, предназначенное для выполнения задач актуариев. BFF — это микросервис, похожий на шлюз API, но он специализирован для одного приложения. Если вы используете этот шаблон для приложений системы страхования, у вас будут BFF для веб-приложения, BFF для приложения iOS и BFF для настольного приложения для актуариев (рис. 12.6).

Идея BFF заключается в поддержке приложения, построенного на его основе. Это означает, что приложение и BFF тесно связаны: BFF предоставляет необходимую приложению функциональность так, чтобы создание приложения выглядело максимально просто.

Преимущества. Благодаря шаблону BFF каждое приложение может использовать API, идеально соответствующий его нуждам. Если вы используете шлюз API, существует риск того, что с течением времени он раздуется по мере добавления новой функциональности. При использовании BFF этот риск снижается, поскольку он не должен охватывать всю функциональность системы — ему нужна лишь та ее часть, которая требуется для обслуживаемого ею приложения.

Нетрудно узнать, когда что-то можно удалить из BFF: это можно сделать, если не существует активной версии приложения, использующей эту функциональность. Сравните это со шлюзом API, обслуживающим несколько приложений, — вы можете что-то удалить из шлюза API только в том случае, если не существует активных версий ни одного из обслуживаемых приложений. В конечном счете BFF предлагают способ упростить разработку приложения и поддерживать серверную часть специализированной и распределенной.

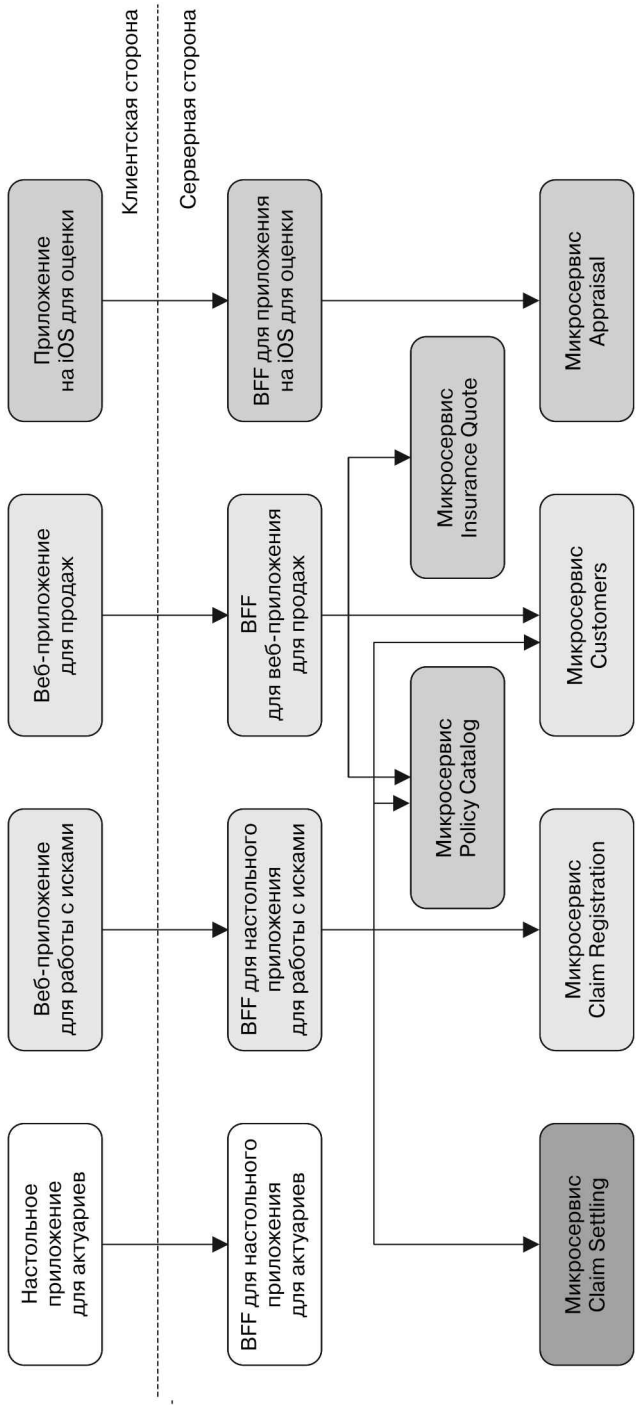


Рис. 12.6. BFF используются отдельными приложениями. BFF — это единственная точка входа для приложения, которое его использует, и поэтому оно обслуживает каждый запрос этого приложения. Он отвечает на запросы путем их делегирования другим микросервисам, отвечающим за бизнес-возможности

Недостатки. В случаях, когда вам нужно создать несколько приложений, предоставляющих конечным пользователям аналогичную или нересекающуюся функциональность, например приложения для iOS и Android, нацеленные на один тип пользователей, шаблон BFF приводит к написанию дублирующегося кода в нескольких BFF. Также проявляются обычные недостатки дубликации — вам нужно выполнять одну и ту же работу несколько раз при внесении изменений в дублирующиеся фрагменты кода, дублированные части, как правило, по прошествии времени начинают отличаться друг от друга и работать немного по-разному в разных приложениях.

В заключение заметим: шаблон BFF поможет выдержать баланс между необходимостью интегрировать микросервисы в приложение и создавать шлюз API, который может разрастись с течением времени.

Когда использовать каждый из шаблонов

Теперь, когда вы знаете о трех шаблонах для создания работающих с системой микросервисов приложений для конечных пользователей, неизбежно возникает вопрос: какой из них использовать? Все три шаблона имеют свои плюсы и приносят пользу, поэтому я не могу порекомендовать какой-то один. Когда вы создаете приложение, вам нужно сделать выбор. Я принимаю это решение на основе ответов на следующие вопросы.

- ❑ *Сколько функций вы хотите вложить в приложение?* Для бизнес-приложений, которые используются только внутри брандмауэра компании и только на машинах компании, вы можете создать настольное приложение, выполняющее большое количество функций. В этом случае очевидным выбором станет шаблон «Составное приложение».

Для приложения-магазина, которое должно запускаться в любом, в том числе старом, браузере, когда существует риск, что кто-то попытается взломать приложение, вы можете решить не помещать много функций в приложение, что делает шаблон «Составное приложение» менее привлекательным.

- ❑ *Сколько планируется приложений?* Если их будет много, насколько они отличаются друг от друга? Если вы поручили приложению немного функций и у вас будет только одно приложение или же все приложения предоставляют похожую функциональность, возможно, даже схожими способами, вам, вероятно, подойдет шлюз API. Если у вас будет несколько приложений и они предоставляют разные наборы функций, вам подойдет шаблон BFF. В случае, если вы используете шлюз API или BFF, функциональность находится на бэкенде. Шлюз API работает хорошо до тех пор, пока он остается связанным, то есть до тех пор, пока весь предоставленный им набор конечных точек соблюдает определенную систему того, как приложения должны использовать их и как они структурированы.

Если некоторые конечные точки придерживаются стиля «дистанционный вызов процедуры» (remote procedure call, RPC), а другие — стиля «передача состояния представления» (representation state transfer, REST), они не соблюдают систему и связность в базе кода шлюза API, скорее всего, будет низкой. В таких случаях вам следует задуматься о том, чтобы использовать шаблон BFF. С помощью

BFF вы можете объединить приложения, которые работают с API в стиле RPC, в одном BFF, а приложения, которые используют REST API, — в другом BFF, не нарушая связанности. Каждый BFF может быть связанным и устойчивым сам по себе, но вам не нужна связанность среди BFF с точки зрения стиля API.

- ❑ *Насколько велика система?* В большой системе — с точки зрения количества доступной функциональности — шлюз API может стать неуправляемой базой кода, которая имеет множество недостатков монолитных систем. Для подобных систем лучше использовать несколько BFF, нежели один большой шлюз API. В то же время, если система не так велика, шлюз API может оказаться проще, чем BFF.

Наконец, следует отметить, что вам не нужно принимать одинаковые решения для всех приложений. Вы можете начать со шлюза API и построить с его помощью несколько приложений, а затем решить, что новое приложение с инновационным подходом к решению задач не может работать со шлюзом API, и предоставить ему BFF. Аналогично у вас могут быть внутренние приложения, которые используют шаблон «Составное приложение», и внешние приложения, которые проходят через шлюз API, состоящий из нескольких BFF.

Отрисовка на стороне клиента или на стороне сервера?

Мы рассмотрели три шаблона для построения приложений на основе микросервисов — «Составные приложения», «Шлюз API» и BFF. Если вы строите веб-приложения с помощью этих шаблонов, возникает еще один вопрос: где выполнять отрисовку, на стороне сервера или на стороне клиента? То есть нужно ли вам генерировать готовый HTML на сервере с помощью, например, Razor (<http://mng.bz/I73n>) или же следует отрисовывать HTML в приложении JavaScript, используя один из множества фреймворков приложений JavaScript вроде Angular (<https://angularjs.org>), Ember (<http://emberjs.com>), Aurelia (<http://aurelia.io>) или React (<https://facebook.github.io/react>)?

Этот вопрос не имеет точного ответа, все зависит от того, какое приложение вы хотите построить. Насколько динамическим будет это приложение? Оно работает с данными или только отображает и принимает их? Чем более динамическим является приложение и чем больше его поток выполнения связан с манипуляциями с данными, тем больше я склоняюсь к отрисовке на стороне клиента. Если же приложение более статическое и связано скорее с просмотром и вводом данных, то я склоняюсь к отрисовке на стороне сервера. Основная идея, однако, заключается в том, что выбор способа отрисовки зависит от того, какое приложение вы хотите создать, а не от того, что вы выбрали архитектуру микросервисов на стороне сервера.

Все три шаблона поддерживают отрисовку и на стороне клиента, и на стороне сервера, вплоть до того, что некоторые части приложения отрисовываются на сервере, а некоторые — на клиенте. Например, каталог полисов в системе страхования является статическим и в большинстве случаев не изменяется, так что, скорее всего, имеет смысл отрисовывать его на стороне сервера. А калькулятор оценки является более динамическим компонентом, который позволяет пользователям изменять параметры перед сохранением окончательного результата, поэтому, возможно, его

лучше отрисовывать в приложении JavaScript. Оба модуля могут сосуществовать в одном приложении.

- ❑ Если вы создаете составное приложение, оно может получать отрисованный на сервере каталог полисов, а также приложение JavaScript для калькулятора оценки. Микросервис, ответственный за каталог полисов, предоставит сгенерированный на сервере интерфейс, а микросервис, ответственный за оценку, — приложение JavaScript.
- ❑ Если вы используете шлюз API, он может содержать конечные точки, которые возвращают HTML, и конечные точки, которые возвращают данные — например, в формате JSON. Он даже может содержать конечные точки, которые возвращают данные в форматах HTML или JSON в зависимости от заголовка Accept запроса. Поэтому приложение может содержать отрисованный на сервере каталог полисов, а также отрисованный на стороне клиента калькулятор оценки.
- ❑ Если вы используете BFF, вы также можете создавать конечные точки, которые возвращают HTML, данные или и то и другое. В дополнение к этому BFF позволяют вам принимать разные решения для разных приложений: в одном BFF каталог полисов будет отрисовываться на стороне сервера, но в другом вы можете сделать это на стороне клиента.

На выбор между отрисовкой на стороне клиента и на стороне сервера в веб-интерфейсе не влияет тот факт, что серверная сторона использует микросервисы. Все рассмотренные нами шаблоны, необходимые для построения приложений на основе микросервисов, поддерживают оба типа отрисовки.

12.3. Пример: корзина товаров и список продуктов

Рассмотрим конкретный пример и увидим код, который нужен для того, чтобы реализовать несколько фрагментов функциональности в одном приложении. В этом примере используется только один шаблон, остальные подробно не рассматриваются. Вы увидите, как объединить функциональность разных микросервисов в одном приложении. Этот принцип лежит в основе и двух других шаблонов.

В оставшейся части этой главы мы рассмотрим пример реализации корзины товаров для интернет-магазина, о котором шла речь в ранних главах. Сначала я назову этот пример, затем покажу небольшой интерфейс корзины и списка товаров, после чего вы наконец реализуете их.

В интернет-магазине пользователи могут просматривать продукты и помещать их в корзину. Когда пользователь добавляет продукт в корзину, срабатывает процесс, показанный на рис. 12.7. Происходит следующее.

1. Предмет добавляется в корзину.
2. Определяется новая сумма для содержимого корзины.
3. Микросервисы Recommendation и Shopper Tracking оповещаются об изменениях в корзине с помощью ленты событий.

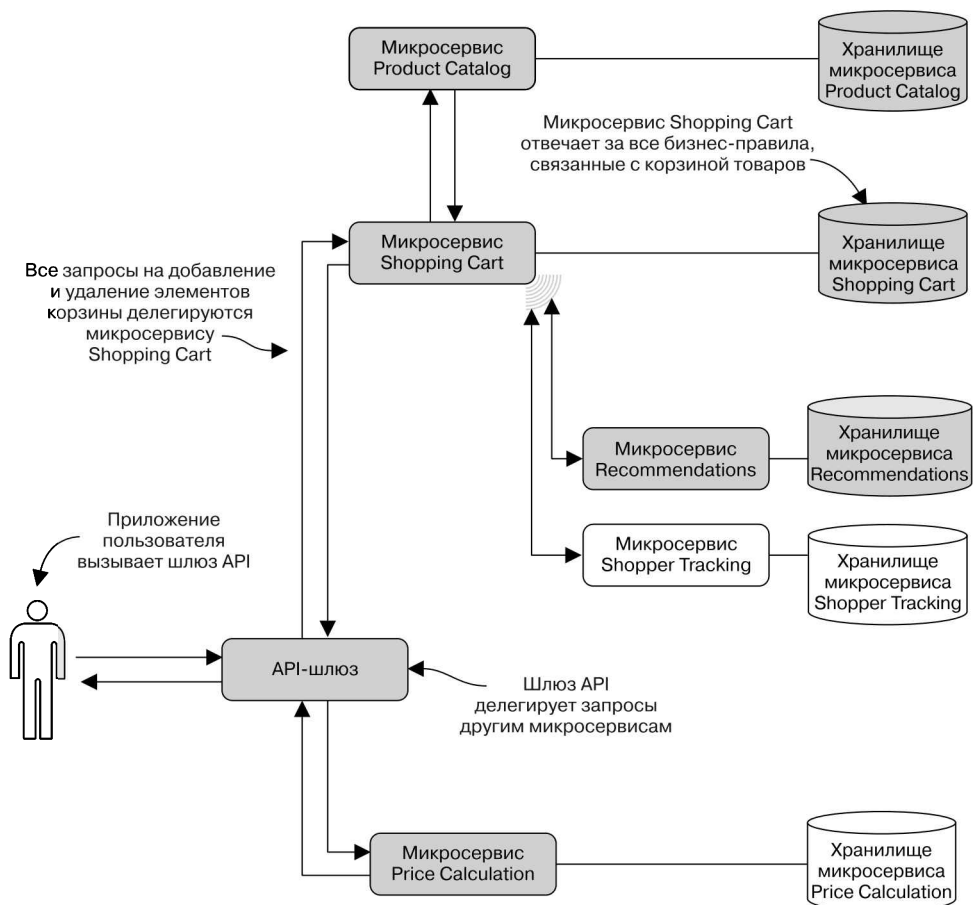


Рис. 12.7. Микросервис Shopping Cart отвечает за хранение и поддержку корзин от лица пользователей, но шлюз API предоставляет эту функциональность (вместе с другими фрагментами функциональности) конечным пользователям в рамках веб-приложения

В следующих разделах вы частично реализуете этот процесс, а также простой список продуктов, который позволяет пользователям добавлять предметы в свои корзины. На рис. 12.8 показана та часть системы, которую вы будете реализовывать.

Этого будет достаточно для того, чтобы показывать пользователям страницу, представленную на рис. 12.9. Она позволяет пользователям видеть список продуктов и добавлять продукты в корзину. Когда пользователь добавляет продукт в корзину, правая часть страницы — та, где показано содержимое корзины, — обновится и отобразит новое содержимое. Страница позволяет пользователю также удалить продукты из корзины.

Чтобы реализовать этот интерфейс, нужно сделать следующие шаги.

1. Повторно использовать микросервисы Product Catalog и Shopping Cart, показанные в главе 5.

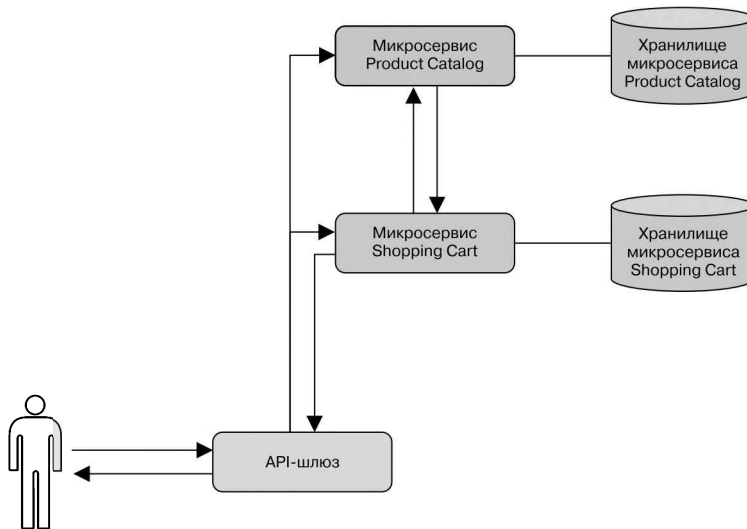


Рис. 12.8. В этом примере показаны шлюз API и микросервисы Product Catalog и Shopping Cart. Вы реализуете часть приложения с их помощью

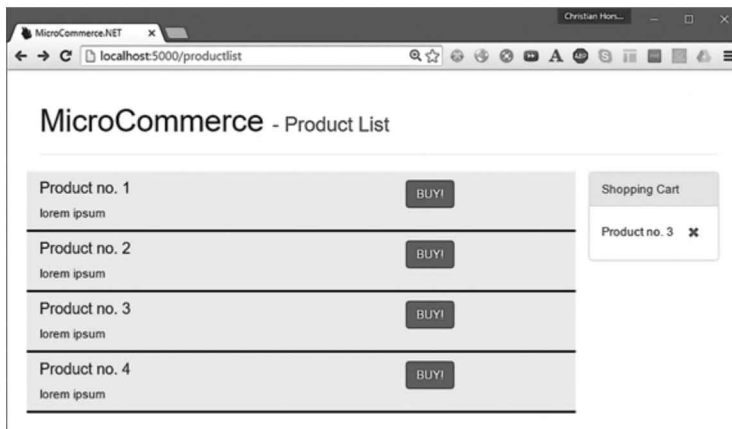


Рис. 12.9. Фрагмент приложения e-commerce, который вы реализуете, показывает список продуктов и позволяет пользователям добавлять продукты в корзину и удалять их из нее

2. Создать шлюз API.

3. Создать список продуктов на основе рис. 12.9:

- создать конечную точку в шлюзе API, которая будет получать страницу со списком продуктов;
- сделать так, чтобы новая конечная точка считывала список продуктов из микросервиса Product Catalog;
- создать представление со списком продуктов и вернуть его из новой конечной точки.

4. Добавить на веб-страницу корзину на основе рис. 12.9:
 - сделать так, чтобы API получал текущее состояние корзины из микросервиса Shopping Cart;
 - добавить корзину на веб-страницу.
5. Создать в шлюзе API конечную точку для добавления продуктов в корзину:
 - добавить конечную точку POST в шлюз API для добавления продуктов в корзину;
 - вызвать микросервис Shopping Cart из шлюза API для того, чтобы добавить новый продукт;
 - обновить представление, видимое пользователю, чтобы показать, что продукт был добавлен в корзину.
6. Создать конечную точку в API для удаления продуктов из корзины.

Вы будете использовать микросервисы Product Catalog и Shopping Cart, показанные в главе 5, в новом шлюзе API, не внося в них никаких изменений. К этому моменту можете добавить платформу микросервисов из главы 11, но для краткости мы опустим этот шаг. Вам следует убедиться, что микросервисы работают на разных портах, чтобы вы могли запустить их одновременно на той машине, где разрабатываете приложение. Порты указываете в соответствующих файлах микросервисов `project.json` — я использую порт 5100 для Product Catalog и порт 5200 для Shopping Cart.

ВНИМАНИЕ

Чтобы эта глава не разрасталась, я сделал несколько сокращений. Например, вы не будете реализовывать систему авторизации. Вместо этого жестко кодируете идентификатор пользователя. Это означает, что приложение сможет управлять корзиной только одного пользователя. В главе 10 вы увидели, как решаются проблемы, связанные с безопасностью, включая аутентификацию конечных пользователей.

Создание шлюза API

Создайте новый проект для шлюза API точно так же, как вы создавали проекты до этого момента. Назовите его ApiGateway. Далее платформу микросервисов, которую вы разработали в главе 11, добавьте в новый проект как пакет NuGet. Помните, что платформа потянет за собой несколько других пакетов вроде Nancy. Список зависимостей в файле `project.json` должен выглядеть так (листинг 12.1).

Листинг 12.1. Зависимости в шлюзе API

```
"dependencies": {
  "Microsoft.NETCore.App": {
    "version": "1.0.0-rc2-3002702",
    "type": "platform"
  },
  "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
  "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
  ..
}
```

```

"Microsoft.AspNetCore.StaticFiles": "1.0.0",
"Microsoft.AspNetCore.Owin": "1.0.0",
"Microsoft.NET.Platforms": "1.0.0",
"Serilog.Sinks.ColoredConsole": "2.0.0-beta-700"
},

```

Платформа зависит от Nancy, поэтому этот фрагмент кода также устанавливает Nancy

Далее в новом проекте измените файл `Startup.cs`, чтобы выполнить инициализацию Nancy, что вы уже делали ранее, а также инициализацию платформы микросервисов (листинг 12.2).

Листинг 12.2. Инициализируем Nancy и платформу микросервисов

```

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        var logger = ConfigureLogger();
        app.UseStaticFiles();
        app.UseOwin()
            .UseMonitoringAndLogging(logger, HealthCheck)
            .UseNancy(opt => opt.Bootstrapper = new Bootstrapper(logger));
    }

    private ILogger ConfigureLogger()
    {
        MicroservicePlatform.Configure(
            tokenUrl: "http://localhost:5001/",
            clientName: "api_gateway",
            clientSecret: "secret");
        return new LoggerConfiguration()
            .Enrich.FromLogContext()
            .WriteTo.ColoredConsole(
                LogEventLevel.Verbose,
                "{NewLine}{Timestamp:HH:mm:ss} [{Level}] ({CorrelationToken})
                ➡ {Message}{NewLine}{Exception}")
            .CreateLogger();
    }

    private static Task<bool> HealthCheck()
    {
        return Task.FromResult(true);
    }
}

public class Bootstrapper : DefaultNancyBootstrapper
{
    private ILogger logger;
    public Bootstrapper(ILogger logger)
    {
        this.logger = logger;
    }
    protected override void ApplicationStartup(

```

Конфигурирует промежуточное ПО для мониторинга, находящееся в платформе микросервисов

Конфигурирует ASP.NET Core так, чтобы он подавал файлы JavaScript и CSS из файловой системы

Предоставляет доступ к инструменту журналирования платформе микросервисов и Nancy, передавая его в загрузчик

Выполняет статическую конфигурацию платформы микросервисов

Функция-заполнитель для проверки состояния

Удерживает средство журналирования

```

    TinyIoCContainer container,
    IPipelines pipelines)
{
    container.Register(logger);
    container.UseHttpClientFactory(new NancyContext());
}

protected override void RequestStartup(
    TinyIoCContainer container,
    IPipelines pipelines,
    NancyContext context)
{
    base.RequestStartup(container, pipelines, context);
    container.UseHttpClientFactory(context);
}
}

```

Делится средством журналирования с любым модулем, которому нужна зависимость от интерфейса ILogger

Конфигурирует фабрику HttpClientFactory и регистрирует ее в контейнере

Теперь у вас есть нутстой нроект, в котором вы можете реализовать шлюз API.

Создаем интерфейс списка продуктов

Следующим шагом является создание той части приложения, в которой перечисляются нродукты. Добавьте новый модуль Нансу, который называется `GatewayModule`, в шлюз API, а затем добавьте в этот модуль конечную точку `/productlist`. `GatewayModule` будет содержать все конечные точки, которые нредоставляют доступ к приложению конечным пользователям: в нем будут находиться конечные точки, которые нредоставляют конечным пользователям веб-интерфейс, а также конечные точки, которые будет применять в этом интерфейсе JavaScript.

Чтобы проект поначалу оставался простым, начнем с конечной точки, которая не делает ничего. Затем вы добавите интерфейс, основываясь на жестко закодированном списке нродуктов, и, наконец, получите реальный список нродуктов из микросервиса Product Catalog.

`GatewayModule` будет выдавать веб-фронтенд конечным нользователям, но мы начнем с конечной точки, которая всегда возвращает пустой ответ 501 (Not Implemented) (501 Не реализовано) (листинг 12.3).

Листинг 12.3. Заполнитель реализации конечной точки в шлюзе API

```

namespace ApiGateway
{
    using System;
    using System.Threading.Tasks;
    using Nancy;

    public class GatewayModule : NancyModule
    {
        public GatewayModule()
        {
            Get("/productlist", _ => 501);
        }
    }
}

```

Следующим шагом является добавление интерфейса, который показывает список продуктов. До этого момента на протяжении всей книги вы возвращали данные, например, в формате JSON из всех конечных точек. Теперь будете возвращать интерфейс в виде сгенерированного на сервере HTML. Неудивительно, что Nancy поддерживает эту функциональность. Nancy поставляется с собственным движком, который называется Super Simple View Engine (SSVE, <http://mng.bz/ydy4>), его вы и будете использовать в этом примере.

Nancy, SSVE и Razor

Nancy позволяет использовать несколько разных движков для представления. Он поставляется с движком Super Simple View Engine (SSVE), но, если захотите применить другой движок для представления, например Microsoft's Razor, можете это сделать. Как и все остальные части модуля Nancy, движок для представления может быть заменен или дополнен другим движком. Например, чтобы использовать Razor вместе с Nancy, вы устанавливаете пакет NuGet Nancy.Viewengines.Razor, который содержит код, необходимый для того, чтобы адаптировать Razor для Nancy, который подтянет и сам Razor.

В этой главе мы будем пользоваться SSVE. Это действительно простой движок для представления, но он поддерживает передачу объекта модели в представления и использование этого объекта модели для создания простых шаблонов, например условий if, а также получения свойств модели и итерирования по перечисляемым свойствам модели. Он также поддерживает более продвинутую функциональность, например частичные представления и мастер-страницы. Синтаксис для всей функциональности в SSVE имеет префикс @, как показано в следующем примере. Сначала вы определяете тип объекта модели, использованного в примере:

```
public class MyModel
{
    public string Headline { get; set; }
    public bool SomeCondition { get; set; }
    public IEnumerable<string> SomeList { get; set; }
}
```

Теперь используете его в представлении:

```
<html>
  <body>
    <h1>@Model.Headline</h1>
    @If.SomeCondition
      <p>the condition is true</p>
    @Endif
    <ol>
      @Each.SomeList
        <li>@Current</li>
      @EndEach
    </ol>
  </body>
</html>
```

Это представление применяет свойство Headline из объекта модели в качестве заголовка, проверяет значение булевой переменной SomeCondition, отрисовывая небольшой абзац только в том случае, если оно имеет значение true, и отрисовывает упорядоченный список, добавляя маркер списка к каждой строке, в SomeList.

Чтобы конечная точка `/productlist` вернула представление, измените ее так, как показано в листинге 12.4. Она возвращает представление, которое называется `productlist`, и передает жестко закодированный список продуктов в представление как объект модели.

Листинг 12.4. Жестко закодированная реализация конечной точки в шлюзе API

```
namespace ApiGateway
{
    using System;
    using System.Threading.Tasks;
    using Nancy;

    public class GatewayModule : NancyModule
    {
        public GatewayModule()
        {
            Get("/productlist", _ =>
            {
                var products = new[] ← Жестко закодированный
                {                               список продуктов
                    new Product {ProductId = 1, ProductName = "T-shirt"},
                    new Product {ProductId = 2, ProductName = "Hoodie"},
                    new Product {ProductId = 1, ProductName = "Trousers"},
                };
                return View["productlist", new { ProductList = products }]; ←
            });
        }
    }
}
```

Возвращает представление с именем `productlist` и передает список продуктов в представление. Представление `productlist` показано в листинге 12.5

В листинге 12.4 используется та функция модуля Nancy, которую вы еще не видели, — `View["productlist", new { ProductList = products }];`. Именно так вы и возвращаете представление из модуля Nancy. Первый аргумент — это имя представления, а второй — необязательный объект модели, который будет передан представлению и может использоваться при отрисовке представления.

Чтобы реализовать представление `productlist`, создайте новый файл с именем `productlist.sshml` рядом с файлом `GatewayModule.cs`, как показано в листинге 12.5. Расширение файла `.sshml` указывает Nancy, что он содержит представление SSVE. Файл `productlist.sshml` содержит простое представление, которое итерирует по списку продуктов в объекте модели и строит из продуктов список HTML. Чтобы страница приобрела структуру, вы импортируете фреймворк Bootstrap CSS (<http://getbootstrap.com>) и добавляете несколько классов Bootstrap CSS.

Листинг 12.5. Простое представление списка продуктов

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" ← Импортирует Bootstrap и использует
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css" его для задания стиля
    >
</head>
<body>
<div>MicroCommerce.NET</div>
</body>
</html>
```

```

</head>
<body>
  <div class="container">
    <div class="page-header">
      <h1>MicroCommerce <small>- Product List</small></h1>
    </div>
    <div class="row">
      <div class="col-md-8">
        @Each.ProductList
          <div class="row"
            style="background-color: #eee; border-bottom-style: solid">
              <div class="col-md-8">
                <h4>@Current.ProductName</h4>
                <p>lorem ipsum</p>
              </div>
              <div class="col-md-4">
                <p></p>
                <button class="btn btn-primary" type="button">BUY!</button>
              </div>
            </div>
          @EndEach
        </div>
      </div>
    </div>
  </body>
</html>

```

Добавляет к странице заголовок

Итерирует по всем продуктам из списка

Добавляет строку для каждого продукта

Записывает имя каждого продукта

Конеч итерирования по продуктам

Размещает кнопку-заполнитель BUY! (Купить) для каждого продукта. Эта кнопка пока не работает

В этом коде отрисовывается список нродуктов, но эти продукты жестко закодированы в шлюзе API. Их нужно получать из микросервиса Product Catalog. Для того чтобы сделать это, измените конечную точку `/productlist` в модуле `GatewayModule` так, чтобы в ней выполнялся запрос HTTP к микросервису Product Catalog, что позволит получить список продуктов (листинг 12.6).

Листинг 12.6. Финальная реализация конечной точки в шлюзе API

```

namespace ApiGateway
{
  using System;
  using System.Collections.Generic;
  using System.Linq;
  using System.Threading.Tasks;
  using Microsoft.NET.Platforms;
  using Nancy;
  using Nancy.ModelBinding;
  using Newtonsoft.Json;
  using RestSharp;
  using Serilog;

  public class GatewayModule : NancyModule
  {
    public GatewayModule(IHttpClientFactory clientFactory, ILogger logger)
    {
      Get("/productlist", async _ =>
      {

```

Здесь внедряются HttpClientFactory и ILogger, настроенные в разделе Startup

```

var client = await
    clientFactory.Create(
        new Uri("http://localhost:5100/"),
        "product_catalog_read");
var response = await
    client.GetAsync("/products?productIds=1,2,3,4");
var content = await
    response?.Content.ReadAsStringAsync();
productList =
    JsonConvert
        .DeserializeObject<List<Product>>(content)
        .ToArray();
logger.Information(productList);
return View("productlist", new { ProductList = productList });
});
}

public class Product
{
    public string ProductName;
    public int ProductId;
}
}
}

```

Создает HttpClient и указывает ему на микросервис ProductCatalog, расположенный на порте 5100

Отправляет запрос HTTP GET микросервису ProductCatalog

Десериализует список продуктов, полученный из микросервиса ProductCatalog

Передает список продуктов в представление

Теперь, когда вы получили список продуктов из микросервиса Product Catalog, в представлении показаны правильные продукты (рис. 12.10).

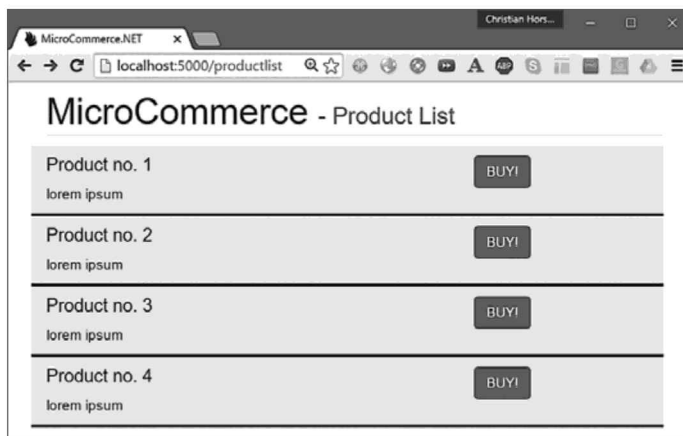


Рис. 12.10. Когда вы получили список продуктов из микросервиса Product Catalog, можете увидеть их в интерфейсе

Создание интерфейса корзины

Далее нужно создать интерфейс, отображающий содержимое корзины. Для этого расширьте конечную точку /productlist так, чтобы она вызывала не только микросервис Product Catalog, но и микросервис Shopping Cart (листинг 12.7).

Листинг 12.7. Расширяем конечную точку /productlist так, чтобы получать корзину

```

namespace ApiGateway
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Threading.Tasks;
    using System.Net.Http;
    using Microsoft.NET.Platforms;
    using Nancy;
    using Nancy.ModelBinding;
    using Newtonsoft.Json;
    using Serilog;
    using static System.Text.Encoding;

    public class GatewayModule : NancyModule
    {
        public GatewayModule(IHttpClientFactory clientFactory, ILogger logger)
        {
            Get("/productlist", async parameters =>
            {
                var userId = (int)parameters.userid;

                var client = await
                    clientFactory.Create(
                        new Uri("http://localhost:5100/"),
                        "product_catalog_read");
                var response = await
                    client.GetAsync("/products?productIds=1,2,3,4");
                var content = await response?.Content.ReadAsStringAsync();
                logger.Information(content);
                productList =
                    JsonConvert
                        .DeserializeObject<List<Product>>(content)
                        .ToArray();

                client = await
                    clientFactory.Create(new Uri(
                        "http://localhost:5200/"),
                        "shopping_cart_write");
                response = await client.GetAsync($"$/shoppingcart/{userId}");
                content = await response?.Content.ReadAsStringAsync();
                logger.Information(content);
                var basketProducts = GetBasketProductsFromResponse(content);

                return View["productlist",
                    new
                    {
                        ProductList = productList,
                        BasketProducts = basketProducts
                    }
                ];
            });
        }

        private List<Product> GetBasketProductsFromResponse(string responseBody)
        {

```

? — это оператор с условием null в C# 6

Создает новый клиент HttpClient, который вызывает микросервис ShoppingCart

Получает корзину из микросервиса ShoppingCart

Десериализует ответ микросервиса ShoppingCart

```

return
    JsonConvert
        .DeserializeObject<ShoppingCart>(responseBody)
        .Items
        ?.Select(item =>
            new Product
            {
                ProductName = item.ProductName,
                ProductId = item.ProductCatalogueId
            })
        ?.ToList() ?? new List<Product>();
}

public class Product
{
    public string ProductName;
    public int ProductId;
}

public class ShoppingCart
{
    public IEnumerable<ShoppingCartItem> Items { get; set; }
}

public class ShoppingCartItem
{
    public int ProductCatalogueId { get; set; }
    public string ProductName { get; set; }
}
}
}

```

Преобразует ответ из ShoppingCart в модель представления

Передаёт список продуктов из корзины в представление

Модель ShoppingCart модуля API GatewayModule

Далее расширим представление для отрисовки содержимого корзины в правой части страницы (листинг 12.8).

Листинг 12.8. Расширяем представление так, чтобы оно включало в себя корзину

```

<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWp
Gmkzs7"
crossorigin="anonymous">
<title>MicroCommerce.NET</title>
</head>
<body>
<div class="container">
<div class="page-header">
<h1>MicroCommerce <small> - Product List</small></h1>
</div>
<div class="row">
<div class="col-md-8">
@Each.ProductList
<div class="row"

```

```

        style="background-color: #eee; border-bottom-style: solid">
        <div class="col-md-8">
            <h4>@Current.ProductName</h4>
            <p>lorem ipsum</p>
        </div>
        <div class="col-md-4">
            <p></p>
            <button class="btn btn-primary" type="button">BUY!</button>
        </div>
    </div>
    @EndEach
</div>
<div class="col-md-4"> ← Добавляет колонку справа от корзины
    <div class="panel panel-info">
        <div class="panel-heading">Basket</div>
        <div class="panel-body">
            @Each.BasketProducts ← Итерирует по продуктам корзины
                <div>
                    @Current.ProductName
                    <button class="btn btn-link">
                        <span class="glyphicon glyphicon-remove" ← Кнопка-заполнитель
                            aria-hidden="true"></span>
                        которая выглядит как крестик.
                    </button>
                    Она пока не работает
                </div>
            @EndEach
        </div>
    </div>
</div>
</div>
</div>
</body>
</html>

```

Это представление итерирует по продуктам из корзины и показывает их все. После добавления списка продуктов и корзины представление будет выглядеть так, как на рис. 12.11.

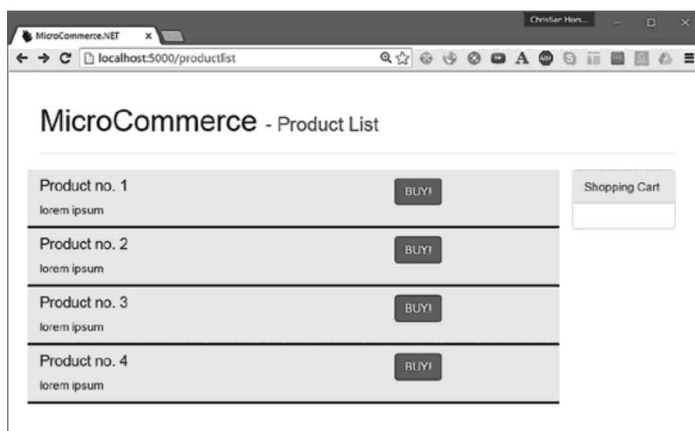


Рис. 12.11. Теперь, когда вы создали первую часть приложения, оно показывает список продуктов и пустую корзину

Теперь у вас есть полный интерфейс приложения, но нет функциональности. Кнопки BUY! не работают, как и крестики в корзине. Вы измените это в следующих двух разделах, где добавите поведение для приложения.

Позволяем пользователям добавлять продукты в корзину

Первая часть поведения, которую вы сейчас добавите, заставит работать кнопку BUY! в списке продуктов. Необходимо сделать следующее.

1. Добавьте конечную точку в модуль `GatewayModule`, который позволяет приложению добавлять продукты в корзину. Эта конечная точка в шлюзе API псевелика, она делегирует работу микросервису `Shopping Cart`.
2. Добавьте кнопке BUY! функцию `OnClick`, которая вызывает новую конечную точку в модуле `GatewayModule`.

В листинге 12.9 показана конечная точка в модуле `GatewayModule`, которая позволяет приложению добавить продукт в корзину.

Листинг 12.9. Конечная точка для добавления продукта в корзину

```
public GatewayModule(IHttpClientFactory clientFactory, ILogger logger)
{
    ...

    Post("/shoppingcart/{userid}", async parameters => ← Новая конечная точка
    {
        var productId = this.Bind<int>(); ← Считывает идентификатор
        var userId = (int) parameters.userid; ← продукта из тела запроса

        var client = await
            clientFactory.Create(
                new Uri("http://localhost:5200/"),
                "shopping_cart_write"); ← Создает клиент HttpClient
        var response = await ← для вызова микросервиса
            client.PostAsync( ← ShoppingCart
                $"/shoppingcart/{userId}/items",
                new StringContent(
                    JsonConvert.SerializeObject(new[] { productId }),
                    UTF8,
                    "application/json")); ← Отправляет запрос
        var content = await response?.Content.ReadAsStringAsync();
        var basketProducts = GetBasketProductsFromResponse(content);
        logger.Information("{@basket}", basketProducts); ← в микросервис ShoppingCart,
        return View("productlist", ← чтобы добавить продукт
            new
            {
                ProductList = productList, ← Передает
            } ← обновленный список
        ); ← продуктов из ответа
    }
}
```

```

        BasketProducts = basketProducts
    });
}

```

Эта конечная точка получает некоторые данные — идентификатор продукта — в теле запроса POST и делегирует вызов микросервису Shopping Cart, отправив ему запрос HTTP POST.

Микросервис Shopping Cart, как вы заметили в предыдущих главах, обрабатывает добавление продукта в корзину, сохранение обновленной корзины и генерирование события, которое уведомляет подписчиков об обновлении.

Чтобы использовать эту конечную точку из приложения, вам нужно добавить в представление немного кода JavaScript: функцию, которая вызывает новую конечную точку и заменяет текущую страницу страницей, возвращенной из конечной точки (листинг 12.10).

Листинг 12.10. Вызываем конечную точку для того, чтобы добавить продукт в корзину

```

    Проверяет успешность
    выполнения запроса

    function buy(productId) {
        var xhttp = new XMLHttpRequest();
        xhttp.onreadystatechange = function() {
            if (xhttp.readyState == 4 && xhttp.status == 200) {
                document.write(xhttp.responseText);
                document.close();
            }
        }
        xhttp.open('POST', '/shoppingcart/123', true);
        xhttp.setRequestHeader('Content-type', 'application/json');
        xhttp.send( JSON.stringify(productId));
    }

    Отправляет запрос POST

```

Функция, которая вызывается, когда завершается запрос к шлюзу API

Подготавливает запрос POST к шлюзу API

Заменяет содержимое текущей страницы на код HTML, взятый из ответа на запрос POST

Эта функция добавляет продукт в корзину. Ее нужно вызывать в том случае, когда пользователь нажимает одну из кнопок BUY!. Поэтому добавьте обработчик функции onclick для кнопки BUY!, заменив строку в представлении, которая отображает кнопку BUY! для каждого продукта из списка, как показано далее:

```

<html>
...
    <button class="btn btn-primary" type="button"
        onclick="buy(@Current.ProductId);" >
        BUY!
    </button>
...
</html>

```

Вызывает функцию buy с идентификатором текущего продукта при итерировании по списку продуктов

Это все, что вам нужно сделать для того, чтобы кнопки BUY! начали работать.

Позволяем пользователям удалять продукты из корзины

Последний фрагмент функциональности, который вам нужно реализовать, заключается в том, чтобы позволить пользователям удалять продукты из корзины. По аналогии с предыдущим разделом это означает, что вам нужно добавить конечную точку в модуль `GatewayModule` и обработчик события `onclick` для кнопок с крестиком в той части приложения, которая отвечает за корзину. Добавьте конечную точку **DELETE**, которая в основном служит для делегирования работы микросервису `Shopping Cart` (листинг 12.11).

Листинг 12.11. Конечная точка для удаления продукта из корзины

```
public GatewayModule(IHttpClientFactory clientFactory, ILogger logger)
{
    ...

    Delete("/shoppingcart/{userid}", async parameters =>
    {
        var productId = this.Bind<int>();
        var userId = (int) parameters.userid;

        HttpClient client = await
            clientFactory.Create(
                new Uri("http://localhost:5200/"),
                "shopping_cart_write");
        var request =
            new HttpRequestMessage( ← Подготавливает запрос DELETE
                                   для удаления продукта из корзины
                HttpMethod.Delete,
                $"/shoppingcart/{userId}/items")
            {
                Content = new StringContent(
                    JsonConvert.SerializeObject(new[] { productId }),
                    UTF8, "application/json")
            };
        var response = await client.SendAsync(request); ← Отправляет
        var content = await response?.Content.ReadAsStringAsync();
        var basketProducts = GetBasketProductsFromResponse(content);
        logger.Information("{@basket}", basketProducts);
        return View["productlist",
            new
            {
                ProductList = productList,
                BasketProducts = basketProducts
            }
        ];
    });
}
```

Чтобы использовать эту конечную точку, добавьте в представление еще одну функцию JavaScript (листинг 12.12).

Листинг 12.12. Вызываем конечную точку для того, чтобы удалить продукт из корзины

```
function removeFromBasket(productId) {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (xhttp.readyState == 4 && xhttp.status == 200) {
      document.write(xhttp.responseText);
      document.close();
    }
  }
  xhttp.open('DELETE', '/shoppingcart/123', true);
  xhttp.setRequestHeader('Content-type', 'application/json');
  xhttp.send(JSON.stringify(productId));
}
```

Готовится отправить
запрос DELETE
в шлюз API

Далее используйте эту функцию JavaScript из кнопки с крестиком в той части представления, которая отвечает за корзину:

```
<html>
...
  <button class="btn btn-link"
    onclick="removeFromBasket(@Current.ProductId);" >
    <span class="glyphicon glyphicon-remove"
      aria-hidden="true"></span>
  </button>
...
</html>
```

Вызывает функцию removeFromBasket,
передавая ей идентификатор продукта
для текущего productSummary

После размещения этого кода пример приложения начнет работать! Пользователь может добавить продукты или удалить их из корзины.

12.4. Резюме

- ❑ Существует целый спектр видов приложений, которые можно построить на основе системы микросервисов, от универсальных приложений, охватывающих всю функциональность системы, до небольших специализированных приложений.
- ❑ Приложения на основе микросервисов могут быть построены как составные, которые получают функциональность и компоненты интерфейса из различных микросервисов и объединяют их для того, чтобы сформировать цельное приложение.
- ❑ Шаблон «Составное приложение» позволяет микросервисам оставаться несвязанными, но само составное приложение может стать более сложным.
- ❑ Приложения на основе микросервисов могут быть построены с помощью шаблона «Шлюз API», в рамках которого вы размещаете один универсальный API перед всеми микросервисами. Этот шлюз API является единственным микросервисом, который приложение использует непосредственно, но он делегирует все запросы другим микросервисам, в которых реализованы бизнес-возможности.

- ❑ Шаблон «Шлюз API» может упростить разработку приложения и отвязать его от архитектуры серверной стороны.
- ❑ Шлюз API может увеличиваться с течением времени, поскольку он должен предоставлять доступ ко всей функциональности. Это особенно верно, если несколько приложений используют один шлюз API, поскольку ему нужно поддерживать все сценарии во всех приложениях.
- ❑ Приложения на основе микросервисов могут быть построены на основе шаблона «Бэкенд для фронтенда». BFF — это микросервис, который ведет себя как шлюз API, но только для одного приложения.
- ❑ BFF менее подвержены неконтролируемому росту, в отличие от шлюзов API.
- ❑ BFF идеально подходят одному приложению, которое их использует, они должны максимально упрощать разработку таких приложений.
- ❑ Когда вы создаете веб-приложения на основе микросервисов, вы можете использовать отрисовку на стороне сервера, отрисовку на стороне клиента или же оба подхода сразу.
- ❑ Все три шаблона — «Составное приложение», «Шлюз API» и BFF — поддерживают отрисовку на стороне сервера, отрисовку на стороне клиента, а также комбинацию этих подходов.
- ❑ Реализация шлюза API велика: все конечные точки, которые вы добавили в пример шлюза API, делегируют работу другим микросервисам.
- ❑ Nancy позволяет возвращать отрисованный на стороне клиента HTML с помощью встроенного движка Super Simple View Engine или движка Razor.

Приложения

Приложение А. Настройка среды разработки

В этом приложении описаны настройки среды разработки, необходимые для работы с содержащимися в книге примерами кода. Среда разработки состоит из пяти частей.

- ❑ *IDE*. Вы можете выбрать между Visual Studio 2015 или более новой ее версией¹, Visual Studio Code, ATOM или JetBrains Rider. Visual Studio 2015 рассчитана только на операционную систему Windows, а остальные три работают в операционных системах Windows, OS X и Linux.
- ❑ *dotnet*. Вам понадобится утилита командной строки dotnet.
- ❑ *Генератор Yeoman для платформы ASP.NET*. Необходим для создания проектов ASP.NET Core. При использовании Visual Studio можно создавать проекты с ее помощью, но в этой книге применяется Yeoman. Шаблоны проектов Visual Studio и Yeoman похожи, но не одинаковы.
- ❑ *Postman*. Вам необходим инструмент для выполнения HTTP-запросов. Существует множество подобных утилит, включая cURL и Fiddler, но я рекомендую использовать Postman, удобный и работающий в операционных системах Windows, OS X и Linux.
- ❑ *СУБД с SQL*. В операционной системе Windows можно использовать SQL Server (что мы и делаем в данной книге), но если хотите, можете воспользоваться другой СУБД, например PostgreSQL.

В следующих разделах рассмотрим установку и подготовку к работе среды разработки.

А.1. Установка IDE

Для кода из этой книги можно использовать четыре IDE. Какую из них выбрать — дело вкуса, все они отлично работают со всеми примерами из этой книги. Лично я попеременно переключался между всеми четырьмя при написании кода для книги.

¹ Из-за изменений формата конфигурации проектов в версии Visual Studio 2017, по сравнению с предыдущими версиями, некоторые из приводимых в этой книге инструкций могут оказаться не актуальными для нее. — *Примеч. пер.*

Visual Studio 2015

Visual Studio — традиционный вариант IDE для разработки программ на языке C#. Она включает все, что должно быть в IDE. Что касается кода из этой книги, Visual Studio предоставляет отличный редактор для кода на языке C#, технологию IntelliSense для файлов `project.json`, управление пакетами с помощью NuGet, а также запуск и отладку приложений ASP.NET Core.

Существует несколько версий Visual Studio 2015. В бесплатной версии Visual Studio 2015 Community имеется все необходимое для работы с примерами. Для скачивания Visual Studio 2015 Community перейдите по ссылке <http://mng.bz/7i8F> и выберите Visual Studio Community¹. Скачав программу установки, запустите ее и следуйте инструкциям.

После установки Visual Studio 2015 необходимо для поддержки ASP.NET Core в Visual Studio установить плагин .NET Core для Visual Studio. Найти ссылку на свежую версию этого плагина² можно по адресу <http://mng.bz/nvpd>. Он обеспечивает работу Visual Studio с ASP.NET Core и предоставляет шаблоны проектов, а также обеспечивает использование технологии IntelliSense в файлах `project.json`, автоматическое восстановление пакетов NuGet при редактировании файла `project.json` и возможность отладки приложений ASP.NET Core.

Visual Studio Code

Утилита Visual Studio Code — облегченная кросс-платформенная альтернатива Visual Studio 2015. Спектр ее возможностей не так широк, как у Visual Studio, но с установленным расширением для C# она отлично работает в тех проектах, которые вы будете создавать при чтении этой книги, — приложениях ASP.NET Core. Visual Studio Code предоставляет отличный редактор языка C# и обеспечивает использование технологии IntelliSense в файлах `project.json`, управление пакетами с помощью NuGet, а также возможность запуска и отладки приложений ASP.NET Core.

Скачать Visual Studio Code можно с сайта <https://code.visualstudio.com/>. Нажмите кнопку **Download** (Скачать), чтобы скачать подходящую для вашей операционной системы программу установки. Запустите программу установки и следуйте инструкциям.

Вам понадобится также плагин C# для Visual Studio Code. Чтобы установить его, нажмите в Visual Studio Code комбинацию клавиш **Ctrl+P**, которая откроет панель

¹ После выхода Visual Studio 2017 скачивание Visual Studio 2015 Community подобным образом невозможно. Для удобства читателя приведем прямые ссылки на ISO-образы установочных пакетов: англоязычная версия (7,1 Гбайт) — <https://go.microsoft.com/fwlink/?LinkId=615448&clcid=0x409>; русскоязычная версия (7,1 Гбайт) — http://download.microsoft.com/download/b/c/a/bca512a6-bf63-4425-a65a-219ca620d67d/vs2015.3.com_rus.iso. Обращаем внимание на то, что эти образы не всеобъемлющи — некоторые утилиты можно в дальнейшем скачивать отдельно. — *Примеч. пер.*

² Для полной совместимости с исходными кодами из книги рекомендуется использовать версию 1.0.1. — *Примеч. пер.*

VS Code Quick Open. Введите там команду `ext install csharp` и нажмите клавишу Enter для установки расширения для языка C#.

Редактор ATOM

ATOM — кросс-платформенный редактор с открытым исходным кодом, широко используемый для множества различных языков программирования. Чтобы он работал с языком программирования C#, необходимо установить плагин `omnisharp-atom`, обеспечивающий использование технологии IntelliSense в коде на языке C# и в файлах `project.json`, рефакторинг, возможность исправления кода, автоматическое восстановление пакетов NuGet при редактировании файла `project.json` и возможность запуска приложений ASP.NET Core. На момент написания данной книги возможности отладки приложений ASP.NET Core в редакторе ATOM нет.

Загрузить редактор ATOM можно с сайта <https://atom.io/>. Нажмите кнопку **Download** и следуйте инструкциям. После установки ATOM вы сможете получить плагин `omnisharp-atom` с помощью имеющейся в ATOM системы управления пакетами: выберите **File ▶ Settings** (Файл ▶ Настройки), на странице **Settings** (Настройки) выберите **Packages** (Пакеты) и введите `omnisharp-atom` в поисковом поле. Ниже строки **Community Packages** (Общие пакеты) должен появиться пакет `omnisharp-atom`, и вы сможете установить его нажатием **Install** (Установить).

При установленном плагине `omnisharp-atom` можно будет открыть каталог с файлом `project.json`, и ATOM распознает его как проект .NET Core и обеспечит поддержку C#, NuGet, `project.json` и `dotnet`.

IDE JetBrains Rider

На момент написания данной книги IDE Rider от компании JetBrains можно получить только по программе предварительного доступа, подав заявку на участие в которой можно на сайте <http://www.jetbrains.com/>. Rider — полностью отработанная кросс-платформенная интегрирующая среда разработки для языка C#, основанная на платформе IDE IntelliJ и плагине ReSharper для Visual Studio — проверенных программных продуктах компании JetBrains. Rider даже в предварительной версии предоставляет возможности отличной навигации по коду на языке C#, рефакторинга, исправления кода и применения технологии IntelliSense. В Rider нет хорошей поддержки отладки и системы управления пакетами NuGet для платформы .NET Core, по обоим этим пунктам, а также интегрированная утилита выполнения тестов значатся в «дорожной карте» Rider.

A.2. Установка интерфейса командной строки dotnet

Утилита командной строки используется на протяжении всей этой книги для запуска микросервисов, восстановления пакетов NuGet, создания пакетов NuGet

и запуска тестов. Для ее установки¹ перейдите на сайт <http://dot.net/>, щелкните на **Download .NET Core** (Скачать .NET Core) и следуйте инструкциям, соответствующим предпочтительной для вас платформе разработки.

А.3. Установка генератора Yeoman ASP.NET

Yeoman — утилита для скаффолдинга, основанная на платформе Node.js. Это универсальный инструмент, который можно использовать для генерации как проектов ASP.NET Core, так и проектов для многих других стеков технологий. Чтобы установить его и генерировать проекты ASP.NET Core, вам понадобятся Node.js, Yeoman и плагин ASP.NET-генератор.

Для установки Node.js с помощью программы установки или из tar-архива перейдите по адресу <https://nodejs.org/en/download> и следуйте инструкциям. Если вы предпочитаете устанавливать его с помощью системы управления пакетами, например `apt-get`, `Homebrew` или `Chocolatey`, перейдите по адресу <https://nodejs.org/en/download/package-manager> и следуйте инструкциям.

Установка платформы Node.js приведет также к установке `npm` — системы управления пакетами для Node. Воспользуемся `npm` для установки сначала Yeoman, а затем ASP.NET-генератора. Для установки утилиты Yeoman выполните в командной строке следующую команду:

```
PS> npm install -g yo
```

Теперь можно запустить Yeoman из командной строки с помощью команды `yo`. Для установки плагина ASP.NET-генератора к Yeoman выполните из командной строки следующее²:

```
PS> npm install -g generator-aspnet
```

Теперь вы можете генерировать проекты ASP.NET Core путем выполнения команды `yo aspnet` из командной строки.

А.4. Установка Postman

Postman — удобная утилита с графическим интерфейсом, предназначенная для работы с HTTP-запросами. С ее помощью можно создать любой HTTP-запрос и увидеть все подробности ответа. Это удобно при работе с различными API для протокола HTTP, то есть тем, что в этой книге встречается очень часто. С ее помощью можно также тестировать API для протокола HTTP напрямую и просматривать ответ в случае как успешного, так и неудачного выполнения.

¹ В силу изложенных в предыдущих примечаниях соображений совместимости с кодом примеров из книги рекомендуем использовать версию не выше 1.0.1, которую можно найти в архиве выпусков [dotnet](https://github.com/dotnet/core/blob/master/release-notes/download-archive.md) по адресу <https://github.com/dotnet/core/blob/master/release-notes/download-archive.md>. — *Примеч. пер.*

² Для совместимости с кодом примеров вновь рекомендуем использовать не последнюю версию. Необходимая для установки версии 0.2.4 команда будет выглядеть так: `npm install -g generator-aspnet@0.2.4`. — *Примеч. пер.*

Скачать Postman можно с сайта <http://www.getpostman.com/>. После установки его можно сразу запустить и приступить к выполнению HTTP-запросов. Рисунок А.1, который можно увидеть также в главе 1, демонстрирует создание HTTP-запросов с помощью утилиты Postman.

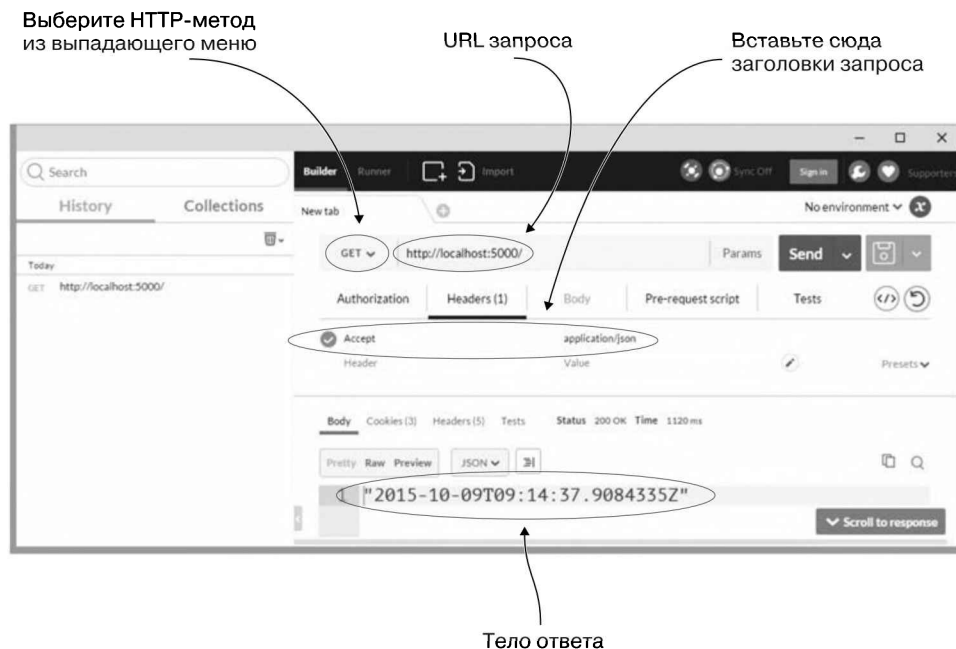


Рис. А.1. Postman облегчает отправку HTTP-запросов и управление их нюансами, например заголовками и HTTP-методом

А.5. Установка SQL Server Express

В главе 5 мы начнем использовать СУБД SQL Server. Скачать программу установки для бесплатной версии сервера базы данных SQL Server Express можно вот отсюда: <http://mng.bz/090m>. Скачайте программу установки, запустите ее и следуйте инструкциям.

Если вы работаете в операционной системе, отличной от Windows, рекомендую воспользоваться PostgreSQL. Применяемую в главе 5 для связи с SQL Server библиотеку Dapper можно столь же легко использовать для связи с PostgreSQL. Для установки PostgreSQL следуйте инструкциям к вашей операционной системе с сайта <http://www.postgresql.org/download>.

Приложение Б. Развертывание для эксплуатации в производственной среде

В этом приложении рассмотрим основные варианты выполнения разработанных на протяжении данной книги микросервисов в производственной среде. Основной фактор, определяющий, где и как можно будет развертывать эти микросервисы, — то, что они основаны на платформе .NET Core.

Обсуждаемые тут микросервисы включают две разновидности процессов: различные API для протокола HTTP и потребителей событий. API для протокола HTTP основываются на платформе ASP.NET Core и поэтому выполняются поверх веб-сервера Kestrel. Потребители событий представляют собой консольные приложения .NET Core.

В силу того что платформа .NET Core может работать под управлением как Windows, так и Linux, это возможно и для ваших событий. Аналогично микросервисы могут работать как на ваших серверах, так и в облаке — Amazon или Azure.

В.1. Развертывание API для протокола HTTP

Все API для протокола HTTP в описываемых в этой книге микросервисах основаны на фреймворке Nancy и выполняются на платформе ASP.NET Core. Это значит, что они выполняются поверх веб-сервера Kestrel. Kestrel не является полнофункциональным веб-сервером, это маленький, быстрый веб-сервер, ориентированный на выдачу динамического контента от приложений ASP.NET Core. При использовании Kestrel в производственной среде рекомендуется размещать его за обратным прокси-сервером. Обратный прокси-сервер может выполнять то, что не умеет Kestrel, например выдавать статические файлы, выполнять SSL termination и сжатие ответов. Эта архитектура в операционных системах Windows/Linux показана на рис. В.1.

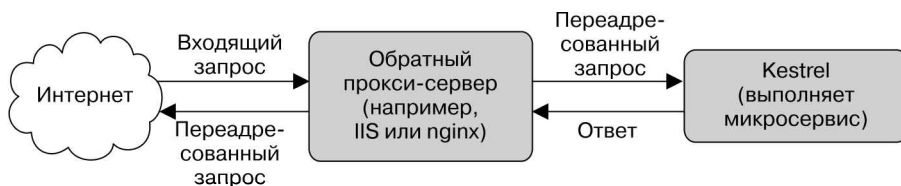


Рис. В.1. Kestrel должен работать за обратным прокси-сервером

Для серверов с операционной системой Windows

Для развертывания API микросервисов для протокола HTTP на сервере с операционной системой Windows необходимо сделать следующее.

1. Перейти в проект микросервиса и создать пакет развертывания с помощью команды `dotnet publish`.
2. Установить на сервере IIS.
3. Установить в IIS пакет .NET Core Windows Server Hosting.
4. Включить в проект микросервиса файл `web.config` с целью конфигурации `AspNetCoreModule` для обработки всех запросов. Файл `web.config` уже включен в проекты, созданные с помощью утилиты Yeoman.
5. Создать в IIS веб-сайт, ссылающийся на этот пакет развертывания и использующий пул приложений, свойство .NET CLR version которого установлено в значение `No Managed Code` (Без управляемого кода).

Подробную документацию по этому вопросу можно найти по адресу <http://mng.bz/YA0j>. Описанное ранее подойдет как для вашего собственного сервера с операционной системой Windows, так как облачных серверов с операционной системой Windows на Amazon и Azure.

Для серверов с операционной системой Linux

Для развертывания API микросервисов для протокола HTTP на сервере с операционной системой Linux, следуйте такой инструкции.

1. Измените код инициализации в файле `program.cs` так, чтобы обеспечить прослушивание Unix-сокета, например `http://unix:/var/microservicenet/hello/kestrel.sock`. А именно: добавьте вызов метода `UseUrls` интерфейса `IWebHostBuilder`. Интерфейс `IWebHostBuilder` уже создан в файле `program.cs` и сконфигурирован во всех остальных отношениях так, что вызов метода `UseUrls` будет просто еще одним в существующей цепочке вызовов.
2. Перейдите в проект вашего микросервиса и создайте пакет развертывания с помощью команды `dotnet publish`.
3. Установите на сервере обратный прокси-сервер `nginx`.
4. Сконфигурируйте сайт на `nginx` так, чтобы все входящие запросы переадресовывались на прослушиваемый микросервисом Unix-сокет.
5. Настройте автоматический запуск демона-супервизора при загрузке сервера для мониторинга микросервиса.

Подробную документацию по этому вопросу можно найти по адресу <http://mng.bz/ZWNh>. Изложенное подойдет как для вашего собственного сервера с операционной системой Linux, так и облачных серверов с операционной системой Linux на Amazon и Azure.

Azure Web Apps

Облачная платформа Azure предлагает услугу уровня PaaS для выполнения веб-приложений на платформе .NET Core — Azure Web Apps (Веб-приложения Azure). Благодаря этому вам не нужно будет самим заниматься установкой обратного прокси-сервера — это делает Azure. Требуется лишь создать на Azure веб-приложение, например, посредством портала Azure. После этого можно будет выполнить развертывание веб-приложения путем создания пакета развертывания с помощью команды `dotnet publish` и скопировать его на Azure по FTP или с помощью множества других вариантов развертывания, включая размещение его в репозитории Git. Подробную документацию по этому вопросу можно найти по адресу <http://mng.bz/1ubQ>.

Azure Service Fabric

Еще одна предлагаемая платформой Azure услуга уровня PaaS носит название Azure Service Fabric и тоже сможет обеспечить работу API для протокола HTTP, основанных на фреймворке Nancy. Для этого вам необходимо будет использовать не тот шаблон проекта, который вы применяли на всем протяжении книги (см. документацию по адресу <http://mng.bz/1WyY>). Когда проект будет готов для развертывания на Azure Service Fabric, вы сможете установить там фреймворк Nancy и другие пакеты NuGet и делать все, что вы делали при чтении этой книги с помощью фреймворков Nancy, Polly и промежуточного ПО OWIN.

В.2. Развертывание потребителей событий

Разработанные в главах 4, 6 и 7 потребители событий представляют собой консольные приложения на платформе .NET Core, которые можно запускать как сервисы операционной системы Windows, так что они работают только на Windows. Потребители событий работают только на Windows именно потому, что они представляют собой Windows-сервисы. Однако весь код для выборки событий из потока, отслеживания того, какие события уже были обработаны, а также обработка событий может точно так же работать в операционной системе Linux. Единственное, что придется реализовать по-другому для запуска на Linux, — класс `ServiceBase`: код, который реагирует на сигналы запуска и остановки от Windows, когда сервис Windows запускается или останавливается. В следующих разделах описаны некоторые альтернативы использованию Windows-сервисов для размещения кода потребителей событий.

Для серверов с операционной системой Windows

На серверах под управлением операционной системы Windows можно установить потребителя событий из главы 4 как Windows-сервис так, как описано в главе 4. Никаких изменений в его коде не требуется.

Для серверов с операционной системой Linux

В Linux потребителя событий можно запустить с помощью демона-супервизора примерно так же, как можно запустить в ней расположенный на веб-сервере Kestrel API для протокола HTTP. Необходимо только немного изменить код: вместо реализации класса `ServiceBase` и запуска таймера в методе `OnStart` таймер нужно запустить в начале выполнения, в методе `Main`.

Для подготовки к запуску потребителя событий под управлением демона-супервизора сначала выполните команду `dotnet publish` в проекте потребителя событий. Далее создайте файл конфигурации супервизора для потребителя событий в каталоге `/etc/supervisor/conf.d/` с примерно следующим содержанием:

```
[program:eventconsumer]
command=dotnet /var/microservicesnet/eventconsumer/eventconsumer.dll
autostart=true
autorestart=true
stderr_logfile=/var/log/eventconsumer.err.log
stdout_logfile=/var/log/eventconsumer.out.log
environment=Hosting__Environment=Production
user=www-data
stopsignal=INT
```

← Путь к библиотеке DLL,
созданной
с помощью команды
<data></data>dotnet publish

Затем вам нужно будет перезапустить демон-супервизор, после чего он подхватит новые настройки и запустит потребитель событий. Перезапустить демон-супервизор можно с помощью следующих двух команд:

```
sudo service supervisor stop
sudo service supervisor start
```

Теперь потребитель событий работает под управлением демона-супервизора.

Azure WebJobs

Сервис Azure WebJobs предоставляет возможность запуска на Azure консольных приложений по расписанию (см. <http://mng.bz/R4m9>). Для этого необходимо удалить реализацию класса `ServiceBase` и выполнять набор команд по обработке события при каждом запуске консольного приложения. Планировщик платформы Azure берет на себя запуск потребителя событий по расписанию. После запуска потребитель событий может делать все, что ему нужно, включая последовательный опрос потока событий и использование хранилища данных для отслеживания обработанных событий, а также выполнять любую другую связанную с обработкой событий логику.

Функции Azure

Сервис Azure Functions («Функции Azure») — в некотором смысле облегченная версия сервиса Azure WebJobs, представляющего собой часть расширенного предложения Azure Service Fabric. Azure Functions также позволяет вам выполнять код .NET в соответствии с управляемым Azure расписанием. Документацию по настройке Azure Functions можно найти по адресу <http://mng.bz/h7D9>.

Чтобы запустить потребителя событий как функцию Azure, необходимо заменить его кодом для выполнения наметного сценария по обработке событий: кодом для опроса потока событий и отслеживанию обработанных событий, а также бизнес-логикой обработки событий. Платформа Azure берет на себя вызов функции по расписанию.

Amazon Lambda

Сервис Amazon Lambda похож на сервис Azure Functions. Тут тоже необходимо заменить код потребителя событий кодом для выполнения наметного сценария по обработке событий. Документацию по настройке Amazon Lambda для выполнения кода C# по расписанию можно найти по адресу <http://mng.bz/5wJd>.

Дополнительная литература

Микросервисы

- ❑ *Cramon J.* Microservices: It's not (only) the size that matters, it's (also) how you use them. Ч. 1–5. 2014–2015 [Электронный ресурс]. — Режим доступа: <http://mng.bz/zQ2a>.
- ❑ *Fowler M.* MicroservicePrerequisites, 2014. — August 28 [Электронный ресурс]. — Режим доступа: <http://martinfowler.com/bliki/MicroservicePrerequisites.html>.
- ❑ *MonolithFirst*, 2015. — June 3 [Электронный ресурс]. — Режим доступа: <http://martinfowler.com/bliki/MonolithFirst.html>.
- ❑ *Lewis J., Fowler M.* Microservices, 2014. — March 25 [Электронный ресурс]. — Режим доступа: <http://martinfowler.com/articles/microservices.html>.
- ❑ *Newman S.* Building Microservices: Designing Fine-Grained Systems. — O'Reilly Media, 2015.
- ❑ *Pattern: Backends for Frontends*, 2015. — November 18 [Электронный ресурс]. — Режим доступа: <http://samnewman.io/patterns/architectural/bff>.
- ❑ *Tilkov S.* Don't start with a monolith, 2015. — June 9 [Электронный ресурс]. — Режим доступа: <http://martinfowler.com/articles/dontstart-monolith.html>.

Проектирование программного обеспечения и архитектуры в целом

- ❑ *Бек К.* Экстремальное программирование: разработка через тестирование. — СПб.: Питер, 2017.
- ❑ *Вон В.* Реализация методов предметно-ориентированного проектирования. — М.: Вильямс, 2017.
- ❑ *Хамбл Д., Фарли Д.* Непрерывное развертывание ПО: автоматизация процессов сборки, тестирования и внедрения новых версий программ. — М.: Вильямс, 2017.
- ❑ *Хоп Г., Вульф Б.* Шаблоны интеграции корпоративных приложений. Проектирование, создание и развертывание решений, основанных на обмене сообщениями. — М.: Вильямс, 2016.
- ❑ *Эванс Э.* Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. — М.: Вильямс, 2017.

- ❑ *Conway M. E.* How Do Committees Invent? Datamation, 1968. — April [Электронный ресурс]. — Режим доступа: www.melconway.com/research/committees.html.
- ❑ Defense in Depth. Open Web Application Security Project (OWASP) [Электронный ресурс]. — Режим доступа: www.owasp.org/index.php/Defense_in_depth.
- ❑ *Fowler M.* TestPyramid, 2012. — May 1 [Электронный ресурс]. — Режим доступа: <http://martinfowler.com/bliki/TestPyramid.html>.
- ❑ *Freeman Steve and Pryce Nat.* Growing Object-Oriented Software, Guided by Tests. — Addison-Wesley Professional, 2009.
- ❑ IntegrationContractTest, 2011. — January 12 [Электронный ресурс]. — Режим доступа: <http://martinfowler.com/bliki/IntegrationContractTest.html>.
- ❑ *Martin R. C.* The Single Responsibility Principle, 2014. — May 5 [Электронный ресурс]. — Режим доступа: <http://mng.bz/RZgU>.
- ❑ *Nygard M. T.* Release It! Design and Deploy Production-Ready Software. — Pragmatic Programmers, 2007.
- ❑ SRP: The Single Responsibility Principle [Электронный ресурс]. — Режим доступа: <http://mng.bz/zQyz>.

Список использованных в книге технологий

- ❑ ASP.NET Core (www.asp.net/core).
- ❑ Dapper (<http://mng.bz/7LHZ>).
- ❑ Elasticsearch (<https://info.elastic.co/Getting-Started-ES.html>).
- ❑ Event Store (<https://geteventstore.com>).
- ❑ IdentityServer (<https://identityserver.github.io/Documentation>).
- ❑ Kibana (www.elastic.co/products/kibana. Nancy. <http://nancyfx.org>).
- ❑ Nancy documentation (<https://github.com/NancyFx/Nancy/wiki/Documentation>).
- ❑ Nancy samples (<https://github.com/NancyFx/Nancy/tree/master/samples>).
- ❑ .NET Core (<https://dotnet.github.io>).
- ❑ NuGet documentation (<http://docs.nuget.org>).
- ❑ OAuth (<http://oauth.net/2>).
- ❑ OpenID Connect (<http://openid.net/connect>).
- ❑ OWIN standard (<http://owin.org>).
- ❑ Polly documentation (<https://github.com/App-vNext/Polly#polly>).
- ❑ Serilog (<https://serilog.net>).
- ❑ xUnit (<https://xunit.github.io>).