

Микросервисы

Паттерны
разработки
и рефакторинга

Крис
Ричардсон



MANNING



Microservices Patterns

WITH EXAMPLES IN JAVA

CHRIS RICHARDSON



MANNING
SHELTER ISLAND

Крис Ричардсон

Микросервисы

**Паттерны разработки
и рефакторинга**



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

2019

Ричардсон Крис

P56 Микросервисы. Паттерны разработки и рефакторинга. — СПб.: Питер, 2019. — 544 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-0996-8

Если вам давно кажется, что вся разработка и развертывание в вашей компании донельзя замедлились — переходите на микросервисную архитектуру. Она обеспечивает непрерывную разработку, доставку и развертывание приложений любой сложности.

Книга, предназначенная для разработчиков и архитекторов из больших корпораций, рассказывает, как проектировать и писать приложения в духе микросервисной архитектуры. Также в ней описано, как делается рефакторинг крупного приложения — и монолит превращается в набор микросервисов.

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

ББК 32.988.02-018

УДК 004.738.5

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

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1617294549 англ.

ISBN 978-5-4461-0996-8

© 2019 by Chris Richardson. All rights reserved

© Перевод на русский язык ООО Издательство «Питер», 2019

© Издание на русском языке, оформление ООО Издательство «Питер», 2019

© Серия «Библиотека программиста», 2019

Краткое содержание

Предисловие.....	16
Благодарности.....	19
О книге.....	21
Глава 1. Побег из монолитного ада	26
Глава 2. Стратегии декомпозиции	63
Глава 3. Межпроцессное взаимодействие в микросервисной архитектуре	97
Глава 4. Управление транзакциями с помощью повествований	146
Глава 5. Проектирование бизнес-логики в микросервисной архитектуре.....	185
Глава 6. Разработка бизнес-логики с порождением событий	223
Глава 7. Реализация запросов в микросервисной архитектуре	264
Глава 8. Шаблоны внешних API.....	301
Глава 9. Тестирование микросервисов, часть 1	343
Глава 10. Тестирование микросервисов, часть 2	373
Глава 11. Разработка сервисов, готовых к промышленному использованию.....	405
Глава 12. Развертывание микросервисов	446
Глава 13. Процесс перехода на микросервисы	495

Оглавление

Предисловие.....	16
Благодарности.....	19
О книге.....	21
Кому следует прочитать эту книгу	21
Структура издания.....	21
О коде.....	23
Онлайн-ресурсы	23
Об авторе.....	23
Об иллюстрации на обложке	24
От издательства	25
Глава 1. Побег из монолитного ада	26
1.1. Медленным шагом в монолитный ад	27
1.1.1. Архитектура приложения FTGO.....	28
1.1.2. Преимущества монолитной архитектуры.....	29
1.1.3. Жизнь в монолитном аду	29
1.2. Почему эта книга актуальна для вас	32

1.3. Чему вы научитесь, прочитав эту книгу.....	33
1.4. Микросервисная архитектура спешит на помощь	34
1.4.1. Куб масштабирования и микросервисы	34
1.4.2. Микросервисы как разновидность модульности	37
1.4.3. У каждого сервиса есть своя база данных.....	38
1.4.4. Микросервисная архитектура для FTGO	38
1.4.5. Сравнение микросервисной и сервис-ориентированной архитектур	40
1.5. Достоинства и недостатки микросервисной архитектуры.....	41
1.5.1. Достоинства микросервисной архитектуры	41
1.5.2. Недостатки микросервисной архитектуры.....	44
1.6. Язык шаблонов микросервисной архитектуры	46
1.6.1. Микросервисная архитектура не панацея	47
1.6.2. Шаблоны проектирования и языки шаблонов	48
1.6.3. Обзор языка шаблонов микросервисной архитектуры.....	51
1.7. Помимо микросервисов: процесс и организация.....	58
1.7.1. Организация разработки и доставки программного обеспечения	59
1.7.2. Процесс разработки и доставки программного обеспечения	60
1.7.3. Человеческий фактор при переходе на микросервисы	61
Резюме	62
Глава 2. Стратегии декомпозиции	63
2.1. Что представляет собой микросервисная архитектура.....	64
2.1.1. Что такое архитектура программного обеспечения и почему она важна.....	64
2.1.2. Обзор архитектурных стилей	67
2.1.3. Микросервисная архитектура как архитектурный стиль	70
2.2. Определение микросервисной архитектуры приложения.....	74
2.2.1. Определение системных операций.....	76
2.2.2. Разбиение на сервисы по бизнес-возможностям	82
2.2.3. Разбиение на сервисы по проблемным областям	85
2.2.4. Методические рекомендации по декомпозиции.....	87
2.2.5. Трудности при разбиении приложения на сервисы	88
2.2.6. Определение API сервисов	92
Резюме	95

Глава 3. Межпроцессное взаимодействие в микросервисной архитектуре	97
3.1. Обзор межпроцессного взаимодействия в микросервисной архитектуре	98
3.1.1. Стили взаимодействия	99
3.1.2. Описание API в микросервисной архитектуре	100
3.1.3. Развивающиеся API	101
3.1.4. Форматы сообщений	103
3.2. Взаимодействие на основе удаленного вызова процедур	105
3.2.1. Использование REST	106
3.2.2. Использование gRPC	109
3.2.3. Работа в условиях частичного отказа с применением шаблона «Предохранитель»	111
3.2.4. Обнаружение сервисов	114
3.3. Взаимодействие с помощью асинхронного обмена сообщениями	119
3.3.1. Обзор механизмов обмена сообщениями	119
3.3.2. Реализация стилей взаимодействия с помощью сообщений	122
3.3.3. Создание спецификации для API сервиса на основе сообщений	124
3.3.4. Использование брокера сообщений	125
3.3.5. Конкурирующие получатели и порядок следования сообщений	129
3.3.6. Дублирование сообщений	130
3.3.7. Транзакционный обмен сообщениями	132
3.3.8. Библиотеки и фреймворки для обмена сообщениями	136
3.4. Использование асинхронного обмена сообщениями для улучшения доступности	139
3.4.1. Синхронное взаимодействие снижает степень доступности	139
3.4.2. Избавление от синхронного взаимодействия	141
Резюме	144
Глава 4. Управление транзакциями с помощью повествований	146
4.1. Управление транзакциями в микросервисной архитектуре	147
4.1.1. Микросервисная архитектура и необходимость в распределенных транзакциях	148
4.1.2. Проблемы с распределенными транзакциями	148
4.1.3. Использование шаблона «Повествование» для сохранения согласованности данных	150
4.2. Координация повествований	154
4.2.1. Повествования, основанные на хореографии	154
4.2.2. Повествования на основе оркестрации	159

4.3. Что делать с недостаточной изолированностью	164
4.3.1. Обзор аномалий.....	165
4.3.2. Контрмеры на случай нехватки изолированности	166
4.4. Архитектура сервиса Order и повествования Create Order	170
4.4.1. Класс OrderService	172
4.4.2. Реализация повествования Create Order	173
4.4.3. Класс OrderCommandHandlers	181
4.4.4. Класс OrderServiceConfiguration	182
Резюме	184
Глава 5. Проектирование бизнес-логики в микросервисной архитектуре.....	185
5.1. Шаблоны организации бизнес-логики	186
5.1.1. Проектирование бизнес-логики с помощью шаблона «Сценарий транзакции»	188
5.1.2. Проектирование бизнес-логики с помощью шаблона «Доменная модель»	189
5.1.3. О предметно-ориентированном проектировании.....	190
5.2. Проектирование доменной модели с помощью шаблона «Агрегат» из DDD....	191
5.2.1. Проблемы с расплывчатыми границами	192
5.2.2. Агрегаты имеют четкие границы.....	194
5.2.3. Правила для агрегатов	195
5.2.4. Размеры агрегатов.....	198
5.2.5. Проектирование бизнес-логики с помощью агрегатов.....	199
5.3. Публикация доменных событий	200
5.3.1. Зачем публиковать события об изменениях	200
5.3.2. Что такое доменное событие	201
5.3.3. Обогащение события	202
5.3.4. Определение доменных событий	202
5.3.5. Генерация и публикация доменных событий.....	204
5.3.6. Потребление доменных событий.....	207
5.4. Бизнес-логика сервиса Kitchen	208
5.4.1. Агрегат Ticket	210
5.5. Бизнес-логика сервиса Order	214
5.5.1. Агрегат Order.....	215
5.5.2. Класс OrderService	220
Резюме	222

Глава 6. Разработка бизнес-логики с порождением событий	223
6.1. Разработка бизнес-логики с использованием порождения событий	224
6.1.1. Проблемы традиционного сохранения данных	225
6.1.2. Обзор порождения событий	227
6.1.3. Обработка конкурентных обновлений с помощью оптимистичного блокирования	234
6.1.4. Порождение и публикация событий	235
6.1.5. Улучшение производительности с помощью снимков	236
6.1.6. Идемпотентная обработка сообщений	238
6.1.7. Развитие доменных событий	239
6.1.8. Преимущества порождения событий	241
6.1.9. Недостатки порождения событий	242
6.2. Реализация хранилища событий	244
6.2.1. Принцип работы хранилища событий Eventuate Local	245
6.2.2. Клиентский фреймворк Eventuate для Java	248
6.3. Совместное использование повествований и порождения событий	252
6.3.1. Реализация повествований на основе хореографии с помощью порождения событий	253
6.3.2. Создание повествования на основе оркестрации	254
6.3.3. Реализация участника повествования на основе порождения событий	256
6.3.4. Реализация оркестраторов повествований с помощью порождения событий	260
Резюме	262
Глава 7. Реализация запросов в микросервисной архитектуре	264
7.1. Выполнение запросов с помощью объединения API	265
7.1.1. Запрос findOrder()	265
7.1.2. Обзор шаблона «Объединение API»	266
7.1.3. Реализация запроса findOrder() путем объединения API	268
7.1.4. Архитектурные проблемы объединения API	269
7.1.5. Преимущества и недостатки объединения API	272
7.2. Применение шаблона CQRS	273
7.2.1. Потенциальные причины использования CQRS	274
7.2.2. Обзор CQRS	277
7.2.3. Преимущества CQRS	280
7.2.4. Недостатки CQRS	281

7.3. Проектирование CQRS-представлений	282
7.3.1. Выбор хранилища данных для представления	283
7.3.2. Структура модуля доступа к данным	285
7.3.3. Добавление и обновление CQRS-представлений	288
7.4. Реализация CQRS с использованием AWS DynamoDB.....	289
7.4.1. Модуль OrderHistoryEventHandlers	290
7.4.2. Моделирование данных и проектирование запросов с помощью DynamoDB	291
7.4.3. Класс OrderHistoryDaoDynamoDb.....	296
Резюме	299
Глава 8. Шаблоны внешних API.....	301
8.1. Проблемы с проектированием внешних API.....	302
8.1.1. Проблемы проектирования API для мобильного клиента FTGO	303
8.1.2. Проблемы с проектированием API для клиентов другого рода	306
8.2. Шаблон «API-шлюз»	307
8.2.1. Обзор шаблона «API-шлюз».....	308
8.2.2. Преимущества и недостатки API-шлюза	315
8.2.3. Netflix как пример использования API-шлюза	316
8.2.4. Трудности проектирования API-шлюза.....	316
8.3. Реализация API-шлюза	320
8.3.1. Использование готового API-шлюза	320
8.3.2. Разработка собственного API-шлюза.....	322
8.3.3. Реализация API-шлюза с помощью GraphQL.....	329
Резюме	341
Глава 9. Тестирование микросервисов, часть 1	343
9.1. Стратегии тестирования микросервисных архитектур.....	345
9.1.1. Обзор методик тестирования	345
9.1.2. Трудности тестирования микросервисов	352
9.1.3. Процесс развертывания	358
9.2. Написание модульных тестов для сервиса.....	360
9.2.1. Разработка модульных тестов для доменных сущностей	363
9.2.2. Написание модульных тестов для объектов значений	364
9.2.3. Разработка модульных тестов для повествований.....	364
9.2.4. Написание модульных тестов для доменных сервисов	366

9.2.5. Разработка модульных тестов для контроллеров	368
9.2.6. Написание модульных тестов для обработчиков событий и сообщений.....	370
Резюме	371
Глава 10. Тестирование микросервисов, часть 2	373
10.1. Написание интеграционных тестов.....	374
10.1.1. Интеграционные тесты с сохранением	376
10.1.2. Интеграционное тестирование взаимодействия в стиле «запрос/ответ» на основе REST	378
10.1.3. Интеграционное тестирование взаимодействия в стиле «издатель/подписчик»	382
10.1.4. Интеграционные тесты контрактов для взаимодействия на основе асинхронных запросов/ответов.....	386
10.2. Разработка компонентных тестов	391
10.2.1. Определение приемочных тестов.....	392
10.2.2. Написание приемочных тестов с помощью Gherkin	392
10.2.3. Проектирование компонентных тестов.....	395
10.2.4. Написание компонентных тестов для сервиса Order.....	396
10.3. Написание сквозных тестов.....	401
10.3.1. Проектирование сквозных тестов	402
10.3.2. Написание сквозных тестов	402
10.3.3. Выполнение сквозных тестов	403
Резюме	403
Глава 11. Разработка сервисов, готовых к промышленному использованию.....	405
11.1. Разработка безопасных сервисов	406
11.1.1. Обзор безопасности в традиционном монолитном приложении	407
11.1.2. Обеспечение безопасности в микросервисной архитектуре	411
11.2. Проектирование конфигурируемых сервисов	420
11.2.1. Вынесение конфигурации вовне с помощью пассивной модели	421
11.2.2. Вынесение конфигурации вовне с помощью активной модели	423
11.3. Проектирование наблюдаемых сервисов	424
11.3.1. Использование API проверки работоспособности.....	426
11.3.2. Применение шаблона агрегации журналов	428
11.3.3. Использование шаблона распределенной трассировки	430
11.3.4. Применение шаблона «Показатели приложения»	434

11.3.5. Шаблон отслеживания исключений	437
11.3.6. Применение шаблона «Ведение журнала аудита»	439
11.4. Разработка сервисов с помощью шаблона микросервисного шасси	440
11.4.1. Использование шасси микросервисов	441
11.4.2. От микросервисного шасси до сети сервисов	442
Резюме	444
Глава 12. Развертывание микросервисов	446
12.1. Развертывание сервисов с помощью пакетов для отдельных языков	449
12.1.1. Преимущества использования пакетов для конкретных языков	452
12.1.2. Недостатки применения пакетов для конкретных языков	452
12.2. Развертывание сервисов в виде виртуальных машин	454
12.2.1. Преимущества развертывания сервисов в виде VM	456
12.2.2. Недостатки развертывания сервисов в виде VM	457
12.3. Развертывание сервисов в виде контейнеров	458
12.3.1. Развертывание сервисов с помощью Docker	460
12.3.2. Преимущества развертывания сервисов в виде контейнеров	463
12.3.3. Недостатки развертывания сервисов в виде контейнеров	463
12.4. Развертывание приложения FTGO с помощью Kubernetes	463
12.4.1. Обзор Kubernetes	464
12.4.2. Развертывание сервиса Restaurant в Kubernetes	467
12.4.3. Развертывание API-шлюза	470
12.4.4. Развертывание без простоя	471
12.4.5. Использование сети сервисов для отделения развертывания от выпуска	472
12.5. Бессерверное развертывание сервисов	481
12.5.1. Обзор бессерверного развертывания с помощью AWS Lambda	482
12.5.2. Написание лямбда-функции	483
12.5.3. Вызов лямбда-функций	484
12.5.4. Преимущества использования лямбда-функций	485
12.5.5. Недостатки использования лямбда-функций	485
12.6. Развертывание RESTful-сервиса с помощью AWS Lambda и AWS Gateway	486
12.6.1. Архитектура сервиса Restaurant на основе AWS Lambda	487
12.6.2. Упаковывание сервиса в виде ZIP-файла	491
12.6.3. Развертывание лямбда-функций с помощью бессерверного фреймворка	492
Резюме	493

Глава 13. Процесс перехода на микросервисы	495
13.1. Переход на микросервисы	496
13.1.1. Зачем переходить с монолита на что-то другое	496
13.1.2. «Удушение» монолита	497
13.2. Стратегии перехода с монолита на микросервисы	501
13.2.1. Реализация новых возможностей в виде сервисов	501
13.2.2. Разделение уровня представления и внутренних компонентов	503
13.2.3. Извлечение бизнес-возможностей в сервисы	505
13.3. Проектирование взаимодействия между сервисом и монолитом	512
13.3.1. Проектирование интеграционного слоя	513
13.3.2. Обеспечение согласованности данных между сервисом и монолитом	518
13.3.3. Аутентификация и авторизация	523
13.4. Реализация новой возможности в виде сервиса	525
13.4.1. Архитектура сервиса Delayed Delivery	526
13.4.2. Проектирование интеграционного слоя для сервиса Delayed Order	528
13.5. Разбиение монолита на части: извлечение управления доставкой	530
13.5.1. Обзор возможностей существующего механизма управления доставкой	530
13.5.2. Обзор сервиса Delivery	532
13.5.3. Проектирование доменной модели сервиса Delivery	533
13.5.4. Структура интеграционного слоя для сервиса Delivery	536
13.5.5. Изменение монолита для взаимодействия с сервисом Delivery	538
Резюме	541

Столкнувшись с неправдой, неравенством или несправедливостью, не молчите, ведь это ваша страна. Это ваша демократия. Будьте ее творцом, защитником и носителем.

*Тэргуд Маршалл (Thurgood Marshall), судья
Верховного суда США*

Предисловие

Будущее уже здесь — оно просто не очень равномерно распределено.

*Уильям Гибсон (William Gibson),
писатель-фантаст*

Это одна из моих любимых цитат. Ее суть в том, что новые идеи и технологии становятся общепринятыми и распространяются в обществе далеко не сразу. Хорошим примером этого может служить мой опыт знакомства с микросервисами. Все началось в 2006 году, когда, вдохновившись выступлением AWS-евангелиста, я вступил на путь, который в итоге привел меня к созданию первой версии Cloud Foundry (хотя с текущей версией ее роднит лишь название). Она представляла собой PaaS (Platform-as-a-Service — платформа как услуга) для автоматизации развертывания Java-приложений в EC2. Как и все другие мои промышленные приложения на Java, Cloud Foundry имело монолитную архитектуру, состоящую из единого файла формата WAR (Java Web Application Archive).

Объединение в монолит разнообразных сложных функций, таких как выделение ресурсов, конфигурирование, мониторинг и управление, привело к проблемам, связанным как с разработкой системы, так и с ее использованием. Например, вы не могли изменить пользовательский интерфейс без тестирования и повторного развертывания всего приложения. А поскольку компонент для мониторинга и управления зависел от системы обработки сложных событий, хранившей свое состояние в памяти, мы не могли запустить больше одного экземпляра Cloud Foundry! В этом стыдно признаться, но я лишь разработчик программного обеспечения, и никто из нас не без греха.

Очевидно, что приложение быстро переросло свою монолитную архитектуру, но какие у нас были альтернативы? В то время подходящее решение уже было известно разработчикам и применялось в таких компаниях, как eBay и Amazon. Например, в Amazon миграция с монолита началась примерно в 2002 году. Новая архитектура состояла из набора слабо связанных сервисов. За каждый сервис отвечала команда

«на две пиццы» (в терминологии Amazon) — достаточно компактная для того, чтобы всех ее членов можно было накормить двумя пиццами.

Компания Amazon приняла на вооружение эту архитектуру, чтобы ускорить темпы разработки, активизировать инновации и повысить конкурентоспособность. Результаты оказались впечатляющими: как сообщается, изменения на промышленном уровне в Amazon происходят каждые 11,6 с!

В начале 2010 года, после того как я переключился на другие проекты, будущее программного проектирования наконец стало актуальным и для меня. Именно тогда я прочитал книгу Майкла Т. Фишера (Michael T. Fisher) и Мартина Л. Эббота (Martin L. Abbott) *The Art of Scalability: Scalable Web Architecture, Processes and Organizations for the Modern Enterprise* (AddisonWesley Professional, 2009). Ее ключевой идеей было трехмерное представление модели масштабирования приложения в виде куба. В соответствии с ним масштабирование по оси Y обозначает разбиение приложения на сервисы. Сейчас такой подход кажется довольно очевидным, но в то время у меня как будто открылись глаза. Наконец-то я мог решить проблемы двухгодичной давности, спроектировав Cloud Foundry в виде набора сервисов!

В апреле 2012 года я впервые описал этот метод проектирования в своем докладе под названием «Декомпозиция приложений для улучшения развертываемости и масштабируемости» (www.slideshare.net/chris.e.richardson/decomposing-applications-for-scalability-and-deployability-april-2012). На тот момент у подобной архитектуры не было общепринятого названия. Иногда я называл ее модульной, многоязычной, поскольку сервисы могут быть написаны на разных языках.

Еще одним примером неравномерного распределения будущего может служить тот факт, что термин «микросервис» (ru.wikipedia.org/wiki/Микросервисная_архитектура) использовался на семинаре по архитектуре программного обеспечения еще в 2011 году, хотя я впервые услышал его во время выступления Фреда Джорджа (Fred George) на конференции Oreddev 2013 и мне он пришелся по душе!

В январе 2014 года я создал сайт microservices.io, чтобы описать архитектуру и шаблоны проектирования, с которыми столкнулся. А в марте 2014-го Джеймс Льюис (James Lewis) и Мартин Фаулер (Martin Fowler) опубликовали статью о микросервисах (martinfowler.com/articles/microservices.html), которая популяризировала этот термин и сплотила сообщество вокруг новой концепции.

Идея небольших слабо связанных между собой команд, которые быстро и надежно разрабатывают и доставляют микросервисы, постепенно набирает популярность в сообществе программистов. Важные промышленные бизнес-приложения сегодня обычно являются монолитными и разрабатываются крупными командами. Выпуск новых версий происходит редко и, как правило, создает проблемы для всех, кто вовлечен в этот процесс. Информационные технологии часто не успевают за потребностями бизнеса. Как же перейти на микросервисную архитектуру, учитывая сказанное?

Ответом на этот вопрос должна послужить данная книга. Прочитав ее, вы начнете хорошо понимать архитектуру микросервисов, узнаете о преимуществах и недостатках этого подхода, а также научитесь использовать его в подходящих ситуациях. В книге описываются способы решения бесчисленных проблем проектирования,

с которыми вы будете сталкиваться, включая работу с распределенными данными. Здесь также рассматриваются методы перевода монолитного приложения на микросервисную архитектуру. Но эта книга не манифест микросервисов. Она организована в виде набора шаблонов проектирования. Каждый шаблон — это универсальное решение проблемы, возникающей в определенном контексте. Красота такого подхода состоит в том, что наряду с преимуществами того или иного решения описываются и его недостатки, которые следует учитывать при работе. По своему опыту могу сказать, что объективность при обдумывании решения дает гораздо лучшие результаты, чем ее отсутствие. Надеюсь, вы получите удовольствие от чтения и научитесь успешно разрабатывать микросервисы.

Благодарности

Труд писателя требует уединения, но, чтобы превратить рукопись в готовую книгу, нужно множество людей.

Прежде всего я хочу поблагодарить Эрвина Тухейя (Erin Twohey) и Майкла Стивенса (Michael Stevens) из издательства Manning за то, что они неизменно одобряют идею написания очередной книги. Также хотел бы сказать спасибо редакторам-консультантам Синтии Кейн (Cynthia Kane) и Марине Майклс (Marina Michaels). Синтия помогла мне начать и поработала над несколькими первыми главами. Дальше эстафету приняла Марина, и с ней мы дошли до конца. Я всегда буду ей признателен за скрупулезную и конструктивную критику написанного. Хотел бы выразить благодарность и остальным сотрудникам Manning, которые работали над изданием этой книги.

Спасибо научному редактору Кристиану Меннерику (Christian Mennerich), корректору Энди Майлзу (Andy Miles) и всем внештатным рецензентам: Энди Киршу (Andy Kirsch), Антонио Пессолано (Antonio Pessolano), Арег Мелик-Адамяну (Areg Melik-Adamyan), Кейджу Слагелю (Cage Slagel), Карлосу Куротто (Carlos Curotto), Дропу Хелперу (Dror Helper), Эросу Педрини (Eros Pedrini), Хьюго Крузу (Hugo Cruz), Ирине Романенко (Irina Romanenko), Джесси Росалье (Jesse Rosalia), Джо Джастесену (Joe Justesen), Джону Гатри (John Guthrie), Кирти Шетти (Keerthi Shetty), Мишель Мауро (Michele Mauro), Полу Гребенцу (Paul Grebenc), Петуру Раджу (Pethuru Raj), Потито Колучелли (Potito Coluccelli), Шобхе Айер (Shobha Iyer), Симеону Лейзерзону (Simeon Leyzerzon), Срихари Сридгарану (Srihari Sridharan), Тиму Муру (Tim Moore), Тони Суитсу (Tony Sweets), Тренту Уайтли (Trent Whiteley), Веси Шаддиксу (Wes Shaddix), Уильяму И. Уилеру (William E. Wheeler) и Золтану Хамори (Zoltan Hamori).

Я хотел бы также поблагодарить всех, кто купил МЕАР и оставил свои отзывы на форуме или связался со мной напрямую.

Спасибо организаторам и участникам всех конференций и встреч, на которых я выступал, за шанс представить и проверить мои идеи. И спасибо моим клиентам по всему миру, для которых я проводил консультации и тренинги, за возможность им помочь и претворить мои задумки в жизнь.

Хочу выразить благодарность моим коллегам по Eventuate, Inc., Эндрю, Валентину, Артему и Станиславу, за их вклад в продукт, над которым работает Eventuate, и проекты с открытым исходным кодом.

Наконец, я хотел бы сказать спасибо жене Лауре и детям Элли, Томасу и Джанет за поддержку и понимание на протяжении последних 18 месяцев. Будучи прикованным к своему ноутбуку, я не мог ходить на футбольные матчи Элли, наблюдать за тем, как Томас учится летать на авиасимуляторе, и посещать новые рестораны вместе с Джанет.

Спасибо вам всем!

О книге

Цель этой книги — научить вас успешно разрабатывать приложения с использованием микросервисной архитектуры.

Здесь обсуждаются не только преимущества, но и недостатки микросервисов. Вы узнаете, в каких ситуациях имеет смысл применять их, а когда лучше подумать о монолитном подходе.

Кому следует прочесть эту книгу

Основное внимание в книге уделяется архитектуре и разработке. Она рассчитана на любого, в чьи обязанности входят написание и доставка программного обеспечения, в том числе на разработчиков, архитекторов, технических директоров и начальников отделов по разработке.

Акцент здесь делается на шаблонах микросервисной архитектуры и других концепциях. Я старался сделать материал доступным вне зависимости от того, какой стек технологий вы используете. Вам лишь нужно познакомиться с основами архитектуры и проектирования приложений уровня предприятия. В частности, понадобятся понимание таких концепций, как трехуровневая архитектура, проектирование веб-приложений, реляционные базы данных, межпроцессное взаимодействие на основе обмена сообщениями и REST, а также базовые знания программной безопасности. Код примеров создан на языке Java. Чтобы извлечь из примеров максимальную пользу, вы должны быть знакомы с фреймворком Spring.

Структура издания

Книга состоит из 13 глав.

- ❑ Глава 1 описывает симптомы монолитного ада, которые проявляются, когда монолитное приложение перерастает свою архитектуру, и дает советы относительно того,

как выйти из этой ситуации путем перехода на микросервисную архитектуру. В ней также предоставлен краткий обзор терминов, которые используются в шаблонах проектирования микросервисов и упоминаются на протяжении почти всей книги.

- ❑ Глава 2 объясняет, почему программная архитектура имеет большое значение, и описывает шаблоны, с помощью которых приложение можно разбить на отдельные сервисы. Кроме того, в ней показаны способы преодоления проблем, с которыми разработчики обычно сталкиваются на этом пути.
- ❑ В главе 3 описываются разные шаблоны для организации надежного межпроцессного взаимодействия в рамках микросервисной архитектуры. Из нее вы узнаете, почему асинхронное взаимодействие, основанное на обмене сообщениями, часто наилучший выбор.
- ❑ В главе 4 показано, как поддерживать согласованность данных между сервисами с помощью шаблона «Повествование» (Saga). Повествование (иногда встречается термин «сага») — это последовательность локальных транзакций, которые координируются с помощью асинхронного обмена сообщениями.
- ❑ Глава 5 описывает процесс построения бизнес-логики сервиса с использованием предметно-ориентированного проектирования, шаблонов агрегирования и доменных событий.
- ❑ Глава 6 является логическим продолжением главы 5. В ней рассказывается, как разработать бизнес-логику с помощью шаблона порождения событий, который представляет собой метод структурирования бизнес-логики и постоянного хранения доменных объектов.
- ❑ Глава 7 посвящена реализации запросов для извлечения данных, разбросанных по разным сервисам. Мы рассмотрим два разных шаблона: объединение API и CQRS (command query responsibility segregation).
- ❑ Глава 8 охватывает шаблоны проектирования внешних API для обработки запросов от разного рода клиентов, таких как мобильные приложения, браузерные JavaScript-приложения и сторонние программы.
- ❑ Глава 9 — первая из двух глав, посвященных методикам автоматического тестирования микросервисов. В ней вы познакомитесь с такими важными понятиями, как пирамида тестирования, описывающая относительные пропорции каждого из типов тестов в вашем тестовом наборе. Вы также научитесь писать модульные тесты, которые станут фундаментом пирамиды тестирования.
- ❑ Глава 10 — логическое продолжение главы 9. Она посвящена написанию тестов других типов, входящих в пирамиду тестирования, включая интеграционные и компонентные тесты, а также проверку потребительских контрактов.
- ❑ В главе 11 освещаются различные аспекты разработки сервисов промышленного уровня, включая безопасность, шаблон вынесения конфигурации вовне и шаблоны наблюдаемости сервисов. В число последних входят агрегация журналов, метрики приложения и распределенная трассировка.

- ❑ В главе 12 описываются различные методы развертывания сервисов, включая виртуальные машины, контейнеры и бессерверные платформы. В ней также раскрываются преимущества использования сети сервисов — слоя сетевого программного обеспечения, который служит посредником при взаимодействии в рамках микросервисной архитектуры.
- ❑ В главе 13 объясняется, как шаг за шагом превратить монолитную архитектуру в микросервисную путем применения шаблона «Подавляющее приложение» (Strangler application). Это подразумевает реализацию новых возможностей в виде сервисов и извлечение модулей из монолитной системы с их последующим преобразованием в сервисы.

По мере чтения вы познакомитесь с разными аспектами микросервисной архитектуры.

О коде

Эта книга содержит множество примеров исходного кода в виде пронумерованных листингов и небольших фрагментов-вставок. В обоих случаях для форматирования используется такой моноширинный шрифт, позволяющий отделить код от остального текста. Во многих случаях оригинальный исходный код был переформатирован — издатель добавил переносы строк и убрал отступы, чтобы оптимально использовать место на странице. Кроме того, из большей части листингов были удалены комментарии, если код описывается в тексте. Нередко код содержит пояснения, которые выделяют важные концепции.

Все главы, кроме 1, 2 и 13, содержат код из сопутствующей демонстрационной программы, которая находится в репозитории GitHub: github.com/microservices-patterns/ftgo-application.

Онлайн-ресурсы

Еще одним отличным ресурсом для изучения микросервисной архитектуры является мой сайт microservices.io. Он содержит не только полный набор шаблонов проектирования, но и ссылки на другие ресурсы, включая статьи, презентации и примеры кода.

Об авторе

Крис Ричардсон (Chris Richardson) — разработчик, архитектор и автор книги *POJOs in Action* (Manning, 2006), в которой описывается процесс построения Java-приложений уровня предприятия с помощью фреймворков Spring и Hibernate. Он носит почетные звания Java Champion и JavaOne Rock Star.

Крис разработал оригинальную версию CloudFoundry.com — раннюю реализацию платформы Java PaaS для Amazon EC2.

Ныне он считается признанным идейным лидером в мире микросервисов и регулярно выступает на международных конференциях. Крис создал сайт microservices.io, на котором собраны шаблоны проектирования микросервисов. А еще он проводит по всему миру консультации и тренинги для организаций, которые переходят на микросервисную архитектуру. Сейчас Крис работает над своим третьим стартапом Eventuate.io. Это программная платформа для разработки транзакционных микросервисов.

Об иллюстрации на обложке

Рисунок на обложке называется «Одевание раба-мориска в 1568 году»¹ и позаимствован из четырехтомника Томаса Джеффериса (Thomas Jefferys) *A Collection of the Dresses of Different Nations, Ancient and Modern* («Коллекция платья разных народов, старинного и современного»), изданного в Лондоне в 1757–1772 годах. На титульном листе говорится, что это гравюра, раскрашенная вручную с применением гуммиарабика.

Томаса Джеффериса (1719–1771) называли географом при короле Георге III. Он был английским картографом и ведущим поставщиком карт своего времени. Он чертил и печатал карты для правительства и других государственных органов. На его счету целый ряд коммерческих карт и атласов, в основном Северной Америки. Занимаясь картографией в разных уголках мира, он интересовался местными традиционными нарядами, которые впоследствии были блестяще переданы в данной коллекции. В конце XVIII века заморские путешествия для собственного удовольствия были относительно новым явлением, поэтому подобные коллекции пользовались популярностью, позволяя получить представление о жителях других стран как настоящим туристам, так и «диванным» путешественникам.

Разнообразие иллюстраций в книгах Джеффериса — яркое свидетельство уникальности и оригинальности народов мира в то время. С тех пор тенденции в одежде сильно изменились, а региональные и национальные различия, такие значимые 200 лет назад, постепенно сошли на нет. В наши дни бывает сложно отличить друг от друга жителей разных континентов. Если взглянуть на это с оптимистичной точки зрения, мы пожертвовали культурным и внешним разнообразием в угоду более насыщенной личной жизни или в угоду более разнообразной и интересной интеллектуальной и технической деятельности.

В то время как большинство компьютерных изданий мало чем отличаются друг от друга, компания Manning выбирает для своих книг обложки, основанные на богатом региональном разнообразии, которое Джефферис воплотил в иллюстрациях два столетия назад. Это ода находчивости и инициативности современной компьютерной индустрии.

¹ Habit of a Morisco Slave in 1568.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу **comp@piter.com** (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства **www.piter.com** вы найдете подробную информацию о наших книгах.

Побег из монолитного ада

В этой главе

- Симптомы монолитного ада и их устранение путем перехода на микросервисную архитектуру.
- Неотъемлемые характеристики микросервисной архитектуры, ее достоинства и недостатки.
- Каким образом микросервисы позволяют разрабатывать крупные сложные приложения в стиле DevOps.
- Язык шаблонов проектирования микросервисов, и почему вы должны его использовать.

Понедельник превосходно начался для Мэри, технического директора компании Food to Go, Inc. (FTGO), но уже к обеденному перерыву она почувствовала разочарование. Мэри провела предыдущую неделю вместе с другими архитекторами и разработчиками программного обеспечения на отличной конференции, знакомясь с последними веяниями в мире программирования, включая непрерывное развертывание и микросервисную архитектуру. Она встретилась с сокурсниками по университету Северной Каролины и поделилась с ними историями о борьбе за технологическое лидерство. Конференция укрепила ее в желании улучшить процесс разработки в FTGO.

К сожалению, это чувство быстро испарилось. Она только что потратила первое утро новой недели на неприятное совещание с бизнес-руководством и старшими разработчиками. На протяжении двух часов они обсуждали очередную задержку

в выпуске критически важной версии продукта. Увы, но последние несколько лет подобного рода совещания проходили все чаще и чаще. Несмотря на использование гибких методик, темп разработки продолжил замедляться, делая бизнес-требования почти невыполнимыми. И что хуже всего, простого решения, по всей видимости, не было.

Благодаря конференции Мэри осознала, что ее компания столкнулась с проблемой *монолитного ада* и что панацеей станет переход на микросервисную архитектуру. Но такой подход и сопутствующие ему новейшие методики разработки казались недостижимой мечтой. Мэри сложно было представить, как она будет исправлять текущую ситуацию и одновременно улучшать процесс разработки в FTGO.

К счастью, как вы вскоре сможете убедиться, это можно сделать. Но сначала рассмотрим проблемы, с которыми сталкивается компания FTGO, и попытаемся понять, что их вызвало.

1.1. Медленным шагом в монолитный ад

С момента основания в 2005 году компания FTGO демонстрировала стремительный рост. Сейчас она входит в число лидеров на американском рынке доставки еды. У руководства даже имеются планы расширения за рубеж, но они находятся под угрозой срыва из-за задержек в реализации необходимых возможностей.

По своей сути приложение FTGO довольно простое. Клиенты заказывают еду в местных ресторанах на сайте компании или с помощью мобильного приложения. FTGO координирует сеть курьеров, которые доставляют заказы. Компания также отвечает за оплату услуг курьеров и ресторанов. Последние с помощью веб-сайта FTGO редактируют меню и управляют заказами. Приложение использует различные веб-сервисы: Stripe для платежей, Twilio для обмена сообщениями и Amazon Simple Email Service (SES) для электронной почты.

Как и многие другие устаревающие промышленные приложения, FTGO представляет собой монолит, состоящий из единого файла в формате Java WAR (Web Application Archive). С годами это приложение стало большим и сложным. Несмотря на все усилия команды разработчиков, оно превратилось в живую иллюстрацию антишаблона под названием «*большой комок грязи*» (Big Ball of Mud; www.laputan.org/mud/, https://ru.wikipedia.org/wiki/Большой_комок_грязи). Цитируя Фута и Йодера, авторов этого антишаблона, это «беспорядочно структурированные, растянутые, неряшливые, словно перемотанные на скорую руку изоляционной лентой и проводами, джунгли спагетти-кода». Темпы выпуска новых версий замедлились. Но что еще хуже, приложение FTGO было написано с использованием устаревших фреймворков. Налицо все симптомы монолитного ада.

В следующем подразделе описывается архитектура приложения FTGO. В нем также объясняется, почему монолитная архитектура хорошо себя проявляла в самом начале. Мы поговорим о том, как приложение FTGO переросло свою архитектуру и каким образом это привело к монолитному аду.

1.1.1. Архитектура приложения FTGO

FTGO — это типичное Java-приложение уровня предприятия. Его архитектура, выполнена в виде шестиугольника (рис. 1.1). Мы подробнее обсудим этот архитектурный стиль в главе 2. В гексагональной архитектуре ядром приложения является бизнес-логика, которую окружают различные адаптеры, реализующие пользовательский интерфейс и выполняющие интеграцию с внешними системами.

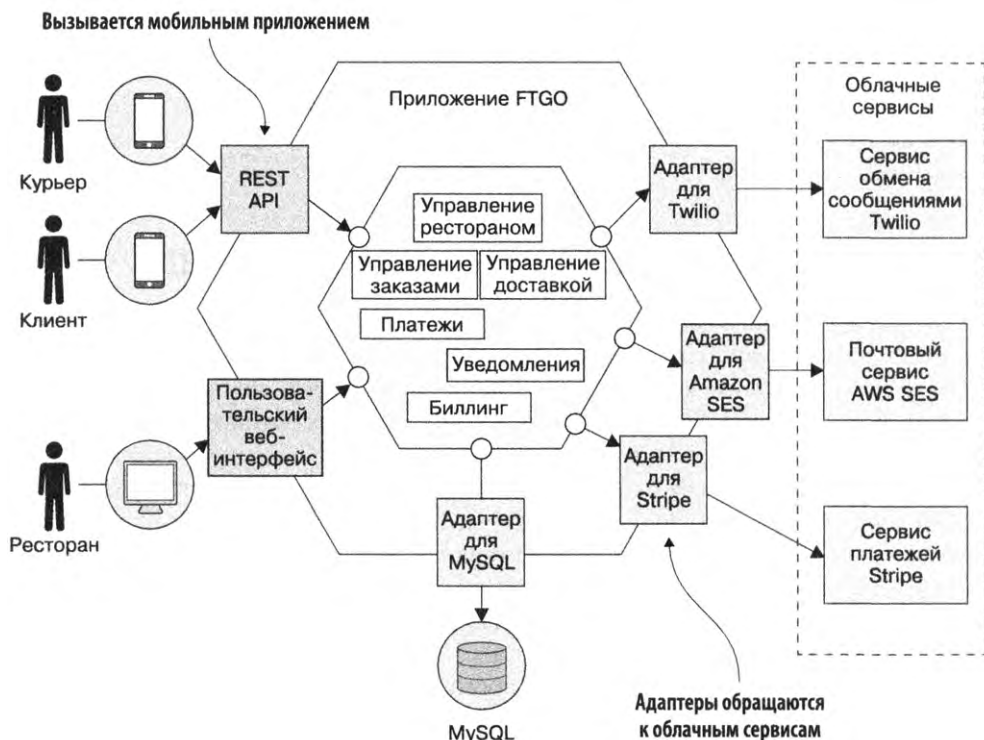


Рис. 1.1. Приложение FTGO имеет гексагональную архитектуру. Оно состоит из бизнес-логики, окруженной адаптерами, которые реализуют пользовательские интерфейсы и взаимодействуют с внешними системами, например мобильными клиентами и облачными сервисами для платежей, электронной почты и обмена сообщениями

Бизнес-логика состоит из модулей, каждый из которых представляет собой набор доменных объектов. В качестве примеров модулей можно привести *управление заказами*, *управление доставкой*, *биллинг* и *платежи*. Здесь также есть несколько адаптеров, взаимодействующих с внешними системами. Некоторые адаптеры направлены *вовнутрь* и обслуживают запросы путем обращения к бизнес-логике — это относится к *REST API* и *пользовательскому веб-интерфейсу*. Остальные адаптеры направлены *вовне*, позволяя бизнес-логике получать доступ к MySQL и работать с такими облачными сервисами, как Twilio и Stripe.

Несмотря на то что приложение FTGO имеет логически модульную структуру, оно упаковано в единый WAR-файл. Это пример широко распространенного *монолитного* стиля программной архитектуры, в соответствии с которым система структурируется в виде одного исполняемого файла или развертываемого компонента. Если бы приложение FTGO было написано на языке Go (Golang), это был бы один исполняемый файл. В случае с Ruby или NodeJS это было бы единое дерево каталогов с исходным кодом. В монолитной архитектуре как таковой нет ничего плохого. Разработчики FTGO приняли верное решение, выбрав ее для своего приложения.

1.1.2. Преимущества монолитной архитектуры

На ранних этапах развития приложение FTGO было относительно небольшим, поэтому монолитная архитектура давала ему множество преимуществ.

- ❑ *Простота разработки* — IDE и другие инструменты разработки сосредоточены на построении единого приложения.
- ❑ *Легкость внесения радикальных изменений* — вы можете поменять код и структуру базы данных, а затем собрать и развернуть полученный результат.
- ❑ *Простота тестирования* — разработчики написали сквозные тесты, которые запускали приложение, обращались к REST API и проверяли пользовательский интерфейс с помощью Selenium.
- ❑ *Простота развертывания* — разработчику достаточно было скопировать WAR-файл на сервер с установленной копией Tomcat.
- ❑ *Легкость масштабирования* — компания FTGO запускала несколько экземпляров приложения, размещенных за балансировщиком нагрузки.

Однако со временем разработка, тестирование и масштабирование существенно усложнились. Посмотрим почему.

1.1.3. Жизнь в монолитном аду

К своему сожалению, разработчики FTGO обнаружили, что у такого подхода есть огромный недостаток. Успешные приложения, такие как FTGO, имеют склонность вырастать из монолитной архитектуры. Каждый раз команда разработчиков FTGO реализовывала несколько новых возможностей, увеличивая тем самым кодовую базу. Более того, успех компании постепенно приводил к росту числа программистов. А это не только ускорило темпы разрастания кодовой базы, но и повысило накладные расходы на администрирование.

Как видно на рис. 1.2, с годами небольшое простое приложение FTGO превратилось в чудовищный монолит. Точно так же некогда компактная команда разработчиков теперь состоит из нескольких `scrum`-команд, каждая из которых работает над конкретной функциональной областью. Переросшее свою архитектуру приложение FTGO попало в монолитный ад. Разработка стала медленной и мучительной.

Разрабатывать и развертывать код с применением гибких методов больше невозможно. Посмотрим, почему так получилось.

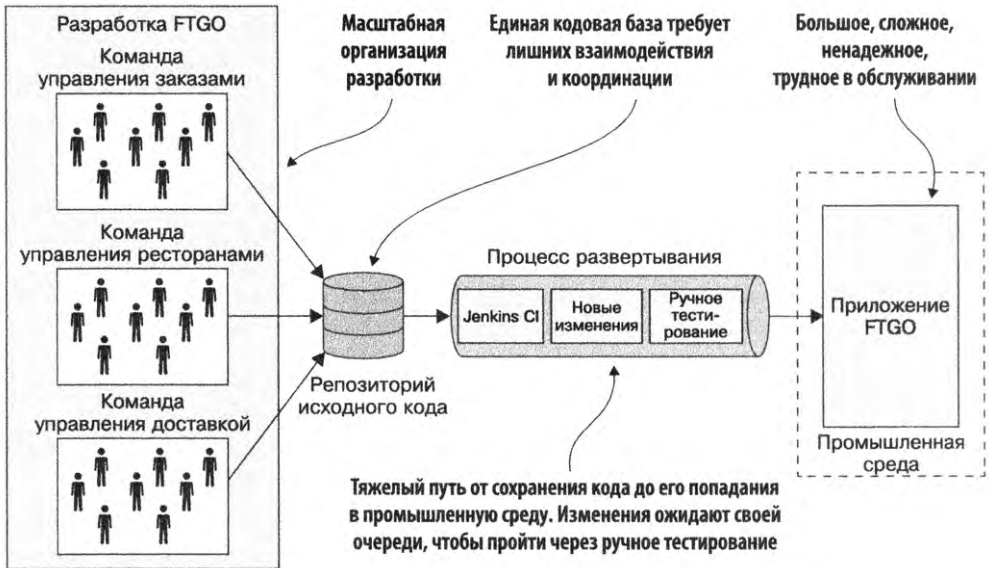


Рис. 1.2. Пример монолитного ада. Большая команда разработчиков FTGO сохраняет свои изменения в едином репозитории исходного кода. Между сохранением и попаданием кода в промышленную среду лежит длинный и тяжелый путь, который подразумевает ручное тестирование. Приложение FTGO стало большим, сложным, ненадежным и трудным в обслуживании

Высокая сложность пугает разработчиков

Основная проблема приложения FTGO заключается в его чрезмерной сложности. Оно слишком большое для того, чтобы один разработчик мог его понять. В итоге исправление ошибок и реализация новых возможностей усложнились и стали занимать много времени. Разработчики не успевали в срок.

Усугубляет проблему то, что сложность, и так чрезмерная, обычно повышается экспоненциально. Если кодовая база плохо поддается пониманию, разработчик не сможет внести изменения подходящим образом. Каждое изменение усложняет код и делает его еще менее понятным. Элегантная модульная архитектура, показанная на рис. 1.1, не отражает действительности. Приложение FTGO постепенно превращается в большой комок грязи.

Мэри вспомнила, что на конференции, где недавно присутствовала, она встретила разработчика, который писал инструмент для анализа зависимостей между тысячами JAR-файлов в приложении, состоящем из миллиона строчек кода. Тогда ей показалось, что этот инструмент можно будет применить для FTGO. Теперь она в этом не уверена. Что-то подсказывает Мэри, что лучшим решением будет переход на архитектуру, более приспособленную к крупным приложениям, — микросервисы.

Медленная разработка

Помимо борьбы с чрезмерной сложностью, разработчикам FTGO приходится иметь дело с замедлением ежедневных технических задач. Большое приложение перегружает и замедляет их IDE. Сборка кода занимает много времени. Более того, из-за своей величины приложение долго запускается. В итоге затягивается цикл написания — сборки — запуска — тестирования кода, что плохо сказывается на продуктивности.

Длинный и тяжелый путь от сохранения изменений до их развертывания

Еще одна проблема с приложением FTGO состоит в том, что доставка изменений в промышленную среду является долгим и тяжелым процессом. Команда разработчиков обычно развертывает обновления раз в месяц, как правило в ночь пятницы или субботы. Мэри постоянно читает о том, что новейшим подходом к обслуживанию приложений типа SaaS (Software-as-a-Service) является *непрерывное развертывание*: доставка изменений в промышленную среду многократно в течение суток и в рабочее время. Как сообщается, по состоянию на 2011 год компания Amazon.com развертывала изменения каждые 11,6 с, никак не затрагивая при этом конечного пользователя! Для разработчиков FTGO обновление продукта чаще чем раз в месяц кажется несбыточной мечтой. А использование непрерывного развертывания выглядит чем-то нереальным.

Компания FTGO частично применяет гибкую методологию разработки. Команда разработчиков делится на группы и выполняет двухнедельные «спурты». К сожалению, путь готового кода к промышленной среде оказывается длинным и тяжелым. Работа такого большого количества программистов над одной и той же кодовой базой часто приводит к тому, что сборку нельзя выпустить. Разработчики FTGO пытались решить эту проблему, создавая отдельные ветви для новых возможностей, но это вылилось в затяжные и мучительные слияния кода. Как следствие, по окончании «спурта» следует длительный период тестирования и стабилизации кодовой базы.

Еще одна причина того, почему изменения так долго доходят до промышленной среды, связана с длительным тестированием. Код настолько сложен, а эффект от внесенного изменения так неочевиден, что разработчикам и серверу непрерывной интеграции (Continuous Integration, CI) приходится выполнять весь набор тестов. Некоторые участки системы даже требуют ручного тестирования. Кроме того, значительное время затрачивается на диагностику и исправление причин проваленных тестов. В итоге на завершение цикла тестирования требуется несколько дней.

Трудности с масштабированием

Команда FTGO испытывает проблемы и с масштабированием своего приложения. Дело в том, что требования к ресурсам разных программных модулей конфликтуют между собой. Например, данные о ресторанах хранятся в большой резидентной базе, которую желательно развертывать на серверах с большим объемом оперативной

памяти. Для сравнения: модуль обработки изображений сильно нагружает ЦПУ и в идеале должен работать на серверах с большими вычислительными ресурсами. Но, поскольку эти модули входят в одно и то же приложение, компании приходится идти на компромисс при выборе серверной конфигурации.

Сложно добиться надежности монолитного приложения

Еще одна проблема с приложением FTGO заключается в его недостаточной надежности. В результате в работе промышленной среды часто возникают перебои. Одна из причин — то, что из-за большого размера приложения его сложно как следует протестировать. Недостаточное тестирование означает, что ошибки попадают в итоговую версию программы. Что еще хуже, приложению не хватает *локализации неисправностей*, поскольку все модули выполняются внутри одного процесса. Время от времени ошибка в одном модуле (например, утечка памяти) приводит к поочередному сбою всех экземпляров системы. Разработчикам FTGO не очень нравится, когда их вызывают посреди ночи из-за поломки в промышленной среде. Недовольно и руководство, ведь при этом страдают доходы и доверие к компании.

Зависимость от постепенно устаревающего стека технологий

И последнее, с чем столкнулась команда FTGO, — то, что архитектура, обусловившая монолитный ад, заставляет использовать постепенно устаревающий стек технологий. При этом разработчикам сложно переходить на новые фреймворки и языки программирования. Переписать все монолитное приложение, применив новые и, предположительно, лучшие технологии, было бы чрезвычайно дорого и рискованно. Как следствие, программистам приходится работать с теми инструментами, которые они выбрали при запуске проекта. Из-за этого они часто обязаны поддерживать код, написанный с помощью устаревших средств.

Фреймворк Spring продолжает развиваться, поддерживая обратную совместимость, поэтому теоретически у команды FTGO должна быть возможность перехода на новые версии. К сожалению, в приложении используются версии фреймворков, несовместимые с новейшими выпусками Spring. Разработчики так и не нашли времени для их обновления. В итоге существенная часть кода написана с применением устаревших технологий. Кроме того, программистам хотелось бы поэкспериментировать с языками, не основанными на JVM, такими как GoLang или NodeJS. К сожалению, монолитная архитектура исключает такую возможность.

1.2. Почему эта книга актуальна для вас

Вполне вероятно, что вы разработчик, архитектор, технический директор или начальник отдела разработки. И отвечаете за приложение, которое переросло монолитную архитектуру. Как и Мэри из FTGO, вы испытываете сложности с доставкой программного обеспечения и хотите узнать, как уйти из монолитного ада. Или, возможно, боитесь, что ваша организация уже на пути к монолитному аду, и хотите

как-то изменить курс, пока еще не слишком поздно. Если вы хотите избежать такой ситуации или выбраться из нее, эта книга для вас.

Большое внимание здесь уделяется концепциям микросервисной архитектуры. Я стремился сделать материал доступным вне зависимости от того, какой стек технологий вы используете. Но нужно, чтобы вы были знакомы с основами архитектуры и проектирования приложений уровня предприятия, в частности знали:

- ☐ трехуровневую архитектуру;
- ☐ проектирование веб-приложений;
- ☐ разработку бизнес-логики в объектно-ориентированном стиле;
- ☐ применение реляционных СУБД — SQL- и ACID-транзакции;
- ☐ межпроцессное взаимодействие с использованием брокера сообщений и REST API;
- ☐ безопасность, включая аутентификацию и авторизацию.

Примеры кода в этой книге выполнены на языке Java с применением фреймворка Spring. Поэтому вы должны быть знакомы с этим фреймворком, чтобы извлечь максимальную пользу из представленного кода.

1.3. Чему вы научитесь, прочитав эту книгу

Дойдя до последней страницы, вы усвоите такие темы.

- ☐ Основные характеристики микросервисной архитектуры, ее достоинства и недостатки, а также сценарии, в которых ее следует использовать.
- ☐ Методы работы с распределенными данными.
- ☐ Эффективные стратегии тестирования микросервисов.
- ☐ Варианты развертывания микросервисов.
- ☐ Стратегии перевода монолитного приложения на микросервисную архитектуру.

Вы также получите следующие навыки.

- ☐ Проектирование приложений с применением микросервисной архитектуры.
- ☐ Разработка бизнес-логики для микросервисов.
- ☐ Использование повествований для обеспечения согласованности данных между сервисами.
- ☐ Реализация запросов, охватывающих несколько сервисов.
- ☐ Эффективное тестирование микросервисов.
- ☐ Разработка безопасных, настраиваемых и наблюдаемых сервисов, готовых к промышленному применению.
- ☐ Разбиение существующих монолитных приложений на сервисы.

1.4. Микросервисная архитектура спешит на помощь

Мэри пришла к выводу, что ее компания должна перейти на микросервисную архитектуру.

Что интересно, программная архитектура имеет мало общего с функциональными требованиями. Вы можете реализовать набор *сценариев* (функциональных требований к приложению) с использованием любой архитектуры. На самом деле таким успешным приложениям, как FTGO, свойственно быть большими и монолитными.

Конечно, архитектура тоже важна, ведь она определяет так называемые требования к *качеству обслуживания*, известные также как *нефункциональные требования* или *атрибуты качества*. Рост приложения FTGO сказался на различных его атрибутах качества, особенно на тех, которые влияют на скорость доставки программного обеспечения: обслуживаемости, расширяемости и тестируемости.

С одной стороны, дисциплинированная команда способна замедлить процесс скатывания в монолитный ад. Программисты могут усердно поддерживать модульность своего приложения. А еще — написать комплексные автоматические тесты. С другой стороны, у них не получится избежать проблем, свойственных большим командам, которые работают над одной монолитной кодовой базой. Они также не смогут ничего поделать с постоянно устаревающим стеком технологий. В их власти лишь отсрочить неизбежное. Чтобы убежать из монолитного ада, придется мигрировать на новую, *микросервисную* архитектуру.

Сегодня все больше специалистов сходятся на том, что при построении крупного, сложного приложения следует задуматься об использовании микросервисов. Но что именно мы имеем в виду под *микросервисами*? К сожалению, само название мало о чем говорит, так как основной акцент в нем делается на размере. У микросервисной архитектуры есть бесчисленное количество определений. Одни понимают название слишком буквально и утверждают, что сервис должен быть крошечным, например состоять из 100 строчек кода. Другие считают, что на разработку сервиса должно уходить не более двух недель. Адриан Кокрофт (Adrian Cockcroft), ранее работавший в Netflix, определяет микросервисную архитектуру как сервис-ориентированную, состоящую из слабо связанных элементов с ограниченным контекстом. Это неплохое определение, но ему недостает ясности. Попробуем придумать что-нибудь получше.

1.4.1. Куб масштабирования и микросервисы

Мое определение микросервисной архитектуры навеяно прекрасной книгой Мартина Эбботта (Martin Abbott) и Майкла Фишера (Michael Fisher) *The Art of Scalability* (Addison-Wesley, 2015). В ней описывается практичная трехмерная модель масштабирования в виде *куба* (рис. 1.3).

Эта модель определяет три направления для масштабирования приложений: X, Y и Z.

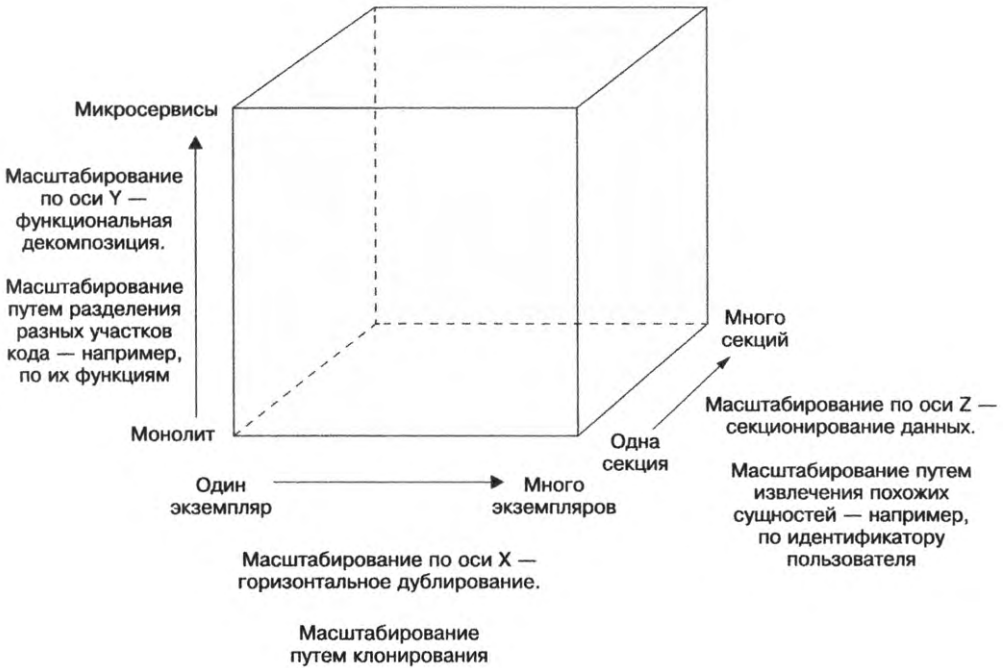


Рис. 1.3. Модель определяет три направления для масштабирования приложения: масштабирование по оси X распределяет нагрузку между несколькими идентичными экземплярами, по оси Z — направляет запросы в зависимости от их атрибутов, ось Y разбивает приложение на сервисы с разными функциями

Масштабирование по оси X распределяет запросы между несколькими экземплярами

Масштабирование по *оси X* часто применяют в монолитных приложениях. Принцип работы этого подхода показан на рис. 1.4. Запускаются несколько экземпляров программы, размещенных за балансировщиком нагрузки. Балансировщик распределяет запросы между N одинаковыми экземплярами. Это отличный способ улучшить мощность и доступность приложения.

Масштабирование по оси Z направляет запросы в зависимости от их атрибутов

Масштабирование по *оси Z* тоже предусматривает запуск нескольких экземпляров монолитного приложения, но в этом случае, в отличие от масштабирования по *оси X*, каждый экземпляр отвечает за определенное подмножество данных (рис. 1.5). Маршрутизатор, выставленный впереди, задействует атрибут запроса, чтобы направить его к подходящему экземпляру. Для этого, к примеру, можно использовать поле `userId`.



Рис. 1.4. Масштабирование по оси X связано с запуском нескольких идентичных экземпляров монолитного приложения, размещенных за балансировщиком нагрузки

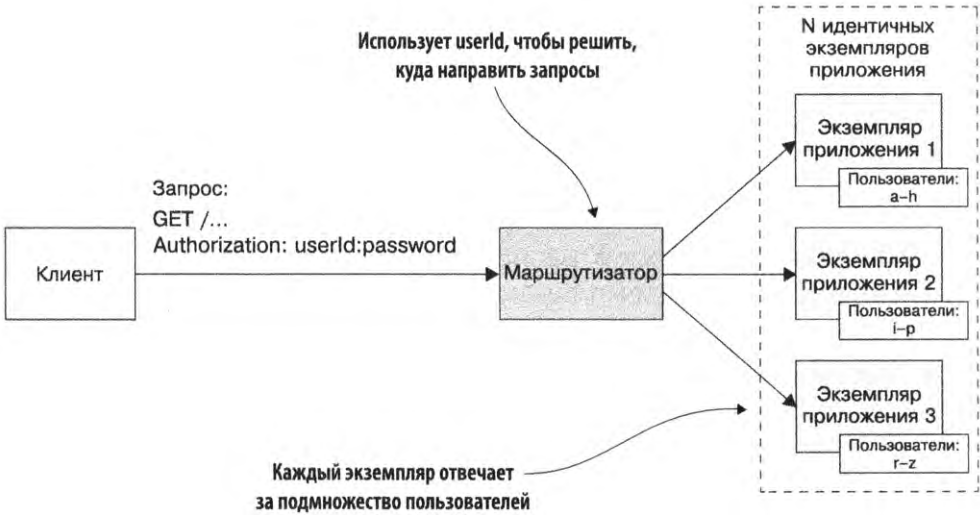


Рис. 1.5. Масштабирование по оси Z связано с запуском нескольких идентичных экземпляров монолитного приложения, размещенных за маршрутизатором, который направляет запросы в зависимости от их атрибутов. Каждый экземпляр отвечает за подмножество данных

В данном сценарии каждый экземпляр приложения отвечает за подмножество пользователей. Маршрутизатор проверяет поле `userId`, указанное в заголовке запроса `Authorization`, чтобы выбрать одну из N идентичных копий программы. Масштабирование по оси Z отлично помогает справиться с участвующими транзакциями и растущими объемами данных.

Масштабирование по оси Y разбивает приложение на сервисы с разными функциями

Масштабирование по осям *X* и *Z* увеличивает мощность и доступность приложения. Но ни один из этих подходов не решает проблем с усложнением кода и процесса разработки. Чтобы справиться с ними, следует применить масштабирование по *оси Y*, или *функциональную декомпозицию* (разбиение). То, как это работает, показано на рис. 1.6: монолитное приложение разбивается на отдельные сервисы.



Рис. 1.6. Масштабирование по оси Y разбивает приложение на отдельные сервисы. Каждый из них отвечает за определенную функцию и масштабируется по оси X (а также, возможно, по оси Z)

Сервис — это мини-приложение, реализующее узкоспециализированные функции, такие как управление заказами, управление клиентами и т. д. Сервисы масштабируются по оси *X*, некоторые из них могут использовать также ось *Z*. Например, сервис Order имеет несколько копий, нагрузка на которые балансируется.

Обобщенное определение микросервисной архитектуры (или микросервисов) звучит так: это стиль проектирования, который разбивает приложение на отдельные сервисы с разными функциями. Заметьте, что размер здесь вообще не упоминается. Главное, чтобы каждый сервис имел четкий перечень связанных между собой обязанностей. Позже мы поговорим о том, что это означает.

Теперь рассмотрим микросервисную архитектуру как разновидность модульности.

1.4.2. Микросервисы как разновидность модульности

Модульность является неотъемлемой частью разработки крупных сложных приложений. Современные приложения, такие как FTGO, слишком велики для разработки в одиночку и слишком сложны для того, чтобы в них мог разобраться отдельный

человек. Приложения следует разбивать на модули, которые разрабатывают и в которых разбираются разные люди. В монолитном проекте модули представляют собой сочетание концепций языка программирования, таких как пакеты в Java, и ресурсов, участвующих в сборке, таких как JAR-файлы. Но, как выяснили разработчики FTGO, этот подход обычно не очень практичен. Монолитные приложения с длинным жизненным циклом, как правило, превращаются в «большие комки грязи».

В микросервисной архитектуре единицей модульности является сервис. Сервисы обладают API, которые служат непроницаемым барьером. В отличие от пакетов в Java API нельзя обойти, чтобы обратиться к внутреннему классу. В долгосрочной перспективе это намного упрощает поддержание модульности приложения. Использование сервисов в качестве строительных блоков имеет и другие преимущества, например, каждый из них можно развертывать и масштабировать отдельно.

1.4.3. У каждого сервиса есть своя база данных

Ключевой особенностью микросервисной архитектуры является то, что сервисы слабо связаны между собой и взаимодействуют только через API. Слабой связанности можно достичь за счет выделения каждому сервису отдельной базы данных. Например, в онлайн-магазине сервисы `Order` и `Customer` могли бы иметь собственные базы данных с таблицами `ORDERS` и `CUSTOMERS` соответственно. Структуру данных сервиса можно менять на этапе разработки, не координируя свои действия с разработчиками других сервисов. На этапе выполнения сервисы изолированы друг от друга — ни одному из них, например, не придется ждать из-за того, что другой сервис заблокировал БД.

Не волнуйтесь, Ларри Эллисон не наживается на слабой связанности

То, что каждый сервис имеет собственную БД, вовсе не означает, что ему выделяется целый сервер баз данных. То есть вам не нужно тратить в десять раз больше на лицензии для Oracle RDBMS. Подробнее об этом — в главе 2.

Разобравшись с тем, что такое микросервисы и каковы их ключевые характеристики, можем посмотреть, как это все относится к приложению FTGO.

1.4.4. Микросервисная архитектура для FTGO

В дальнейшем на страницах книги мы будем подробно обсуждать микросервисную архитектуру приложения FTGO. Но сначала взглянем на то, как в этом контексте выглядит масштабирование по оси Y. Применив к FTGO декомпозицию, мы получим архитектуру (рис. 1.7). Разбитый на части код состоит из многочисленных сервисов, как клиентских (фронтенд), так и серверных (бэкенд). Мы можем также провести масштабирование по осям X и Z, чтобы на этапе выполнения каждый сервис имел несколько экземпляров.

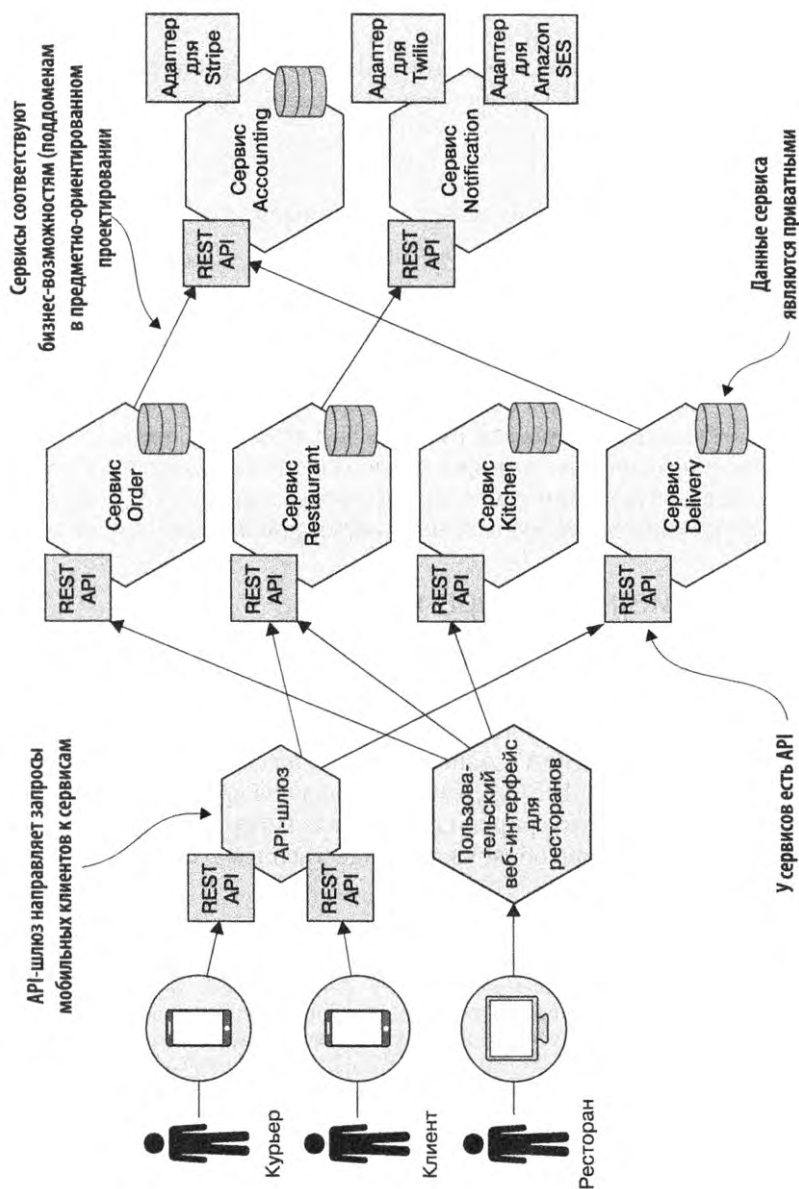


Рис. 1.7. Некоторые из сервисов в микросервисной версии приложения FTGO. API-шлюз направляет запросы мобильных клиентов к сервисам. Сервисы взаимодействуют между собой через API

Клиентские сервисы включают в себя API-шлюз и пользовательский веб-интерфейс для ресторанов. API-шлюз, который, как вы увидите в главе 8, играет роль фасада, предоставляет интерфейсы REST API, применяемые в мобильных приложениях для заказчиков и курьеров. Веб-интерфейс используется ресторанами для управления меню и обработки заказов.

Бизнес-логика приложения FTGO состоит из многочисленных бэкэнд-сервисов. У каждого из них есть REST API и собственная приватная база данных. В их число входят такие сервисы:

- ❑ **Order** — управляет заказами;
- ❑ **Delivery** — управляет доставкой заказов из ресторана клиентам;
- ❑ **Restaurant** — хранит информацию о ресторанах;
- ❑ **Kitchen** — отвечает за подготовку заказов;
- ❑ **Accounting** — управляет биллингом и платежами.

Многие сервисы являются аналогами модулей, описанных ранее в этой главе. Разница в том, что все сервисы и их API имеют четкие определения. Каждый из них можно разрабатывать, тестировать, развертывать и масштабировать независимо от остальных. Такая архитектура помогает поддерживать модульность. Разработчик не может обратиться к внутренним компонентам сервиса, минуя его API. В главе 13 вы узнаете, как преобразовать существующее монолитное приложение в микросервисы.

1.4.5. Сравнение микросервисной и сервис-ориентированной архитектур

Некоторые критики микросервисов утверждают, что в этом подходе нет ничего нового и что это всего лишь разновидность сервис-ориентированной архитектуры (service-oriented architecture, SOA). Действительно, на самом высоком уровне существует некоторое сходство. И SOA, и микросервисная архитектура — это стили проектирования, которые структурируют систему как набор сервисов. Но при более детальном рассмотрении можно обнаружить существенные различия (табл. 1.1).

Таблица 1.1. Сравнение SOA и микросервисов

Параметр	SOA	Микросервисы
Межсервисное взаимодействие	Умные каналы, такие как сервисная шина предприятия, с использованием тяжеловесных протоколов вроде SOAP и других веб-сервисных стандартов	Примитивные каналы, такие как брокер сообщений, или прямое взаимодействие между сервисами с помощью легковесных протоколов наподобие REST или gRPC
Данные	Глобальная модель данных и общие БД	Отдельные модель данных и БД для каждого сервиса
Типовой сервис	Крупное монолитное приложение	Небольшой сервис

SOA и микросервисная архитектура обычно используют разные стеки технологий. В приложениях на основе SOA, как правило, применяются тяжеловесные стан-

дарты веб-сервисов наподобие SOAP. Им часто нужна сервисная шина предприятия (Enterprise Service Bus, ESB) — *умный канал*, который интегрирует сервисы с помощью бизнес-логики и кода для обработки сообщений. Приложения, спроектированные в виде микросервисов, обычно задействуют легковесные технологии с открытым исходным кодом. Сервисы взаимодействуют через *примитивные каналы*, такие как брокеры сообщений или простые протоколы, подобные REST или gRPC.

SOA и микросервисная архитектура также по-разному обращаются с данными. SOA-приложения обычно имеют глобальную модель данных и общую БД. Для сравнения: как уже упоминалось, у каждого микросервиса есть собственная база данных. Более того, в главе 2 вы увидите, что каждому сервису, как правило, отводится отдельная доменная модель.

Еще одно важное различие между двумя архитектурами заключается в размере сервисов. SOA обычно используется для интеграции крупных, сложных, монолитных приложений. В микросервисной архитектуре сервисы не всегда являются крошечными, но почти всегда они оказываются намного меньше. Так что большинство SOA-приложений состоит из нескольких больших частей, а микросервисные проекты чаще всего разбиты на десятки или сотни мелких сервисов.

1.5. Достоинства и недостатки микросервисной архитектуры

Начнем с достоинств, а затем рассмотрим недостатки.

1.5.1. Достоинства микросервисной архитектуры

Микросервисная архитектура имеет следующие преимущества.

- ❑ Она делает возможными непрерывные доставку и развертывание крупных, сложных приложений.
- ❑ Сервисы получаются небольшими и простыми в обслуживании.
- ❑ Сервисы развертываются независимо друг от друга.
- ❑ Сервисы масштабируются независимо друг от друга.
- ❑ Микросервисная архитектура обеспечивает автономность команд разработчиков.
- ❑ Она позволяет экспериментировать и внедрять новые технологии.
- ❑ В ней лучше изолированы неполадки.

Рассмотрим каждое из этих преимуществ.

Делает возможными непрерывные доставку и развертывание крупных, сложных приложений

Главное достоинство микросервисной архитектуры — возможность непрерывных доставки и развертывания крупных, сложных приложений. Как вы увидите в разделе 1.7, непрерывные доставка и развертывание входят в *DevOps* — набор методик

для быстрого, частого и надежного выпуска обновлений. Организации с высокопроизводительным DevOps обычно не испытывают больших проблем с развертыванием изменений в промышленной среде.

Три свойства микросервисной архитектуры делают возможными непрерывные доставку и развертывание.

- ❑ *Она обеспечивает уровень тестируемости, необходимый для непрерывных доставки и развертывания.* Автоматическое тестирование — ключевой аспект непрерывных доставки и развертывания. Поскольку все сервисы в микросервисной архитектуре относительно небольшого размера, написание автоматических тестов значительно упрощается, а их выполнение занимает меньше времени. В результате приложение будет содержать меньше ошибок.
- ❑ *Она обеспечивает уровень развертываемости, необходимый для непрерывных доставки и развертывания.* Каждый сервис может быть развернут независимо от других. Если разработчикам, которые занимаются сервисом, нужно выкатить изменение, затрагивающее только этот сервис, они могут не координировать свои действия с другими командами. Так что частое развертывание изменений в промышленной среде намного упрощается.
- ❑ *Она позволяет сделать команды разработчиков автономными и слабо связанными между собой.* Вы можете организовать отдел разработки в виде набора небольших команд (например, по принципу двух пицц). В этом случае каждая команда полностью отвечает за разработку и развертывание одного или нескольких связанных между собой сервисов. Она может разрабатывать, развертывать и масштабировать свои сервисы независимо от остальных разработчиков (рис. 1.8). Это значительно ускоряет темп разработки.

Возможность выполнять непрерывные доставку и развертывание обеспечивает несколько бизнес-преимуществ.

- ❑ Сокращается время выхода на рынок, что позволяет компании быстро реагировать на отзывы своих клиентов.
- ❑ Компания обеспечивает уровень надежности своих услуг, соответствующий современным ожиданиям.
- ❑ Работники довольны, поскольку вместо тушения пожара они уделяют больше времени выпуску новых возможностей.

Как результат, микросервисная архитектура стала неотъемлемой частью любой компании, зависящей от программных технологий.

Все сервисы невелики и просты в обслуживании

Еще одним преимуществом микросервисной архитектуры является то, что каждый сервис получается сравнительно небольшим. Разработчикам проще разобраться в таком коде. Компактная кодовая база не замедляет работу IDE, что повышает продуктивность. К тому же все сервисы запускаются намного быстрее, чем большой монолит, что тоже делает разработку более продуктивной и ускоряет развертывание.

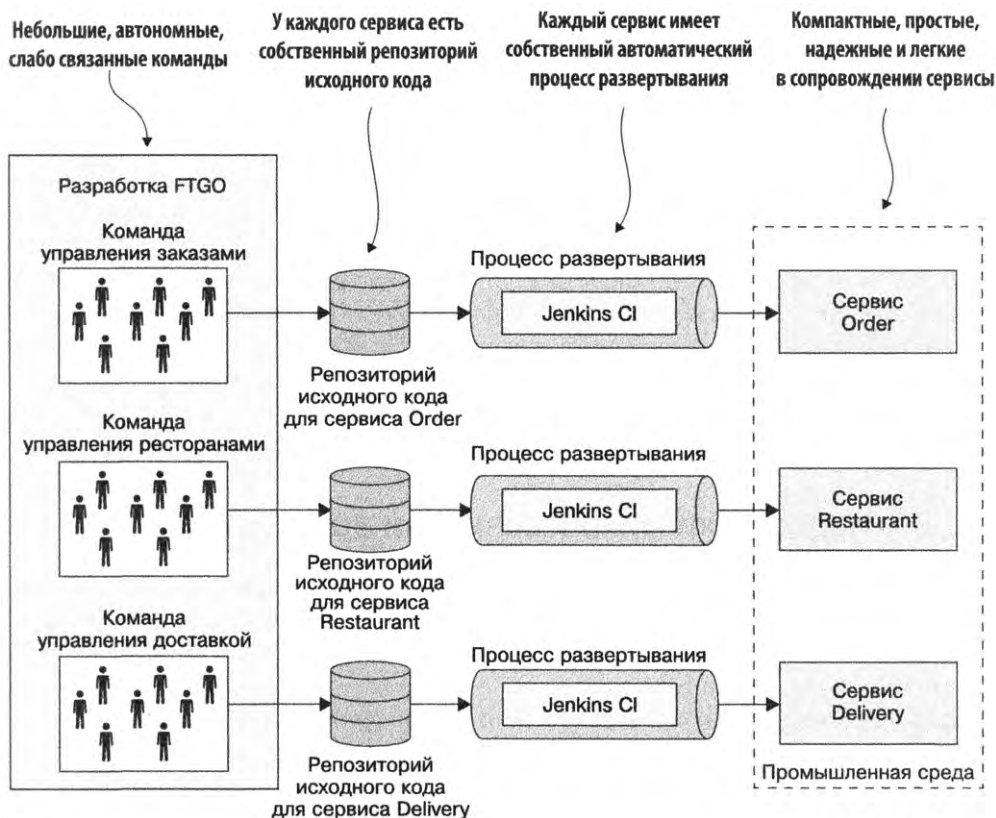


Рис. 1.8. Приложение FTGO, основанное на микросервисах, состоит из набора слабо связанных сервисов. Каждая команда разрабатывает, тестирует и развертывает сервисы независимо от других

Независимое масштабирование сервисов

Каждый сервис в микросервисной архитектуре можно масштабировать отдельно, используя клонирование (ось X) и секционирование (ось Z). Кроме того, любой сервис можно развернуть на оборудовании, которое лучше всего отвечает его требованиям к ресурсам. Для сравнения: в монолитной архитектуре компоненты с разным потреблением ресурсов (например, одним нужен мощный процессор, а другим — много памяти) должны развертываться вместе.

Лучшая изоляция неполадок

В микросервисной архитектуре лучше изолированы неполадки. Например, утечка памяти у сервиса затронет только его, другие сервисы продолжают обрабатывать запросы в обычном режиме. В монолитной же архитектуре вышедший из строя компонент ломает всю систему.

Возможность экспериментировать и внедрять новые технологии

Последним, но немаловажным аспектом является то, что микросервисная архитектура устраняет любую долгосрочную зависимость от стека технологий. В принципе, при создании нового сервиса разработчики могут выбрать наиболее подходящие язык и фреймворки. Во многих организациях этот выбор ограничен, но смысл в том, что вы не зависите от принятых ранее решений.

Более того, поскольку сервисы невелики, вполне реально переписать их с помощью лучших языков и технологий. Если новая технология не оправдала ожиданий, вы можете просто выбросить сделанное, не ставя под угрозу весь проект. При использовании монолитной архитектуры все иначе: технологии, выбранные в самом начале, существенно ограничивают возможность перехода в дальнейшем на другие языки и фреймворки.

1.5.2. Недостатки микросервисной архитектуры

Очевидно, что идеальных технологий не существует и у микросервисной архитектуры тоже есть ряд существенных недостатков и проблем. На самом деле большая часть книги посвящена борьбе с этими изъянами. Но пусть они вас не смущают — позже я объясню, как с ними справиться.

Далее приведены основные недостатки и проблемы микросервисной архитектуры.

- ❑ Сложно подобрать подходящий набор сервисов.
- ❑ Сложность распределенных систем затрудняет разработку, тестирование и развертывание.
- ❑ Развертывание функций, охватывающих несколько сервисов, требует тщательной координации.
- ❑ Решение о том, когда следует переходить на микросервисную архитектуру, является нетривиальным.

По очереди рассмотрим каждую из этих проблем.

Сложности с подбором подходящих сервисов

Одна из проблем, возникающих при использовании микросервисной архитектуры, связана с отсутствием конкретного, хорошо описанного алгоритма разбиения системы на микросервисы. Как и многое в профессии разработчика, это сродни искусству. Но что еще хуже, если вы неправильно разделили систему, у вас получится *распределенный монолит* — набор связанных между собой сервисов, которые необходимо развертывать вместе. Распределенному монолиту присущи недостатки как монолитной, так и микросервисной архитектуры.

Сложность распределенных систем

Еще один недостаток микросервисной архитектуры состоит в том, что при создании распределенных систем возникают дополнительные сложности для разработчиков. Сервисы должны использовать механизм межпроцессного взаимодействия. Это сложнее, чем вызывать обычные методы. К тому же ваш код должен уметь справляться с частичными сбоями и быть готовым к недоступности или высокой латентности удаленного сервиса.

Реализация сценариев, охватывающих несколько сервисов, требует применения незнакомых технологий. Каждый сервис имеет собственную базу данных, что затрудняет реализацию комбинированных транзакций и запросов. Как описывается в главе 4, приложение, основанное на микросервисах, должно использовать так называемые *повествования* (или *сagi*), чтобы сохранять согласованность данных между сервисами. В главе 7 объясняется, что такие приложения не могут извлекать данные из нескольких сервисов с помощью простых запросов. Вместо этого их запросы должны задействовать либо комбинированные API, либо CQRS-представления.

IDE и другие инструменты разработки рассчитаны на создание монолитных приложений и не обеспечивают явной поддержки распределенных приложений. Написание автоматических тестов, затрагивающих несколько сервисов, — непростая задача. Все эти проблемы характерны для микросервисной архитектуры. Следовательно, для успешного применения микросервисов программисты вашей компании должны иметь отточенные навыки разработки и доставки кода.

Вдобавок микросервисная архитектура существенно усложняет администрирование. В промышленной среде приходится иметь дело с множеством экземпляров разнородных сервисов. Для успешного развертывания микросервисов требуется высокая степень автоматизации. Вы должны использовать технологии следующего плана:

- ❑ инструменты для автоматического развертывания, такие как Netflix Spinnaker;
- ❑ общедоступную платформу PaaS, такую как Pivotal Cloud Foundry или Red Hat OpenShift;
- ❑ платформу оркестрации для Docker, такую как Docker Swarm или Kubernetes.

Подробнее о вариантах развертывания я расскажу в главе 12.

Развертывание функций, охватывающих несколько сервисов, требует тщательной координации

Еще одна проблема микросервисной архитектуры связана с тем, что развертывание функций, охватывающих несколько сервисов, требует тщательной координации действий разных команд разработки. Вам необходимо выработать план «выкатывания» обновлений, который запрашивает развертывание сервисов с учетом их зависимостей. Для сравнения: в монолитной архитектуре вы можете легко выпускать автоматические обновления для нескольких компонентов.

Нетривиальность решения о переходе

Еще одна трудность связана с решением о том, на каком этапе жизненного цикла приложения следует переходить на микросервисную архитектуру. Часто во время разработки первой версии вы еще не сталкиваетесь с проблемами, которые эта архитектура решает. Более того, применение сложного, распределенного метода проектирования замедлит разработку. Для стартапов, которым обычно важнее всего как можно быстрее развить свою бизнес-модель и сопутствующее приложение, это может вылиться в непростую дилемму. Использование микросервисной архитектуры делает выпуск начальных версий довольно трудным. Стартапам почти всегда лучше начинать с монолитного приложения.

Однако со временем возникает другая проблема: как справиться с возрастающей сложностью. Это подходящий момент для того, чтобы разбить приложение на микросервисы с разными функциями. Рефакторинг может оказаться непростым из-за запутанных зависимостей. В главе 13 мы поговорим о стратегиях перехода с монолитной архитектуры на микросервисную.

Как видите, несмотря на множество достоинств, микросервисы обладают и существенными недостатками. В связи с этим к переходу на них следует относиться очень серьезно. Но обычно для сложных проектов, таких как пользовательские веб- или SaaS-приложения, это правильный выбор. Такие общеизвестные сайты, как eBay (www.slideshare.net/RandyShoup/the-ebay-architecture-striking-a-balance-between-site-stability-feature-velocity-performance-and-cost), Amazon.com, Groupon и Gilt, в свое время перешли на микросервисы с монолитной архитектурой.

При использовании микросервисов приходится иметь дело с множеством проблем, связанных с проектированием и архитектурой. К тому же многие из этих проблем имеют несколько решений со своими плюсами и минусами. Единого идеального решения не существует. Чтобы помочь вам сделать выбор, я создал язык шаблонов микросервисной архитектуры и буду ссылаться на него на страницах книги. Посмотрим, что представляет собой этот язык и почему он вам пригодится.

1.6. Язык шаблонов микросервисной архитектуры

Архитектура и проектирование в конечном итоге сводятся к принятию решений. Вам нужно решить, какая архитектура лучше всего подходит для вашего приложения — монолитная или микросервисная. Делая этот выбор, вы должны взвесить большое количество за и против. Если остановитесь на микросервисах, вам придется столкнуться с множеством вызовов.

Язык шаблонов — хороший способ описания архитектурных и проектировочных методик и помогает принять решение. Сначала поговорим о том, зачем нужны шаблоны проектирования и соответствующий язык, а затем пройдемся по языку шаблонов микросервисной архитектуры.

1.6.1. Микросервисная архитектура не панацея

В 1986 году Фред Брукс (Fred Brooks), автор книги *The Mythical Man-Month* (Addison-Wesley Professional, 1995)¹, высказал мнение, что при разработке программного обеспечения не существует универсальных решений. Это означает, что нет таких методик или технологий, применение которых повысит вашу продуктивность в десять раз. Тем не менее и по прошествии нескольких десятилетий программисты продолжают рьяно спорить о своих любимых инструментах, будучи уверенными в том, что те дают им огромное преимущество в работе.

Во многих дискуссиях такого рода мнения слишком расходятся. Для этого феномена даже есть специальный термин — *Suck/Rock Dichotomy* (<http://nealford.com/memeagora/2009/08/05/suck-rock-dichotomy.html>, Нил Форд (Neal Ford)), который иллюстрирует ситуацию, когда в мире программного обеспечения все либо очень хорошо, либо очень плохо. Такая аргументация имеет следующий вид: если вы сделаете X, произойдет катастрофа, поэтому вы должны сделать Y. Примером может служить противостояние между синхронным и реактивным программированием, объектно-ориентированным и функциональным подходами, Java и JavaScript, REST и обменом сообщениями. В реальности, естественно, все немного сложнее. У каждой технологии есть недостатки и ограничения, ее приверженцы их часто игнорируют. В итоге переход на ту или иную технологию обычно происходит в соответствии с *циклом зрелости* (https://ru.wikipedia.org/wiki/Gartner#Цикл_хайпа), состоящим из пяти стадий, включая *пик чрезмерных ожиданий* (мы нашли панацею), *избавление от иллюзий* (это никуда не годится) и *плато продуктивности* (теперь мы понимаем все плюсы и минусы и знаем, когда это лучше применять).

Микросервисы в этом смысле не исключение. Подходит ли данная архитектура для вашего приложения, зависит от множества факторов. Следовательно, нельзя воспринимать их как решение на все случаи жизни, равно как и не стоит полностью от них отказываться. Как часто бывает, нужно учитывать конкретную ситуацию.

Основной причиной непримиримых споров о технологиях является то, что людям зачастую движут эмоции. В превосходной книге *The Righteous Mind: Why Good People Are Divided by Politics and Religion* (Vintage, 2013) Джонатан Хайдт (Jonathan Haidt) описывает работу человеческого разума на примере слона и наездника. Слон представляет собой эмоциональную составляющую нашего мозга, которая принимает большинство решений. Наездник представляет рациональную часть: иногда он может повлиять на слона, но чаще всего лишь рационализирует то, что слон уже и так сделал.

Мы, как сообщество разработчиков программного обеспечения, должны побороть свою эмоциональность и найти лучший способ обсуждения и применения технологий. Отличным методом обсуждения и описания технологий является формат *шаблонов проектирования* (ввиду своей объективности). Например, описывая с его помощью инструмент разработки, вы должны упомянуть и о недостатках. Рассмотрим этот формат.

¹ Брукс Ф. Мифический человеко-месяц. — М.: Символ-Плюс, 2010.

1.6.2. Шаблоны проектирования и языки шаблонов

Шаблон проектирования — это многоразовое решение проблемы, возникающей в определенном контексте. Это идея, которая возникает как часть реальной архитектуры и затем показывает себя с лучшей стороны при проектировании программного обеспечения. Концепцию шаблонов предложил Кристофер Александер (Christopher Alexander), практикующий архитектор программных систем. Ему также принадлежит идея *языка шаблонов* — набора шаблонов проектирования, которые решают проблемы в определенной области. В его книге *A Pattern Language: Towns, Buildings, Construction* (Oxford University Press, 1977) описывается язык для архитектуры, состоящей из 253 шаблонов. Там можно найти решения и для высокоуровневых проблем, таких как поиск места для города («доступ к воде»), и для узких задач наподобие проектирования комнаты («освещение двух сторон каждой комнаты»). Каждый из этих шаблонов решает проблему путем упорядочения физических объектов, варьирующихся от целых городов до отдельных окон.

Идеи Кристофера Александера помогли концепциям шаблонов проектирования и языков шаблонов стать общепринятыми в среде программистов. Объектно-ориентированные шаблоны собраны в книге Эриха Гаммы (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralph Johnson) и Джона Влиссидеса (John Vlissides) *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional, 1994)¹. Эта книга сделала шаблоны проектирования популярными у разработчиков. Начиная с середины 1990-х программисты задокументировали бесчисленное количество шаблонов. *Программный шаблон* решает архитектурную проблему, определяя ряд взаимодействующих между собой программных элементов.

Представьте, что вы, к примеру, создаете электронный банкинг, который должен учитывать разнообразные правила превышения кредита. Каждое правило описывает ограничение баланса на счете и комиссию, которая снимается при исчерпании кредитных средств. Эту задачу можно решить с помощью широко известного шаблона «Стратегия» из вышеупомянутой книги. Решение состоит из трех частей:

- ❑ интерфейса *Overdraft*, который инкапсулирует алгоритм превышения кредита;
- ❑ одного или нескольких классов-реализаций, по одному для каждого конкретного контекста;
- ❑ класса *Account*, который использует наш алгоритм.

Шаблон проектирования «Стратегия» является объектно-ориентированным, поэтому элементы решения выполнены в виде классов. Позже в этом разделе я опишу *высокоуровневые* шаблоны проектирования, в которых решения состоят из взаимодействующих между собой сервисов.

¹ Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2016.

Одна из причин, почему шаблоны проектирования так ценятся, — они должны описывать контекст, в котором их можно применять. То, что решение ограничено определенным контекстом и может плохо работать в других ситуациях, — это шаг вперед по сравнению с тем, как обсуждение технологий происходило раньше. Например, решение, которое подходит в масштабах компании Netflix, может оказаться не самым лучшим для приложения с несколькими пользователями.

Необходимость учитывать контекст далеко не единственная положительная сторона шаблонов проектирования. Они заставляют вас описывать другие важные аспекты решения, которые часто упускают из виду. Распространенная структура шаблонов включает в себя три особо важных раздела:

- ❑ причины;
- ❑ итоговый контекст;
- ❑ родственные шаблоны.

Рассмотрим каждый из них, начиная с причин.

Причины: аспекты решаемой вами проблемы

Раздел «*причины*» (forces) в шаблоне проектирования описывает различные аспекты проблемы, которую вы решаете в заданном контексте. Они могут противоречить друг другу, поэтому иногда проблему нельзя решить полностью. То, какие из них более значимы, зависит от контекста, поэтому вы должны расставить приоритеты. Например, код должен быть понятным и производительным. Код, написанный в реактивном стиле, производительнее, чем синхронный код, но зачастую в нем сложнее разобраться. Явное указание причин помогает определиться с тем, какие из аспектов проблемы необходимо решить.

Итоговый контекст: последствия применения шаблона

Раздел «*итоговый контекст*» (resulting context) описывает последствия применения шаблона проектирования. Он состоит из трех частей.

- ❑ *Преимущества* — преимущества шаблона, включая аспекты проблемы, которые он решает.
- ❑ *Недостатки* — недостатки шаблона, включая нерешенные аспекты проблемы.
- ❑ *Замечания* — новые проблемы, появляющиеся в результате применения шаблона.

Итоговый контекст формирует более комплексное и объективное представление о решении, что помогает сделать правильный выбор при проектировании.

Родственные шаблоны пяти типов

Раздел «*родственные шаблоны*» (related patterns) описывает связь между действующим и другими шаблонами проектирования. Связь бывает пяти типов.

- ❑ *Предшественник* — предшествующий шаблон, который обосновывает потребность в данном шаблоне. Например, микросервисная архитектура — это предшественник всех остальных шаблонов в языке шаблонов, кроме монолитной архитектуры.
- ❑ *Преемник* — шаблон, который решает проблемы, порожденные данным шаблоном. Например, при использовании микросервисной архитектуры необходимо применить целый ряд шаблонов-преемников, включая обнаружение сервисов и шаблон «Предохранитель».
- ❑ *Альтернатива* — альтернативное решение по отношению к данному шаблону. Например, монолитная и микросервисная архитектуры — это альтернативные способы проектирования приложения. Нужно выбрать одну из них.
- ❑ *Обобщение* — обобщенное решение проблемы. Например, в главе 12 представлены разные реализации шаблона «Один сервис — один сервер».
- ❑ *Специализация* — специализированная разновидность шаблона. Например, в главе 12 вы узнаете, что развертывание сервиса в виде контейнера — это частный случай шаблона «Один сервис — один сервер».

Кроме того, можно группировать шаблоны по областям, в которых их применяют. Явное описание родственных шаблонов помогает получить представление о том, как эффективно решить ту или иную проблему.

Пример визуального представления связей между шаблонами приведен на рис. 1.9.

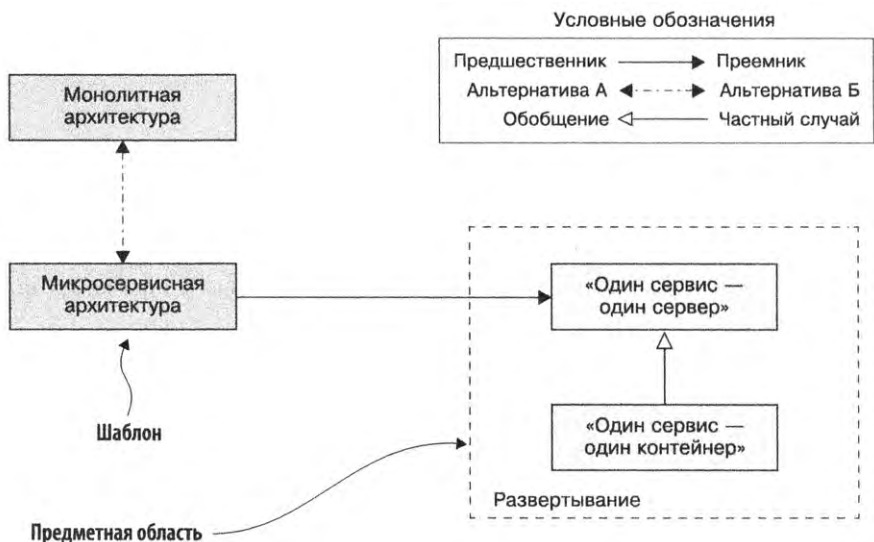


Рис. 1.9. Визуальное представление разного вида связей между шаблонами

На рис. 1.9 представлены следующие типы связей между шаблонами:

- ❑ шаблон-преемник решает проблему, возникшую в результате применения шаблона-предшественника;
- ❑ у одной и той же проблемы может быть несколько альтернативных решений;
- ❑ один шаблон может быть специализированной версией другого;
- ❑ шаблоны, решающие проблемы в одной и той же области, можно сгруппировать или обобщить.

Набор шаблонов проектирования, имеющих подобные связи, иногда можно объединить в так называемый язык шаблонов. Шаблоны, входящие в него, совместно работают над решением проблем в определенной области. Например, я создал язык шаблонов микросервисной архитектуры. Это набор взаимосвязанных методик для проектирования микросервисов. Рассмотрим его подробнее.

1.6.3. Обзор языка шаблонов микросервисной архитектуры

Язык шаблонов микросервисной архитектуры — это набор методик, которые помогают в проектировании приложений на основе микросервисов. Общая структура этого языка показана на рис. 1.10. Прежде всего он помогает определиться с тем, нужно ли вам использовать микросервисную архитектуру. Он описывает оба подхода, микросервисный и монолитный, вместе с их достоинствами и недостатками. А если микросервисная архитектура подходит для вашего приложения, язык шаблонов поможет эффективно ее задействовать, решая различные проблемы проектирования.

Данный язык состоит из нескольких групп шаблонов. В левой части рис. 1.10 представлена группа шаблонов архитектуры приложения, в которую входят монолитные и микросервисные методы проектирования. Это шаблоны, которые обсуждаются в этой главе. Остальная часть языка содержит группы шаблонов для решения проблем, порожденных микросервисной архитектурой. Шаблоны также делятся на три уровня.

- ❑ *Инфраструктурные шаблоны* — решают проблемы, в основном касающиеся инфраструктуры и не относящиеся к разработке.
- ❑ *Инфраструктура приложения* — предназначены для инфраструктурных задач, влияющих на разработку.
- ❑ *Шаблоны приложения* — решают проблемы, с которыми сталкиваются разработчики.

Шаблоны группируются в зависимости от того, какого рода проблемы они решают. Рассмотрим основные группы шаблонов.

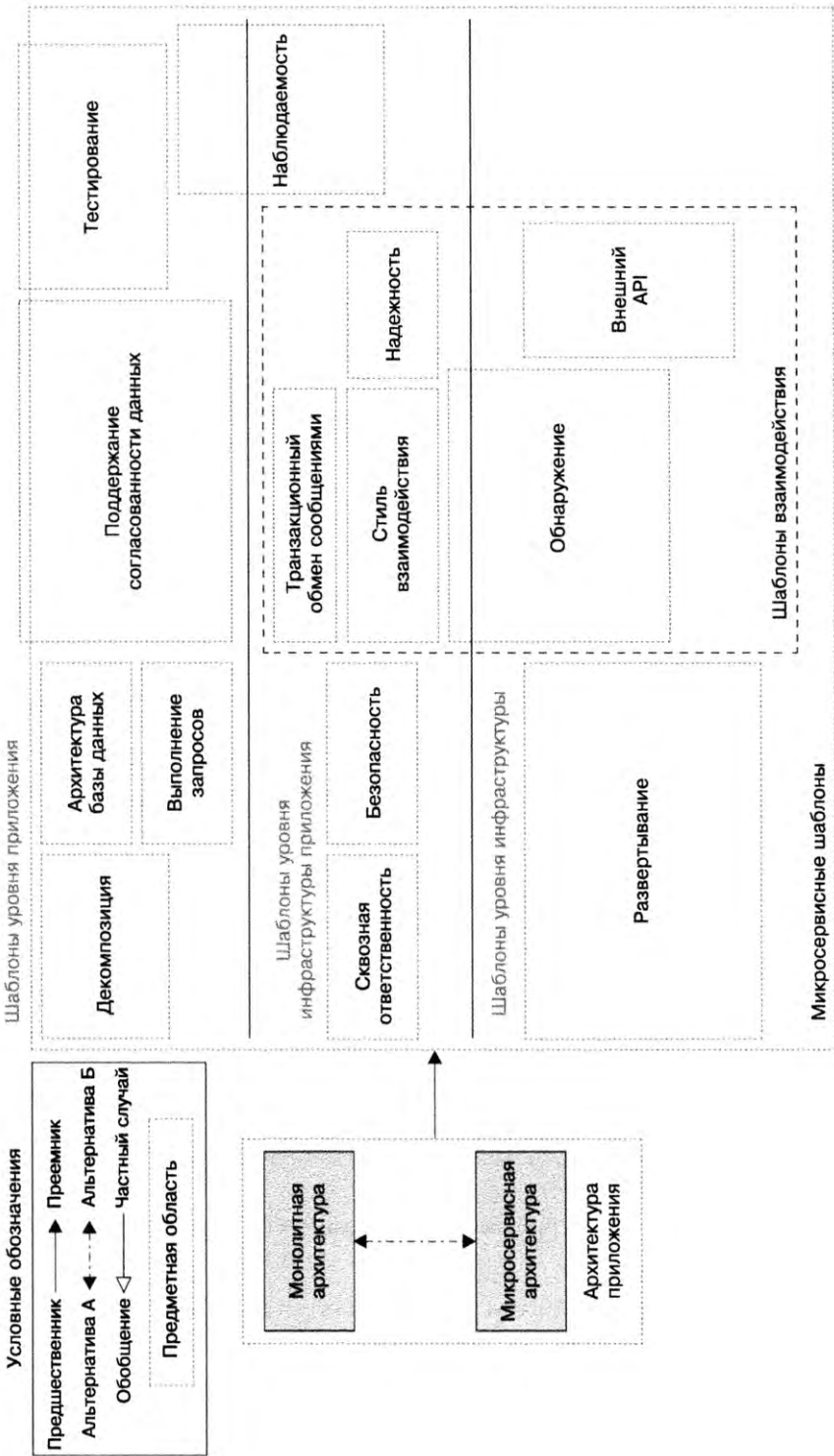


Рис. 1.10. Общая структура языка шаблонов микросервисной архитектуры, описывающая разные проблемные области, на которые рассчитаны шаблоны. Слева находятся методики проектирования приложения — монолитного и микросервисного. Остальные группы шаблонов решают проблемы, возникающие в результате выбора микросервисной архитектуры

Шаблоны для разбиения приложения на микросервисы

Решение о том, каким образом разбить систему на набор сервисов, сродни искусству, но в этом вам может помочь целый ряд стратегий. Два шаблона декомпозиции (рис. 1.11) по-разному подходят к определению архитектуры приложения.



Рис. 1.11. Существует два метода декомпозиции: по бизнес-возможностям (когда сервисы группируются на основе бизнес-возможностей) и по проблемным областям (согласно предметно-ориентированному проектированию)

Эти шаблоны подробно описываются в главе 2.

Шаблоны взаимодействия

Приложение, основанное на микросервисной архитектуре, является распределенной системой. Важную роль в этой архитектуре играет межпроцессное взаимодействие (interprocess communication, IPC). Вам придется принять ряд архитектурных решений о том, как ваши сервисы будут взаимодействовать друг с другом и с внешним миром. На рис. 1.12 показаны шаблоны взаимодействия, разделенные на пять групп.

- ❑ *Стиль взаимодействия.* Какой механизм IPC следует использовать?
- ❑ *Обнаружение.* Каким образом клиент сервиса узнает его IP-адрес, чтобы, например, выполнить HTTP-запрос?
- ❑ *Надежность.* Как обеспечить надежное взаимодействие между сервисами с учетом того, что некоторые из них могут быть недоступны?
- ❑ *Транзакционный обмен сообщениями.* Как следует интегрировать отправку сообщений и публикацию событий с транзакциями баз данных, которые обновляют бизнес-информацию?
- ❑ *Внешний API.* Каким образом клиенты вашего приложения взаимодействуют с сервисами?

В главе 3 рассматриваются первые четыре группы шаблонов: стиль взаимодействия, обнаружение, надежность и транзакционный обмен сообщениями. В главе 8 мы остановимся на шаблонах внешнего API.

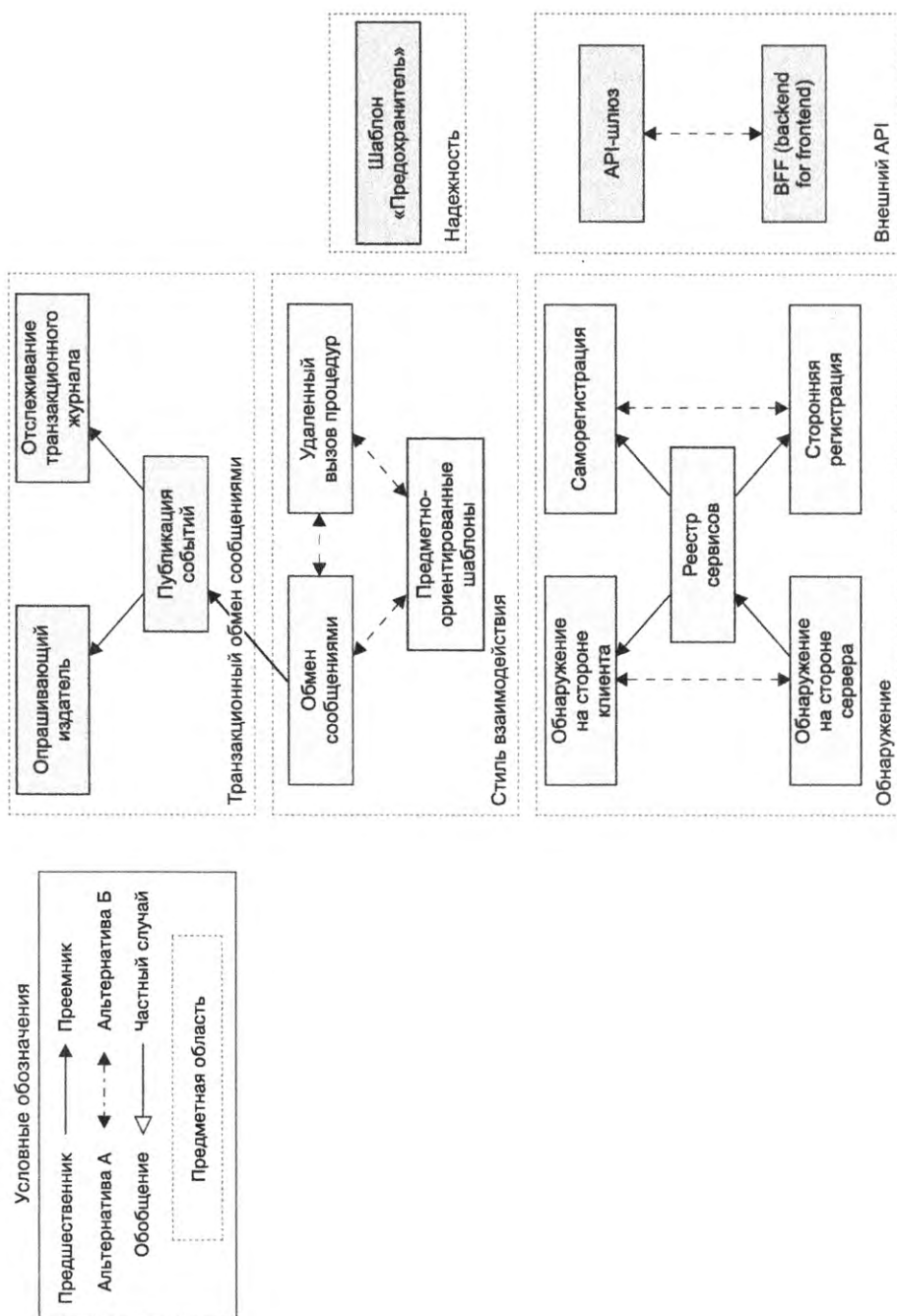


Рис. 1.12. Пять групп коммуникационных шаблонов

Шаблоны согласованности данных для реализации управления транзакциями

Как упоминалось ранее, для обеспечения слабой связанности каждый сервис должен иметь собственную базу данных. К сожалению, такой подход чреват некоторыми существенными проблемами. Из главы 4 вы узнаете, что традиционная методика с использованием распределенных транзакций (2PC) не подходит для современных приложений. Вместо этого согласованность данных следует обеспечивать с помощью шаблона «Повествование». Шаблоны, связанные с данными, показаны на рис. 1.13.

Все эти шаблоны подробно описываются в главах 4–6.



Рис. 1.13. Поскольку каждый сервис работает с собственной БД, для согласования данных между разными сервисами требуется шаблон «Повествование»

Шаблоны запрашивания данных в микросервисной архитектуре

Использование отдельной базы данных в каждом сервисе имеет еще один недостаток: некоторые запросы должны объединять информацию, принадлежащую нескольким сервисам. Данные сервиса доступны только через его API, поэтому вы не можете применять к его БД распределенные запросы. На рис. 1.14 показаны несколько шаблонов, с помощью которых можно реализовать запросы.



Рис. 1.14. Поскольку каждый сервис работает с собственной БД, для извлечения данных, разбросанных по нескольким сервисам, следует использовать один из шаблонов запросов

Иногда можно применять шаблон объединения API, который обращается к API одного или нескольких сервисов и агрегирует результаты. В некоторых ситуациях

приходится прибегать к командным запросам с разделением ответственности (command query responsibility segregation, CQRS), которые хранят одну или несколько копий данных и позволяют легко к ним обращаться.

Шаблоны развертывания сервисов

Процесс развертывания монолитного приложения не всегда прост, но он прямолинеен в том смысле, что нужно развернуть лишь одно приложение, многочисленные экземпляры которого будут находиться за балансировщиком нагрузки.

Приложение, основанное на микросервисах, намного сложнее. У вас могут быть десятки и сотни сервисов, написанных на различных языках с использованием разных фреймворков. И придется управлять намного большим количеством элементов. Шаблоны развертывания показаны на рис. 1.15.



Рис. 1.15. Несколько шаблонов развертывания микросервисов. Традиционный подход состоит в развертывании сервисов в формате упаковки определенного языка. У него есть две современные альтернативы. Первая — развертывание сервисов в виде виртуальных машин (VM) или контейнеров. Вторая — бессерверные технологии: вы просто загружаете свой код, а бессерверная платформа его выполняет. Следует использовать автоматизированную платформу с поддержкой самообслуживания для развертывания и администрирования сервисов

Традиционный (часто ручной) способ развертывания приложений с применением форматов упаковки, характерных для определенного языка (например, WAR-файлов), нельзя масштабировать для поддержки микросервисной архитектуры. Вам нужна высокоавтоматизированная инфраструктура развертывания. В идеале следует использовать платформу, которая позволяет разработчику развертывать

и администрировать свои сервисы с помощью простого пользовательского интерфейса — консольного или графического. Такие платформы обычно основаны на виртуальных машинах (ВМ), контейнерах или бессерверных технологиях. Разные варианты развертывания рассматриваются в главе 12.

Шаблоны наблюдаемости позволяют понять, как себя ведет приложение

Ключевыми моментами администрирования приложения являются понимание того, как оно ведет себя во время работы, и диагностика таких проблем, как неудачные запросы и продолжительное время ожидания. Монолитные приложения не всегда легко анализировать и диагностировать, но относительно простой и прямолинейный механизм обработки запросов облегчает эти задачи. Балансировщик нагрузки направляет каждый входящий запрос к определенному экземпляру приложения, которое выполняет небольшое количество запросов к базе данных и возвращает ответ. Например, если нужно понять, как был обработан тот или иной запрос, вы можете просмотреть журнальный файл того экземпляра приложения, который этим занимался.

Анализ и диагностика проблем в микросервисной архитектуре намного сложнее. Запрос может «прыгать» от одного сервиса к другому, прежде чем клиент получит ответ. Таким образом, вы не можете проследить этот процесс в едином журнальном файле. Точно так же усложняется диагностика времени ожидания, ведь корень проблемы может находиться в разных местах.

Для проектирования наблюдаемых сервисов можно использовать следующие шаблоны.

- ❑ *API проверки работоспособности.* Создайте конечную точку, которая возвращает данные о состоянии сервиса.
- ❑ *Агрегация журналов.* Записывайте поведение сервисов и сохраняйте эти записи на центральном сервере с поддержкой поиска и оповещений.
- ❑ *Распределенная трассировка.* Назначайте каждому внешнему запросу уникальный идентификатор и отслеживайте его перемещение между сервисами.
- ❑ *Отслеживание исключений.* Отправляйте отчеты об исключениях соответствующему сервису, который их дедуплицирует, оповещает разработчиков и отслеживает разрешение каждой исключительной ситуации.
- ❑ *Показатели приложения.* Собирайте количественные и оценочные показатели и делайте их доступными для соответствующего сервиса.
- ❑ *Ведение журнала аудита.* Записывайте в журнал действия пользователей.

Эти шаблоны подробно описываются в главе 11.

Шаблоны автоматического тестирования сервисов

По сравнению с монолитными приложениями микросервисная архитектура облегчает тестирование отдельных сервисов, так как их размер намного меньше. Но в то же время важно убедиться в том, что отдельные сервисы хорошо работают вместе,

избегая при этом использования сложных, медленных и ненадежных сквозных тестов. Следующие шаблоны упрощают эту задачу, позволяя тестировать сервисы изолированно друг от друга.

- ❑ *Тестирование контрактов с расчетом на потребителя* — проверка того, что сервис отвечает ожиданиям клиентов.
- ❑ *Тестирование контрактов на стороне потребителя* — проверка того, что клиент может взаимодействовать с сервисом.
- ❑ *Тестирование компонентов сервиса* — тестирование сервиса в изоляции.

Эти шаблоны тестирования подробно описываются в главах 9 и 10.

Шаблоны для решения сквозных проблем

Микросервисная архитектура предусматривает многочисленные аспекты, которые должны быть реализованы в каждом сервисе, включая шаблоны наблюдаемости и обнаружения. Они также должны реализовать шаблон вынесения конфигурации вовне, который предоставляет сервису такую информацию, как параметры подключения к базе данных на этапе выполнения. Написание всех этих функций с нуля для каждого отдельного сервиса заняло бы слишком много времени. Намного лучше применять шаблон шасси микросервисов и строить сервисы на основе фреймворков с поддержкой этих возможностей. Подробнее об этом — в главе 11.

Шаблоны безопасности

В микросервисной архитектуре аутентификацию пользователей обычно выполняет API-шлюз, который затем должен передать учетную информацию, например идентификатор и роли, вызываемому сервису. Часто для этого применяют токены доступа, такие как JWT (JSON Web token). API-шлюз передает токен сервисам, которые могут его проверить и извлечь из него информацию о пользователе. Шаблон «Токен доступа» подробно обсуждается в главе 11.

Основное внимание в языке шаблонов микросервисной архитектуры уделяется решению задач проектирования, что неудивительно. Очевидно, что для успешной разработки программного обеспечения нужно выбрать подходящую архитектуру, но это еще не все. Вы также должны учитывать такие аспекты, как процесс и организация разработки и доставки.

1.7. Помимо микросервисов: процесс и организация

Микросервисная архитектура обычно становится лучшим решением для крупных и сложных приложений. Но создание успешного продукта требует не только выбора подходящей архитектуры, но и правильной организации и налаживания процессов разработки и доставки. Взаимосвязь между процессом, организацией и архитектурой показана на рис. 1.16.



Рис. 1.16. Регулярный и оперативный выпуск обновлений для крупных и сложных приложений требует сочетания микросервисной архитектуры с DevOps, включая непрерывные доставку, развертывание и небольшие автономные команды

С архитектурой мы уже определились. Рассмотрим организацию и процесс.

1.7.1. Организация разработки и доставки программного обеспечения

Успешная деятельность организации сопряжена с расширением команды разработки. В чем-то это хорошо, ведь чем больше разработчиков, тем больше они могут сделать. Но, как пишет Фред Брукс в книге «Мифический человеко-месяц», затраты на взаимодействие в команде размера N составляют $O(N^2)$. Если команда слишком разрастается, она становится неэффективной из-за медленной коммуникации между участниками. Представьте себе ежедневные пятиминутки, на которых присутствуют 20 человек...

Решение заключается в разбиении большой команды на несколько мелких, численностью не более 8–12 человек. У каждой команды есть четкая бизнес-ориентированная цель: разработка и по возможности администрирование одного или нескольких сервисов, которые реализуют определенную возможность или бизнес-функцию. Команда должна быть многопрофильной и уметь разрабатывать, тестировать и развертывать свои сервисы, не требуя регулярного взаимодействия или координации с другими командами.

Обратный маневр Конвея

Для эффективной доставки программного обеспечения при использовании микросервисной архитектуры необходимо учитывать закон Конвея (ru.wikipedia.org/wiki/Закон_Конвея), который гласит: «Организация, проектирующая системы, обречена

воспроизводить архитектуру, имитирующую структуру собственных коммуникаций» (Мелвин Конвей (Melvin Conway)).

Иными словами, архитектура приложения отражает структуру организации, которая его разработала. По этой причине закон Конвея следует применять задом наперед (www.thoughtworks.com/radar/techniques/inverse-conway-maneuver) и проектировать свою организацию так, чтобы ее структура отражала микросервисную архитектуру. Благодаря этому ваши команды разработчиков будут такими же слабо связанными, как и сами сервисы.

Несколько мелких команд выигрывают в продуктивности у одной крупной. Как говорилось в подразделе 1.5.1, микросервисная архитектура играет ключевую роль в возможности сделать команды автономными. Каждая из них может разрабатывать, развертывать и масштабировать свои сервисы без согласования с другими командами. Кроме того, вы всегда будете знать, к кому обращаться, если сервис не соответствует оговоренному уровню услуг.

Помимо прочего, такой подход значительно повышает масштабируемость организации. Она может расширяться, присоединяя новые команды. Если какая-то из команд станет слишком большой, можно разбить ее и связанные с ней сервисы на несколько частей. И поскольку команды слабо связаны между собой, вы можете избежать проблем с коммуникацией, присущих большим организациям. В итоге появление новых людей не скажется на продуктивности.

1.7.2. Процесс разработки и доставки программного обеспечения

Использование микросервисной архитектуры в сочетании с каскадной моделью разработки напоминает езду на Ferrari, в который запряжена лошадь, — большинство преимуществ микросервисов попросту теряется. Если вы хотите создавать приложения с микросервисной архитектурой, крайне важно внедрить гибкие методики разработки и развертывания, такие как Scrum или Kanban. В дополнение к этому следует практиковать непрерывные доставку и развертывание, которые являются частью DevOps.

Джез Хамбл (Jez Humble) предлагает следующее определение непрерывной доставки: *«Непрерывная доставка — это возможность доставлять изменения всех типов (включая новые возможности, обновления конфигурации, заплатки и экспериментальные функции) в продукт или пользователям безопасным, быстрым и устойчивым способом»* (<https://continuousdelivery.com/>).

Ключевая характеристика непрерывной доставки состоит в том, что программное обеспечение всегда готово к выпуску. Это требует высокого уровня автоматизации, в том числе автоматического тестирования. Логическим развитием является непрерывное автоматическое развертывание готового кода в промышленной среде. Высокопродуктивные организации, практикующие этот подход, развертывают из-

менения по несколько раз в день, сталкиваются с меньшим количеством перебоев в промышленной среде, но если это все же произошло, в состоянии быстро восстановить работу (puppet.com/resources/whitepaper/state-of-devops-report). Как упоминалось в подразделе 1.5.1, микросервисная архитектура напрямую поддерживает непрерывные доставки и развертывание.

Быстро продвигайтесь вперед, не ломая ничего на своем пути

Цель непрерывных доставки и развертывания (и DevOps в целом) — быстрое, но надежное выкатывание изменений. Существует четыре показателя, позволяющих оценить процесс разработки.

- *Частота развертывания.* Как часто программное обеспечение развертывается в промышленной среде.
- *Время выполнения.* Время между регистрацией изменения и его развертыванием.
- *Среднее время восстановления.* Время восстановления после промышленного сбоя.
- *Частота неудачных изменений.* Процент изменений, которые приводят к проблемам в промышленной среде.

В условиях традиционной организации частота развертывания получается низкой, а время выполнения — продолжительным. Изнуренные разработчики и администраторы обычно задерживаются допоздна, пытаются успеть исправить все проблемы до окончания периода обслуживания. Для сравнения: организации, работающие в стиле DevOps, выпускают свой код часто, иногда по несколько раз в день, и имеют куда меньше проблем в промышленной среде. Например, по состоянию на 2014 год компания Amazon развертывала изменения каждые 11,6 с (www.youtube.com/watch?v=dxk8b9rSKOo), а время выполнения одного компонента в Netflix занимает 16 мин (medium.com/netflixtechblog/how-we-build-code-at-netflix-c5d9bd727f15).

1.7.3. Человеческий фактор при переходе на микросервисы

При переходе на микросервисы меняются архитектура, организация труда и процесс разработки. Но при этом меняется и рабочая среда в коллективе, а люди, как уже говорилось, существа эмоциональные. Если этого не учесть, их эмоции могут сделать такой переход не самым гладким. Мэри и другие руководители столкнутся с проблемами при изменении способа разработки в компании FTGO.

В своем бестселлере *Managing Transitions* (Da Capo Lifelong Books, 2017; <https://wmbridges.com/books>) Уильям и Сьюзан Бриджес (William & Susan Bridges) предлагают концепцию *перехода*, которая описывает эмоциональную реакцию людей на изменения. Модель перехода состоит из трех стадий.

1. *Конец, потеря, смирение.* Период эмоционального сдвига и сопротивления, когда люди сталкиваются с изменением, которое заставляет их покинуть зону комфорта. Они часто скупают по старым подходам. Например, если разработчиков

разбить на многопрофильные команды, им будет не хватать прежних товарищей. Точно так же отдел моделирования, отвечающий за глобальную модель данных, придет в ужас от того, что у каждого сервиса теперь будет своя модель.

2. *Нейтральная зона.* Промежуточная стадия между старым и новым порядком, в это время люди часто пребывают в замешательстве и испытывают трудности при изучении новых методов.
3. *Новое начало.* Завершающая стадия, когда люди с энтузиазмом принимают новый подход к работе и начинают ощущать его преимущества.

Книга описывает то, как лучше всего справиться с каждой стадией перехода и повысить шансы на успех внедрения изменений. Очевидно, что компания FTGO страдает из-за монолитного ада и нуждается в миграции на микросервисную архитектуру. Но это может изменить ее структуру и процесс разработки. Чтобы не потерпеть фиаско, следует учитывать модель перехода и эмоции сотрудников.

В следующей главе мы поговорим о назначении программной архитектуры и о том, как разбить приложение на сервисы.

Резюме

- ❑ В соответствии с монолитной архитектурой приложение структурируется в виде единой развертываемой сущности.
- ❑ В микросервисной архитектуре система разбивается на независимо развертываемые сервисы, каждый со своей базой данных.
- ❑ Монолитная архитектура подходит для простых приложений, а микросервисы обычно являются лучшим решением для крупных, сложных систем.
- ❑ Микросервисная архитектура ускоряет темп разработки программного обеспечения, позволяя небольшим автономным командам работать параллельно.
- ❑ Микросервисная архитектура не панацея. У нее есть существенные недостатки, такие как повышенная сложность.
- ❑ Язык шаблонов микросервисной архитектуры — это набор методик, которые облегчают проектирование приложений на основе микросервисов. Он помогает решить, следует ли использовать микросервисную архитектуру, и, если она вам подходит, эффективно ее применять.
- ❑ Для ускорения доставки программного обеспечения одной микросервисной архитектуры недостаточно. Чтобы разработка оказалась успешной, вам также нужно задействовать DevOps и сформировать небольшие автономные команды.
- ❑ Не забывайте о человеческом факторе перехода на микросервисы. Чтобы он стал успешным, следует учитывать эмоциональный настрой работников.

Стратегии декомпозиции

В этой главе

- Архитектура программного обеспечения и ее важность.
- Разбиение приложения на сервисы путем применения шаблона декомпозиции. Декомпозиция по бизнес-функциям и проблемным областям.
- Использование принципа изолированного контекста из предметно-ориентированного программирования для распутывания данных и упрощения декомпозиции.

Иногда следует быть осторожным со своими желаниями. Усердно рекламируя микросервисную архитектуру, Мэри наконец-то убедила руководство в том, что миграция на нее будет правильным решением. Одновременно воодушевленная и обеспокоенная, Мэри собрала архитекторов и провела с ними все утро, обсуждая, с чего лучше начать. В ходе этого стало очевидно, что некоторые аспекты языка шаблонов микросервисной архитектуры, такие как развертывание и обнаружение сервисов, оказались новыми и незнакомыми, хотя и довольно тривиальными. Основным вызовом стало разбиение приложения на сервисы, то есть самая суть данной архитектуры. Таким образом, важнейшим вопросом стало определение сервиса. Столпившись у доски для рисования, члены команды FTGO пытались понять, что именно под ним подразумевается!

В этой главе вы научитесь определять микросервисную архитектуру. Я опишу стратегии разбиения приложения на сервисы. Вы увидите, что сервисы организуются скорее вокруг бизнес-проблем, а не технических аспектов. Я также покажу, как с помощью принципов предметно-ориентированного проектирования устранить так называемые божественные классы (классы, которые используются в разных частях приложения и создают запутанные зависимости, препятствующие декомпозиции).

Начнем эту главу с определения микросервисной архитектуры в терминах проектирования программного обеспечения. После этого я опишу процесс определения микросервисной архитектуры для приложения, начиная с его требований. Мы обсудим стратегии разбиения системы на набор сервисов, препятствия на этом пути и то, как их преодолеть. Сначала рассмотрим концепцию архитектуры программного обеспечения.

2.1. Что представляет собой микросервисная архитектура

В главе 1 говорилось, что ключевой идеей микросервисной архитектуры является функциональная декомпозиция. Вместо разработки одного большого приложения вы создаете набор сервисов. С одной стороны, описание микросервисной архитектуры в виде некой функциональной декомпозиции довольно практично. Но с другой — это оставляет без ответа несколько вопросов. Например, какое отношение этот подход имеет к более общему понятию архитектуры программного обеспечения? Что такое сервис и насколько важен его размер?

Чтобы ответить на них, нужно сделать шаг назад и подумать о том, что мы понимаем под *архитектурой программного обеспечения*. Архитектура приложения — это его общая структура, состоящая из отдельных частей и зависимостей между ними. Как вы увидите в этом разделе, данное понятие является многоуровневым и описать его можно с разных сторон. Архитектура имеет большое значение, потому что она определяет качественные атрибуты приложения, или его «*-ости*». Традиционно архитектура сосредоточена на таких аспектах, как масштабируемость, надежность и безопасность. Но в наши дни важным качеством является также возможность быстрой и безопасной доставки кода. Как вы вскоре увидите, микросервисная архитектура — это стиль проектирования, который делает приложение легко подерживаемым, тестируемым и развертываемым.

Я начну этот раздел с описания концепции *архитектуры программного обеспечения* и того, почему она так важна. Затем мы обсудим идею архитектурного стиля. В конце я дам определение микросервисной архитектуры как стиля проектирования.

2.1.1. Что такое архитектура программного обеспечения и почему она важна

Архитектура, несомненно, имеет значение. Этой теме посвящены как минимум две конференции: O'Reilly Software Architecture Conference (conferences.oreilly.com/software-architecture) и SATURN (resources.sei.cmu.edu/news-events/events/saturn/). Многие разработчики стремятся стать архитекторами. Но что такое архитектура и почему она важна?

Чтобы ответить на этот вопрос, сначала определимся с тем, что мы понимаем под *архитектурой программного обеспечения*. Затем я покажу, что архитектура прило-

жения является многомерной и что лучше всего ее описывать в виде набора представлений или блок-схем. Далее вы узнаете, что важность программной архитектуры связана с ее влиянием на качественные атрибуты приложения.

Определение архитектуры программного обеспечения

Архитектура программного обеспечения имеет бесчисленное количество определений. Некоторые из них можно найти на странице https://en.wikiquote.org/wiki/Software_architecture. Мое любимое определение принадлежит Лену Бассу (Len Bass) и его коллегам по Институту программной инженерии (www.sei.cmu.edu), которые сыграли ключевую роль в становлении этой области в качестве отдельной дисциплины. Они определяют архитектуру программного обеспечения следующим образом: *программная архитектура вычислительной системы — это набор структур, необходимых для ее обсуждения и состоящих из программных элементов, связей между ними и свойств, присущих этим элементам и связям* (Bass L. et al. Documenting Software Architectures).

Это довольно абстрактное определение. Но его суть в том, что архитектура приложения — это его декомпозиция на части (элементы) и связи между ними. Декомпозиция важна по нескольким причинам.

- ❑ Она способствует разделению труда и знаний, так как позволяет нескольким людям или командам с потенциально узкоспециализированными знаниями продуктивно работать над одним приложением.
- ❑ Она определяет то, как взаимодействуют между собой программные элементы.

Разбиение на части и отношения между этими частями определяют качественные характеристики приложения.

Модель представлений архитектуры вида 4 + 1

Говоря конкретней, архитектуру приложения можно рассматривать с разных сторон, точно так же как архитектуру здания можно оценивать с точки зрения конструкции, водопровода, электропроводки и т. д. Филлип Кратчен (Phillip Krutchen) написал классический документ, посвященный модели представлений программной архитектуры вида 4 + 1, — *Architectural Blueprints — The “4 + 1” View Model of Software Architecture* (www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf). Модель 4 + 1 (рис. 2.1) определяет четыре разных представления архитектуры программного обеспечения. Каждое из них описывает определенный аспект архитектуры и состоит из конкретного набора программных элементов и связей между ними.

Эти представления имеют следующие цели.

- ❑ *Логическое представление* — программные элементы, создаваемые разработчиками. В объектно-ориентированных языках это классы и пакеты. Связи между ними соответствуют отношениям между классами и пакетами, включая наследование, взаимосвязи и зависимости.



Рис. 2.1. Модель представлений вида 4 + 1 описывает архитектуру приложения с помощью четырех представлений и сценариев, которые показывают, как элементы внутри каждого представления взаимодействуют для обработки запросов

- ❑ *Представление реализации* — результат работы системы сборки. Это представление состоит из модулей, представляющих упакованный код, и компонентов, которые являются исполняемыми или развертываемыми единицами и содержат один или несколько модулей. В Java модуль имеет формат JAR, а компонентом обычно выступает WAR- или исполняемый JAR-файл. Связь между ними определяется зависимостями между модулями и тем, какие компоненты объединены в тот или иной модуль.
- ❑ *Представление процесса* — компоненты на этапе выполнения. Каждый элемент является процессом, а отношения между процессами представляют межпроцессное взаимодействие.
- ❑ *Развертывание* — то, как процессы распределяются по устройствам. Элементы в этом представлении состоят из серверов (физических или виртуальных) и процессов. Связи между серверами представляют сеть. Это представление также описывает отношение между процессами и устройствами.

Вдобавок к этим четырем представлениям существуют сценарии (+ 1 в модели 4 + 1), которые их оживляют. Каждый сценарий описывает, как различные архитектурные компоненты внутри конкретного представления взаимодействуют между собой, чтобы обработать запрос. Например, сценарий в логическом представлении

демонстрирует взаимодействие классов. Аналогично сценарий в представлении процесса показывает совместную работу процессов.

Модель представлений вида $4 + 1$ — это отличный способ описания архитектуры приложения. Каждое представление иллюстрирует важный аспект архитектуры, а сценарии показывают, как взаимодействуют представления. Теперь посмотрим, почему архитектура так важна.

Почему архитектура имеет значение

Требования к приложению делятся на две категории. Первая включает в себя *функциональные* требования, определяющие назначение кода. Обычно они представлены в виде сценариев использования или пользовательских историй. Архитектура не играет здесь большой роли. Функциональные требования можно реализовать с помощью почти любой архитектуры, даже самой плохой.

Архитектура выходит на первый план, когда речь заходит о второй категории требований, связанной с *качеством обслуживания*. Это так называемые *качественные атрибуты*. Они определяют такие свойства времени выполнения, как масштабируемость и надежность. А также описывают аспекты разработки, такие как простота в обслуживании, тестировании и развертывании. От выбранной вами архитектуры зависит, насколько приложение отвечает этим требованиям к качеству.

2.1.2. Обзор архитектурных стилей

В реальном мире архитектура здания относится к определенному стилю: викторианскому, ар-деко и т. д. Каждый стиль — это набор проектировочных решений, определяющих отличительные признаки здания и строительные материалы. Эту же концепцию можно применить к программному обеспечению. Дэвид Гарлан (David Garlan) и Мэри Шой (Mary Shaw) (An Introduction to Software Architecture, January 1994, www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf), пионеры в области программной архитектуры, дают архитектурному стилю следующее определение: *архитектурный стиль определяет семейство подобных систем с точки зрения структурной организации. В частности, стиль определяет набор компонентов и коннекторов, которые можно применять в реализациях этого стиля, а также ряд правил, согласно которым они могут сочетаться.*

Конкретный архитектурный стиль предоставляет ограниченную палитру элементов (компонентов) и связей (коннекторов), на основе которых вы можете описать представление архитектуры своего приложения. На практике обычно используется сочетание архитектурных стилей. Например, позже в этой главе вы увидите, что монолитная архитектура — это стиль, который структурирует представление реализации в виде единого (исполняемого/развертываемого) компонента. Микросервисная архитектура структурирует приложение в виде набора слабо связанных сервисов.

Многоуровневый архитектурный стиль

Классическим примером архитектурного стиля является *многоуровневая архитектура*. Она распределяет программные элементы по разным уровням. Каждый уровень имеет четко обозначенный набор обязанностей. Также ограничиваются зависимости между уровнями. Уровень может зависеть либо от уровня, находящегося непосредственно под ним (строгое разбиение), либо от любого нижележащего уровня.

Многоуровневую архитектуру можно применить к любому из рассмотренных ранее представлений, трехуровневую архитектуру (ее частный случай) — к логическому представлению. Она разделяет классы приложения на следующие уровни или слои:

- ❑ *уровень представления* — содержит код, реализующий пользовательский интерфейс или внешние API;
- ❑ *уровень бизнес-логики* — содержит бизнес-логику;
- ❑ *уровень хранения данных* — реализует логику взаимодействия с базой данных.

Многоуровневая архитектура — отличный пример архитектурного стиля, но у нее есть некоторые существенные недостатки.

- ❑ *Единый уровень представления* — не учитывает того, что приложение, скорее всего, будет вызываться более чем одной системой.
- ❑ *Единый уровень хранения данных* — не учитывает того, что приложение, скорее всего, будет взаимодействовать более чем с одной базой данных.
- ❑ *Уровень бизнес-логики зависит от уровня хранения данных* — теоретически эта зависимость не позволяет тестировать бизнес-логику отдельно от базы данных.

Кроме того, многоуровневая архитектура искажает зависимости в хорошо спроектированном приложении. Бизнес-логика обычно предусматривает интерфейс или репозиторий интерфейсов, которые определяют методы доступа к данным. Уровень хранения данных определяет классы DAO, которые реализуют интерфейсы репозитория. Иными словами, зависимости являются обратными относительно того, что описывает многоуровневая архитектура.

Рассмотрим альтернативный подход, который позволяет преодолеть эти недостатки, — *шестигранную архитектуру*.

О шестигранном архитектурном стиле

Шестигранная архитектура — это альтернатива многоуровневому стилю проектирования. Она организует логическое представление таким образом, что бизнес-логика оказывается в центре (рис. 2.2). Вместо уровня представления у приложения есть один или несколько *входящих адаптеров*, которые обрабатывают внешние запросы путем вызова бизнес-логики. Аналогично вместо уровня хранения данных используются один или несколько *исходящих адаптеров*, которые вызываются бизнес-логикой и обращаются к внешним приложениям. Ключевой характеристикой и преимуществом данной архитектуры является то, что бизнес-логика не зависит от адаптеров. Все наоборот: адаптеры зависят от нее.

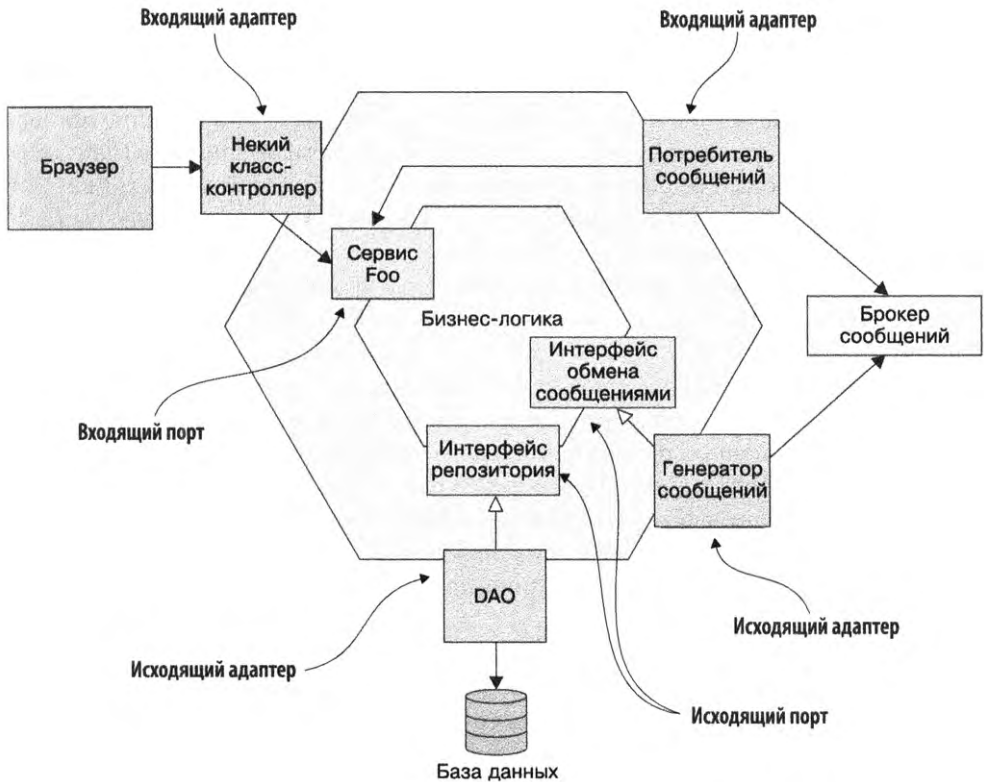


Рис. 2.2. Пример шестигранной архитектуры, состоящей из бизнес-логики и одного или нескольких адаптеров, которые взаимодействуют с внешними системами. Бизнес-логика содержит один или несколько портов. Входящие адаптеры, обрабатывающие запросы от внешних систем, обращаются к входящему порту. Исходящий адаптер реализует исходящий порт и обращается к внешней системе

У бизнес-логики есть один или несколько портов. *Порт* определяет набор операций и то, как и в чем бизнес-логика взаимодействует с внешним кодом. В Java, например, порт часто является Java-интерфейсом. Существует два вида портов: входящие и исходящие. Входящий порт — это API, выставляемый наружу бизнес-логикой и доступный для вызова внешними приложениями. В качестве примера входящего порта можно привести интерфейс сервиса, который описывает его публичные методы. Исходящий порт — это то, как бизнес-логика обращается к внешним системам. Примером может служить интерфейс репозитория, определяющий набор операций для доступа к данным.

Вокруг бизнес-логики размещаются адаптеры. Как и порты, они бывают двух типов: входящие и исходящие. Входящий адаптер обрабатывает запросы из внешнего мира, обращаясь к входящему порту. Примером входящего адаптера может служить контроллер Spring MVC, который реализует либо набор конечных точек формата REST, либо коллекцию веб-страниц. Еще один пример — клиентский брокер

сообщений, который на них подписывается. Несколько входящих адаптеров могут обращаться к одному и тому же входящему порту.

Исходящий адаптер реализует исходящий порт и обрабатывает запросы бизнес-логики, обращаясь к внешнему приложению или сервису. В качестве примера исходящего адаптера можно привести класс *объекта доступа к данным* (data access object, DAO), который реализует операции для работы с базой данных. Еще один пример — класс прокси, вызывающий внешний сервис. Исходящие адаптеры также могут публиковать события.

Важное преимущество шестигранного архитектурного стиля состоит в том, что его адаптеры отделяют бизнес-логику от логики представления и доступа к данным и делают ее независимой.

Благодаря этому изолированное тестирование бизнес-логики намного упрощается. Еще одна сильная сторона этого стиля связана с тем, что он более точно отражает архитектуру современных приложений. Бизнес-логику можно вызывать с помощью разных адаптеров, каждый из которых реализует определенный программный или пользовательский интерфейс. Сама бизнес-логика тоже может обратиться к одному из нескольких адаптеров, вызывающих определенную внешнюю систему. Шестигранный стиль отлично подходит для описания каждого сервиса в микросервисной архитектуре.

Многоуровневая и шестигранная архитектуры — это примеры архитектурных стилей. Каждая из них определяет составляющие приложения и ограничивает отношения между ними. Шестигранная и многоуровневая (в виде трехуровневой) архитектуры организуют логическое представление. Теперь определим микросервисы как архитектурный стиль, который описывает представление реализации.

2.1.3. Микросервисная архитектура как архитектурный стиль

Мы уже обсудили архитектурные стили и модель представлений вида $4 + 1$. Теперь можно дать определение монолитной и микросервисной архитектурам. Обе они являются архитектурными стилями. Монолитная архитектура структурирует представление реализации в виде единого компонента — исполняемого или WAR-файла. Это определение оставляет без внимания другие представления. Монолитное приложение, к примеру, может иметь логическое представление, организованное по принципу шестигранной архитектуры.

Шаблон «Монолитная архитектура»

Структурирует приложение в виде единого исполняемого/развертываемого компонента. См. microservices.io/patterns/monolithic.html.

Микросервисная архитектура тоже является архитектурным стилем. Она структурирует представление реализации в виде набора компонентов — исполняемых или WAR-файлов. Компоненты представлены сервисами, а в качестве коннекторов служат коммуникационные протоколы, которые позволяют этим сервисам взаимодействовать между собой. Каждый сервис имеет собственную архитектуру логического представления (обычно это шестигранная архитектура). На рис. 2.3 показан пример микросервисной архитектуры для приложения FTGO; сервисы соответствуют бизнес-функциям, таким как управление заказами или ресторанами.

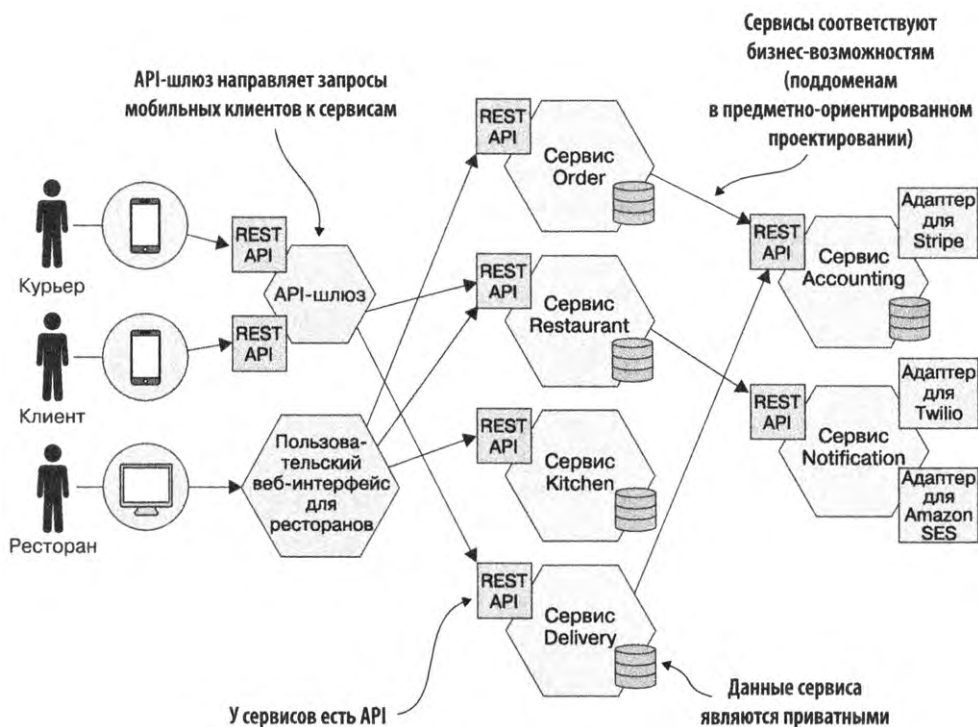


Рис. 2.3. Потенциальная микросервисная архитектура для приложения FTGO, состоящая из множества сервисов

Шаблон «Микросервисная архитектура»

Структурирует приложение в виде набора слабо связанных сервисов, которые развешиваются независимо друг от друга. См. microservices.io/patterns/microservices.html.

Позже в этой главе я объясню, что я имею в виду под *бизнес-функциями*. Коннекторы между сервисами реализуются с помощью механизма межпроцессного

взаимодействия, такого как REST API или асинхронный обмен сообщениями. В главе 3 мы обсудим межпроцессное взаимодействие более подробно.

Ключевое ограничение, которое накладывает микросервисная архитектура, — слабая связанность сервисов. Следовательно, сервисы ограничены в том, как они между собой взаимодействуют. Чтобы лучше это объяснить, попытаюсь дать определение терминам «сервис» и «слабая связанность» и расскажу, почему это важно.

Что такое сервис

Сервис — это автономный, независимо разворачиваемый программный компонент, который реализует определенные полезные функции. На рис. 2.4 показано внешнее представление сервиса (в данном случае *Order*). У него есть API, через который сервис предоставляет доступ к своим функциям. Существует два вида операций: команды и запросы. API состоит из команд, запросов и событий. Команда, такая как `createOrder()`, выполняет действия и обновляет данные. Запрос, такой как `findOrderByid()`, извлекает данные. Сервис также публикует события, например `OrderCreated`, которые потребляются его клиентами.

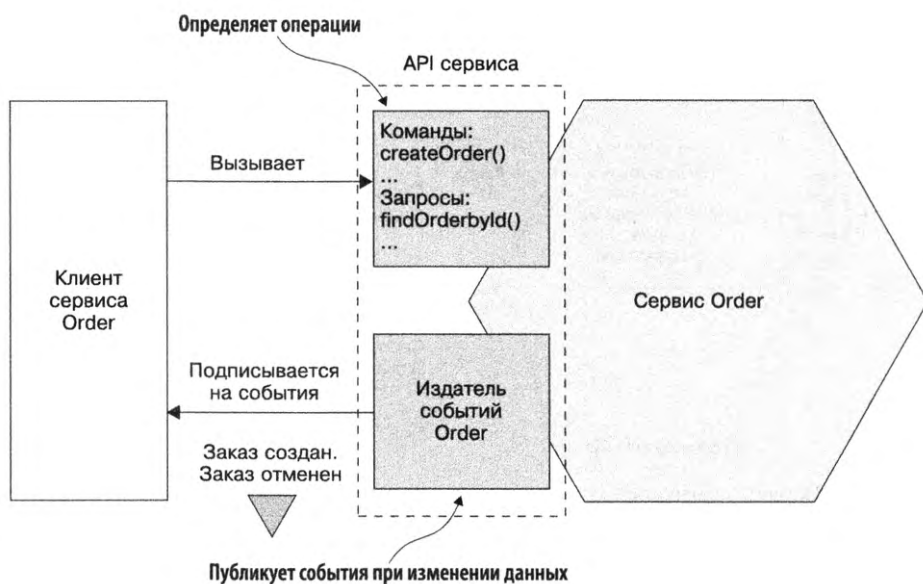


Рис. 2.4. Сервис обладает API, который инкапсулирует реализацию. API определяет операции, вызываемые клиентами. Существует два типа операций: команды (обновляют данные) и запросы (извлекают данные). При изменении своих данных сервис публикует события, на которые могут подписаться его клиенты

API сервиса инкапсулирует его внутреннюю реализацию. В отличие от монолита этот подход не позволяет разработчику писать код, минуя API. Благодаря этому микросервисная архитектура обеспечивает модульность приложения.

Каждый микросервис обладает собственной архитектурой и иногда отдельным стеком технологий. Но обычно сервисы имеют шестигранную архитектуру. Их API реализуются адаптерами, которые взаимодействуют с бизнес-логикой приложения. Бизнес-логика вызывается адаптером операций, а события, которые она генерирует, публикуются адаптером событий.

В главе 12 при обсуждении технологий развертывания вы увидите, что представление реализации сервиса способно принимать множество форм. Компонент может быть автономным процессом, веб-приложением/OSGI-пакетом, запущенным в контейнере, или бессерверной облачной функцией. Однако сервис должен иметь API и развертываться независимо — это основное требование.

Что такое слабая связанность

Важной характеристикой микросервисной архитектуры является слабое связывание сервисов (en.wikipedia.org/wiki/Loose_coupling). Все взаимодействие с сервисом происходит через API, инкапсулирующий подробности его реализации. Это позволяет изменять внутреннее содержание сервиса, не затрагивая его клиентов. Слабо связанные сервисы — это ключ к улучшению скорости разработки приложений, в том числе их поддержки и тестирования. Их намного проще изменять и тестировать, в них проще разобраться.

Требование, согласно которому сервисы должны быть слабо связанными и взаимодействовать только через API, исключает коммуникацию через базу данных. Постоянные данные сервиса должны восприниматься как поля класса и оставаться приватными. Если разработчик изменит структуру базы данных, ему не нужно будет тратить время на согласование с коллегами, которые работают над другими сервисами. Отсутствие общих таблиц БД также улучшает изоляцию на этапе выполнения. Это, например, гарантирует, что сервису не придется ждать из-за того, что другой сервис заблокировал базу данных. Тем не менее позже вы узнаете, что у этого подхода есть и недостатки — например, это усложняет поддержание согласованности данных и выполнение запросов к нескольким сервисам

Роль общих библиотек

Разработчики часто упаковывают функции в библиотеки (модули), чтобы их можно было использовать в нескольких приложениях без дублирования кода. В конце концов, что бы мы сейчас делали без репозиторий Maven или npm? У вас также может возникнуть соблазн задействовать разделяемые библиотеки в микросервисной архитектуре. На первый взгляд это выглядит хорошим решением для того, чтобы избежать дублирования кода. Но нужно убедиться в том, что это не приведет к связыванию ваших сервисов.

Представьте, к примеру, что нескольким сервисам нужно обновить бизнес-объект `Order`. Вы можете упаковать эту функцию в библиотеку, которую станут использовать разные сервисы. С одной стороны, это устраняет дублирующийся код, но с другой — что произойдет, если требования изменятся и это повлияет на бизнес-объект

Order. Вам пришлось бы одновременно пересобрать и заново развернуть все эти сервисы. Вместо этого функции, которые с большой вероятностью в дальнейшем будут меняться, можно оформить в виде отдельного сервиса, что намного лучше.

Старайтесь применять библиотеки для функций, изменение которых маловероятно. Например, в типичном приложении было бы излишним реализовывать класс *Money* в каждом сервисе. Вместо этого стоит создать библиотеку, с которой будут работать эти сервисы.

Размер сервиса обычно не имеет значения

Одна из проблем термина «*микросервис*» — то, что в глаза сразу бросается «*микро*». Это подразумевает, что сервис должен быть очень маленьким. То же самое относится и к другим терминам, основанным на размерах, таким как *мини-сервис* и *наносервис*. В реальности размер не является полезной характеристикой.

Определение хорошо спроектированного сервиса лучше связать с возможностью разрабатывать его в небольшой команде, как можно быстрее и минимально взаимодействуя с другими командами. Теоретически команда должна отвечать только за один сервис, чтобы тот и в самом деле был *микроскопическим*. Если же сервис требует большой команды разработчиков или слишком много времени на тестирование, его, как и саму команду, имеет смысл разделить на части. Если же вам постоянно приходится менять свой сервис из-за изменений в других сервисах, это признак недостаточно слабой связанности. Возможно, у вас даже получился распределенный монолит.

Микросервисная архитектура структурирует приложение в виде небольших слабо связанных сервисов. Это улучшает временные показатели разработки (поддерживаемость, тестируемость, развертываемость и т. д.) и позволяет организации создавать лучшее программное обеспечение в более короткие сроки. А еще положительно сказывается на масштабируемости, хотя это и не главная цель. Чтобы разработать микросервисную архитектуру для своего приложения, вы должны обозначить его сервисы и определить, как они будут взаимодействовать. Посмотрим, как это делается.

2.2. Определение микросервисной архитектуры приложения

Как определить микросервисную архитектуру? Как и для любого другого аспекта разработки, все начинается с формализованных требований. При этом желательно иметь специалистов в данной проблемной области и, возможно, существующее приложение. В сфере программного обеспечения определение архитектуры часто ближе к искусству, чем к науке. В данном разделе этот процесс описан в виде трех шагов (рис. 2.5). Однако необходимо помнить, что это вовсе не инструкция, которую следует выполнять буквально. Решить эту задачу, скорее всего, можно будет постепенно, подключив находчивость.

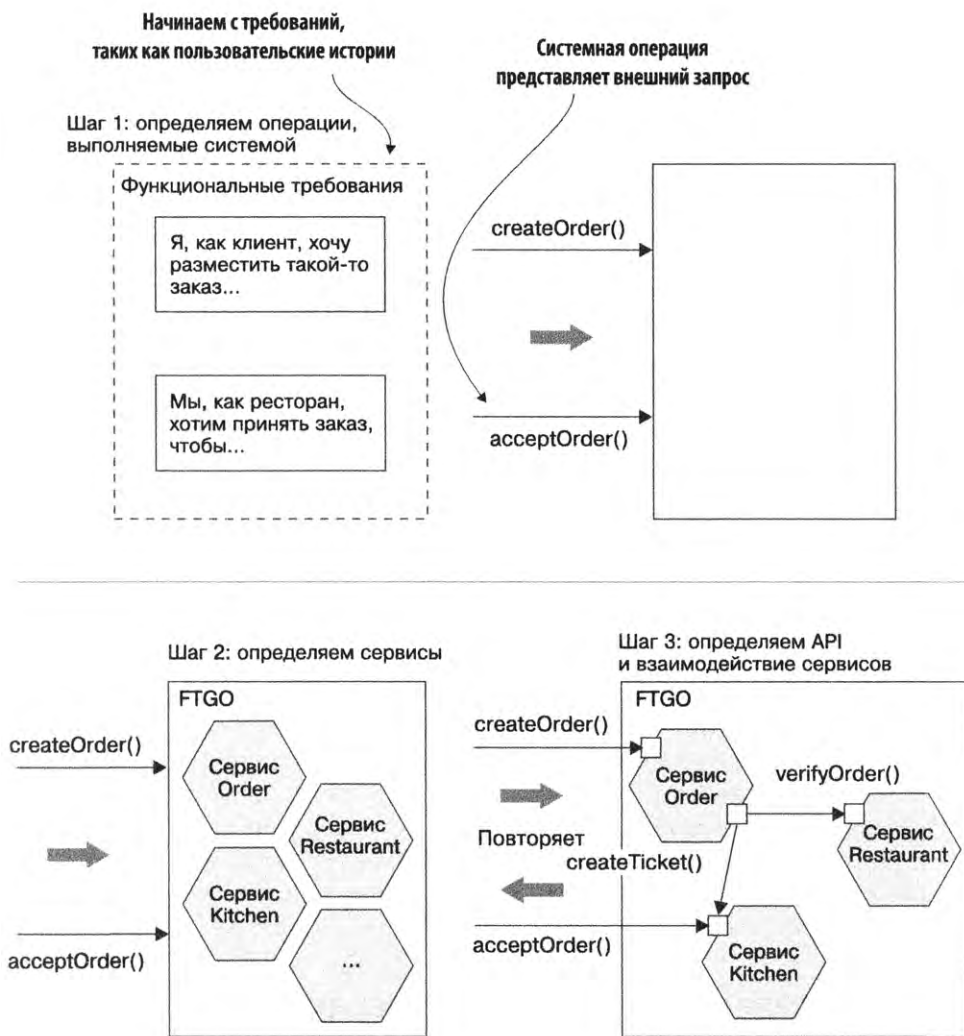


Рис. 2.5. Трехшаговый процесс описания микросервисной архитектуры приложения

Приложение существует, чтобы обрабатывать запросы. Поэтому первым шагом в определении архитектуры станет формирование ключевых запросов на основе требований к приложению. Но вместо того, чтобы описывать запросы в виде конкретных технологий межпроцессного взаимодействия, я использую более абстрактное понятие системной операции. *Системная операция* представляет собой запрос, который приложение должно обработать. Поведение каждой команды определяется в виде абстрактной доменной модели, которая тоже извлекается из требований. Системные операции становятся архитектурными сценариями, иллюстрирующими взаимодействие сервисов.

Вторым шагом в этом процессе будет разбиение на сервисы. В вашем распоряжении несколько стратегий. Одна из них берет начало в сфере бизнес-архитектуры и заключается в том, что сервисы должны соответствовать бизнес-функциям. Другая предусматривает организацию сервисов вокруг подобластей в контексте предметно-ориентированного проектирования. В итоге сервисы будут основаны на бизнес-концепциях, а не на технических аспектах.

Третий шаг в определении архитектуры приложения заключается в описании API для каждого сервиса. Для этого сервисам назначаются все системные операции, определенные на первом шаге. Операцию можно реализовать в виде одного или нескольких сервисов. В последнем случае нужно решить, как они будут взаимодействовать между собой, что обычно требует поддержки дополнительных операций с их стороны. Вам также нужно выбрать один из механизмов IPC, описываемых в главе 3, чтобы реализовать API каждого из сервисов.

Декомпозиция связана с несколькими трудностями. Во-первых, возникают сетевые задержки. Вы можете обнаружить, что определенная схема разбиения непрактична из-за слишком частого обмена данными между сервисами. Во-вторых, синхронное взаимодействие снижает доступность. Вам, возможно, придется прибегнуть к концепции автономных сервисов, которая описывается в главе 3. Третьей проблемой становится необходимость поддержания согласованности данных между сервисами. Для этого обычно используются повествования, рассматриваемые в главе 4. Последнее препятствие на пути к декомпозиции связано с так называемыми божественными классами, применяемыми в разных частях приложения. К счастью, для их устранения можно воспользоваться концепциями из предметно-ориентированного проектирования.

Вначале в этом разделе описывается, как определить операции приложения. После этого рассмотрим стратегии и методические рекомендации по разбиению приложения на сервисы, а также трудности, которые при этом могут возникнуть, и способы их преодоления. В конце я покажу, как описать API для каждого сервиса.

2.2.1. Определение системных операций

Первый шаг при проектировании архитектуры приложения — определение системных операций. За отправную точку берутся требования к приложению, включая пользовательские истории и связанные с ними сценарии использования (стоит отметить, что они отличаются от архитектурных сценариев). Процесс идентификации и определения системных операций состоит из двух шагов (рис. 2.6). Он навеян процессом объектно-ориентированного программирования, описанным в книге Крейга Лармана (Craig Larman) *Applying UML and Patterns* (Prentice Hall, 2004)¹ (см. подробности на www.craigharman.com/wiki/index.php?title=Book_Applying_UML_and_Patterns). На первом шаге создается обобщенная доменная модель, состоящая из ключевых

¹ Ларман К. Применение UML 2.0 и шаблонов проектирования. — М.: Вильямс, 2016.

классов, которые предоставляют словарь для описания системных операций. Сами системные операции, а также их поведение с точки зрения доменной модели описываются на втором шаге.



Рис. 2.6. Системные операции определяются на основе требований к приложению в ходе двухшагового процесса. На первом шаге создается обобщенная доменная модель. На втором шаге в рамках этой модели определяются системные операции

Доменная модель составляется в основном из имен существительных, взятых из пользовательских историй, а системные операции — в основном из глаголов. Доменную модель можно определить также с помощью методики «Событийный штурм», о которой поговорим в главе 5. Поведение каждой системной операции описывается с точки зрения ее влияния на один или несколько доменных объектов и отношений между ними. Системная операция может создавать, обновлять или удалять доменные объекты, а также устанавливать или разрушать их связи.

Рассмотрим процесс определения обобщенной доменной модели. Затем на ее основе я опишу системные операции.

Создание обобщенной доменной модели

Первый шаг на пути определения системных операций — очерчивание обобщенной доменной модели приложения. Имейте в виду, что эта модель намного проще того, что будет реализовано в итоге. У нашего приложения даже не будет единой доменной модели, поскольку, как вы вскоре узнаете, каждый сервис имеет свою собственную. Несмотря на чрезмерную упрощенность, на этой стадии обобщенная доменная модель будет полезна, ведь она определяет словарь для описания поведения системных операций.

Доменная модель создается с помощью стандартных методик, таких как анализ имен существительных в пользовательских историях и консультация с экспертами

в данной проблемной области. Возьмем, к примеру, историю размещения заказа. Мы можем развернуть ее во множество сценариев использования, включая следующий:

- Дано: клиент
 - И ресторан
 - И адрес/время доставки из этого ресторана
 - И суммарная цена заказа, отвечающая среднему показателю ресторана
- Когда клиент размещает заказ
- Тогда банковская карта клиента авторизуется
 - И создается заказ в состоянии PENDING_ACCEPTANCE
 - И заказ привязывается к клиенту
 - И заказ привязывается к ресторану

Имена существительные в этом пользовательском сценарии указывают на существование различных классов, включая Consumer (клиент), Order (заказ), Restaurant (ресторан) и CreditCard (банковская карта).

Точно так же историю приема заказа можно развернуть в сценарий наподобие следующего:

- Дано: заказ в состоянии PENDING_ACCEPTANCE
 - И курьер, доступный для доставки заказа
- Когда ресторан принимает заказ с обязательством приготовить еду к заданному времени
- Тогда состояние заказа меняется на ACCEPTED
 - И полю заказа promiseByTime назначается заданное время
 - И курьер назначается для доставки заказа

Этот сценарий подразумевает существование классов Courier (курьер) и Delivery (доставка). После нескольких этапов анализа итоговым результатом будет доменная модель, состоящая, как и ожидалось, из этих и других классов, таких как MenuItem (пункт меню) и Address (адрес). Схема с ключевыми классами показана на рис. 2.7.

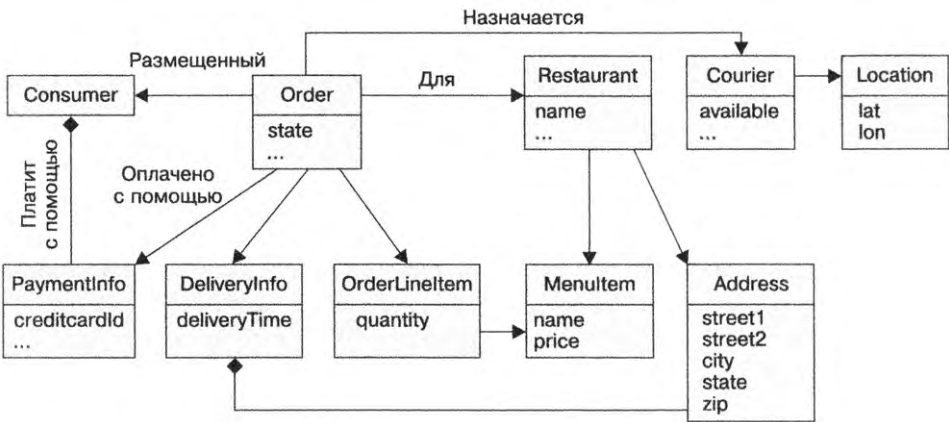


Рис. 2.7. Ключевые классы доменной модели FTGO

Классы имеют следующие обязанности:

- ❑ **Consumer** — клиент, размещающий заказ;
- ❑ **Order** — заказ, размещенный клиентом;
- ❑ **OrderLineItem** — отдельная позиция в заказе;
- ❑ **DeliveryInfo** — время и место доставки заказа;
- ❑ **Restaurant** — ресторан, готовящий заказы для доставки клиентам;
- ❑ **MenuItem** — пункт в меню ресторана;
- ❑ **Courier** — курьер, который доставляет клиентам заказы (отслеживает доступность курьеров и их местоположение);
- ❑ **Address** — адреса клиента и ресторана;
- ❑ **Location** — широта и долгота местонахождения курьера.

Схема классов, приведенная на рис. 2.7, иллюстрирует один из аспектов архитектуры приложения. Но без сценариев, которые ее оживляют, это не более чем красивая картинка. Следующим шагом будет определение системных операций, соответствующих архитектурным сценариям.

Определение системных операций

Следующий шаг после описания обобщенной доменной модели — определение запросов, которые приложение должно обрабатывать. Детали пользовательского интерфейса выходят за рамки этой книги, но, как вы можете представить, в каждом сценарии использования графический интерфейс извлекает и обновляет данные путем выполнения запросов к серверной бизнес-логике. FTGO в основном представляет собой веб-приложение. Это означает, что большинство запросов основаны на HTTP, хотя вполне вероятно, что некоторые клиенты будут применять механизм обмена сообщениями. Таким образом, вместо привязки к конкретному протоколу для представления запросов лучше использовать более абстрактное понятие системной операции.

Системные операции бывают двух видов:

- ❑ *команды* — системные операции для создания, обновления и удаления данных;
- ❑ *запросы* — системные операции для чтения (запрашивания) данных.

Хорошей отправной точкой для определения системных команд будет анализ глаголов в пользовательских историях и сценариях. Возьмем, к примеру, историю размещения заказа. Она явно указывает на то, что система должна предоставлять операцию **Create Order** (создать заказ). Многие другие истории напрямую соответствуют отдельным системным командам. Некоторые ключевые системные команды перечислены в табл. 2.1.

Таблица 2.1. Ключевые системные команды для приложения FTGO

Действующее лицо	История	Команда	Описание
Клиент	Create Order	createOrder()	Создает заказ
Ресторан	Accept Order	acceptOrder()	Указывает на то, что ресторан принял заказ и обязуется приготовить его к заданному времени
	Order Ready for Pickup	noteOrderReadyForPickup()	Указывает на то, что заказ готов к доставке
Курьер	Update Location	noteUpdatedLocation()	Обновляет текущее местоположение курьера
	Delivery picked up	noteDeliveryPickedUp()	Указывает на то, что курьер взял заказ
	Delivery delivered	noteDeliveryDelivered()	Указывает на то, что курьер доставил заказ

У команды есть спецификация, которая определяет ее параметры, возвращаемое значение и поведение в рамках классов доменной модели. Описание поведения состоит из условий двух видов: предварительных и окончательных. Первые должны выполняться в момент вызова операции, а вторые — после. Далее показан пример спецификации для системной операции `createOrder()`.

Операция	<code>createOrder</code> (ID клиента, способ оплаты, адрес доставки, время доставки, ID ресторана, позиции заказа)
Возвращает	<code>orderId...</code>
Предварительные условия	Клиент существует и может размещать заказы. Позиции заказа соответствуют пунктам меню ресторана. Адрес и время доставки выполнимы для ресторана
Окончательные условия	Банковская карта клиента позволила снять сумму заказа. Заказ был создан в состоянии <code>PENDING_ACCEPTANCE</code>

Предварительные условия отражают участок «Дано» в сценарии размещения заказа, приведенном ранее. Окончательные условия отражают участок «Тогда». При вызове системная операция проверяет предварительные условия и производит действия, необходимые для выполнения окончательных условий.

Далее показана спецификация системной операции `acceptOrder()`.

Операция	<code>acceptOrder</code> (<code>restaurantId</code> , <code>orderId</code> , <code>readyByTime</code>)
Возвращает	—
Предварительные условия	<code>order.status</code> равно <code>PENDING_ACCEPTANCE</code> . Курьер доступен для доставки заказа
Окончательные условия	Состояние <code>order.status</code> поменялось на <code>ACCEPTED</code> . Время <code>order.readyByTime</code> поменялось на <code>readyByTime</code> . Курьер назначен для доставки заказа

Ее предварительные и окончательные условия отражают сценарий использования, приведенный ранее.

Большинство операций, важных с архитектурной точки зрения, являются командами. Запросы, извлекающие данные, тоже иногда имеют значение.

Помимо команд, приложение должно реализовать и запросы. Они наполняют пользовательский интерфейс информацией, необходимой для принятия решений. На этом этапе мы еще не придумали, каким будет пользовательский интерфейс приложения FTGO, но взгляните, к примеру, на процесс размещения заказа клиентом.

- ❑ Пользователь вводит адрес и время доставки.
- ❑ Система выводит доступные рестораны.
- ❑ Пользователь выбирает ресторан.
- ❑ Система выводит меню.
- ❑ Пользователь выбирает пункт меню и оплачивает счет.
- ❑ Система создает заказ.

Сценарий использования подразумевает следующие запросы.

- ❑ `findAvailableRestaurants(deliveryAddress, deliveryTime)` — извлекает рестораны, которые могут выполнить доставку по заданному адресу в заданное время.
- ❑ `findRestaurantMenu(id)` — извлекает информацию о ресторане, включая блюда в меню.

Из этих двух запросов `findAvailableRestaurants()`, наверное, имеет наибольшее архитектурное значение и применяет поиск по местности. Поисковая составляющая запроса возвращает все точки (рестораны), находящиеся неподалеку от адреса доставки. Она также отфильтровывает все рестораны, которые будут закрыты в период подготовки и отправки заказа. Здесь крайне важна производительность, так как этот запрос выполняется при размещении каждого заказа.

Обобщенная доменная модель и системные операции описывают работу приложения и помогают определить его архитектуру. Поведение каждой системной операции описывается в рамках доменной модели. Каждая важная системная операция соответствует сценарию, который является важной частью описания архитектуры.

Следующим шагом после определения системных операций будет обозначение сервисов приложения. Как упоминалось ранее, для этого не предусмотрено четких инструкций, которым просто нужно следовать. Однако мы можем воспользоваться различными стратегиями декомпозиции, каждая из которых подходит к проблеме с определенной стороны и использует собственную терминологию. Но какую бы стратегию вы ни выбрали, результат будет один: архитектура, состоящая из сервисов, которые в основном организованы вокруг бизнес-аспектов, а не технических концепций.

Рассмотрим первую стратегию, которая описывает сервисы в соответствии с бизнес-возможностями.

2.2.2. Разбиение на сервисы по бизнес-возможностям

Одной из стратегий создания микросервисной архитектуры является разбиение по бизнес-возможностям. Концепция «*бизнес-возможности*» применяется в моделировании бизнес-архитектур и обозначает то, из чего бизнес генерирует прибыль. Набор возможностей для конкретной компании зависит от того, чем именно она занимается. Например, в число возможностей страховой компании обычно входят андеррайтинг¹, обработка претензий, биллинг, выполнение правовых норм и т. д. Возможности интернет-магазина включают управление заказами, инвентаризацию, отправку товара и пр.

Шаблон «Разбиение по бизнес-возможностям»

Определяет сервисы, соответствующие бизнес-возможностям. См. microservices.io/patterns/decomposition/decompose-by-business-capability.html.

Бизнес-возможности определяют то, чем занимается организация

Бизнес-возможности организации описывают то, *чем* она является. Обычно они стабильны, в отличие от того, *как* организация ведет свой бизнес (этот аспект со временем меняется, иногда до неузнаваемости). Это особенно актуально в наши дни, когда технологии все чаще используются для автоматизации бизнес-процессов. Например, еще совсем недавно для того, чтобы положить сумму с чека на счет, нужно было передать его кассиру в банке. Затем стало возможно сделать это через банкоматы, теперь же в большинстве случаев — с помощью смартфона. Как видите, бизнес-возможность «*депонирования чека*» осталась неизменной, но то, каким способом это делается, изменилось кардинально.

Определение бизнес-возможностей

Бизнес-возможности организации определяются путем анализа ее целей, структуры и бизнес-процессов. Каждую возможность можно представить в виде сервиса, но для этого она должна ориентироваться на бизнес, а не на технические аспекты. Ее спецификация состоит из различных компонентов, включая ввод, вывод и соглашения уровня сервиса. Например, в случае со страховым андеррайтингом вводом является заявление клиента, а вывод будет включать одобрение и цену.

Бизнес-возможность часто сосредоточена на определенном бизнес-объекте. Например, бизнес-объект «*претензия*» лежит в основе возможности «*обработка претензий*». Во многих случаях возможность можно разбить на подвозможности.

¹ Услуги, предоставляемые финансовыми учреждениями, такими как банки, страховые компании, которые гарантируют получение выплат в случае финансовых убытков (<https://ru.wikipedia.org/wiki/Андеррайтинг>). — Примеч. ред.

Например, обработка претензий состоит из обработки информации о претензии, ее рассмотрения и управления соответствующими выплатами.

Несложно себе представить бизнес-возможности приложения FTGO.

❑ Управление поставщиками:

- *управление курьерами* — управление информацией о курьерах;
- *управление информацией о ресторанах* — управление меню ресторана и другими данными, включая местоположение и график работы.

❑ Управление клиентами — управление информацией о клиентах.

❑ Прием и выполнение заказов:

- *управление заказами* — создание заказов и управление ими со стороны клиентов;
- *управление заказами в ресторане* — управление подготовкой заказов в ресторане;
- логистика;
- *управление доступностью курьеров* — управление готовностью курьеров доставить заказы в режиме реального времени;
- *управление доставкой* — доставка заказов клиентам.

❑ Бухучет:

- *отчетность по клиентам* — управление клиентскими платежами;
- *отчетность по ресторанам* — управление платежами, поступающими в рестораны;
- *отчетность по курьерам* — управление платой курьерам.

И так далее.

Возможности верхнего уровня включают в себя управление поставщиками, управление клиентами, принятие и выполнение заказов, бухучет. Скорее всего, этот список будет расширен за счет других возможностей, например связанных с маркетингом. Большинство из них разбиты на подвозможности. Например, пункт «Прием и выполнение заказов» состоит из пяти подпунктов.

Интересная особенность этой иерархии — наличие трех возможностей, относящихся к ресторанам: управление информацией о ресторанах, управление заказами в ресторане и отчетность по ресторанам. Это связано с тем, что данные возможности представляют собой три совершенно разных аспекта работы ресторанов.

Далее вы увидите, как с помощью бизнес-возможностей описать сервисы.

От бизнес-возможностей к сервисам

Определившись с бизнес-возможностями, вы должны описать сервисы для каждой из них или для групп связанных между собой возможностей. Схема соответствия между возможностями и сервисами приложения FTGO показана на рис. 2.8. Иногда сервисы создаются для возможностей верхнего уровня, таких как бухучет, а иногда — для подвозможностей.



Рис. 2.8. Связывание бизнес-возможностей FTGO с сервисами. Сервисам соответствуют возможности разных уровней иерархии

Решение о том, для возможностей какого уровня следует создавать сервисы, отчасти субъективно. Мое обоснование в этом конкретном случае выглядит так.

- ❑ Подвозможности управления поставщиками я связал с двумя сервисами, поскольку рестораны и курьеры — это совершенно разные виды поставщиков.
- ❑ Возможность приема и выполнения заказа соответствует трем сервисам, каждый из которых отвечает за отдельные стадии процесса. Я объединил возможности управления доступностью курьеров и доставкой и связал их с единым сервисом, так как они тесно переплетаются.
- ❑ Я выделил отдельный сервис для бухучета, поскольку разные виды отчетности выглядят похожими.

Позже, возможно, будет разумным разделить платежи (для ресторанов и курьеров) и биллинг (для клиентов).

Ключевое преимущество организации сервисов вокруг возможностей состоит в том, что из-за их стабильности итоговая архитектура тоже получается относительно стабильной. Отдельные компоненты могут эволюционировать вместе с подходами к ведению бизнеса, но сама архитектура останется неизменной.

Тем не менее имейте в виду, что сервисы, показанные на рис. 2.8, — лишь первая попытка описания архитектуры. Они могут меняться по мере более тесного знакомства с проблемной областью приложения. В частности, важный этап процесса проектирования связан с исследованием того, как взаимодействуют между собой ключевые сервисы. Например, вы можете обнаружить, что из-за излишнего межпроцессного взаимодействия какое-то разбиение оказывается неэффективным и некоторые сервисы лучше объединить. В то же время сервис может становиться все сложнее до тех пор, пока его разделение на части не станет оправданным. Кроме того, в подразделе 2.2.5 я перечислю несколько трудностей, связанных с декомпозицией, которые могут заставить вас пересмотреть свое решение.

Рассмотрим другой способ разбиения приложения, основанный на предметно-ориентированном проектировании.

2.2.3. Разбиение на сервисы по проблемным областям

Согласно описанию, данному Эриком Эвансом (Eric Evans) в книге *Domain-driven design* (Addison-Wesley Professional, 2003)¹, предметно-ориентированное проектирование (domain-driven design, DDD) — это способ построения сложных приложений, основанный на разработке объектно-ориентированной доменной (проблемной) модели. *Доменная модель* организует информацию о проблемной области в формате, который можно применять для решения проблем в этой области. Она определяет терминологию, используемую внутри команды, — так называемый *язык описания*. Доменная модель находит свое воплощение в проектировании и реализации приложения. DDD предлагает две концепции, чрезвычайно полезные с точки зрения микросервисной архитектуры: поддомены и изолированные контексты.

Шаблон «Декомпозиция по поддомену»

Определяет сервисы в соответствии с поддоменами в DDD. См. microservices.io/patterns/decomposition/decompose-by-subdomain.html.

DDD довольно сильно отличается от традиционного подхода к промышленному моделированию, в котором для целого предприятия создается единая модель. Такая модель, к примеру, содержала бы определение для каждого бизнес-объекта, такого как клиент, заказ и т. д. Проблема данной методики связана с тем, что создание общей модели, на которую согласны все подразделения организации, — это колоссальная задача. К тому же с точки зрения отдельных подразделений такая модель

¹ Эванс Э. Предметно-ориентированное проектирование (DDD). — М.: Вильямс, 2010.

будет слишком сложной для их конкретных нужд. Доменная модель может выглядеть запутанной, так как разные части организации могут использовать один термин для разных концепций или разные термины для одной и той же концепции. DDD позволяет избежать всех этих проблем за счет определения нескольких доменных моделей, каждая из которых имеет четкую область применения.

DDD определяет отдельную доменную модель для каждого поддомена. Поддомен является частью *домена*, то есть проблемной области приложения в терминологии DDD. Разбиение на поддомены происходит по тому же принципу, что и определение бизнес-возможностей: путем анализа работы бизнеса и определения разных областей знаний. Итоговые поддомены, скорее всего, будут похожи на бизнес-возможности. Примерами поддоменов в контексте FTGO являются принятие заказов, управление заказами, управление кухней, доставка и финансовая отчетность. Как видите, они очень похожи на бизнес-возможности, описанные ранее.

В DDD область применения доменной модели называется *изолированным контекстом*. Изолированный контекст включает в себя код, который реализует модель. При использовании микросервисной архитектуры изолированный контекст соответствует одному или нескольким сервисам. Мы можем создать микросервисную архитектуру, задействуя DDD и определяя сервисы для каждого поддомена. На рис. 2.9 показано, как поддомены привязываются к сервисам, каждый из которых имеет собственную доменную модель.

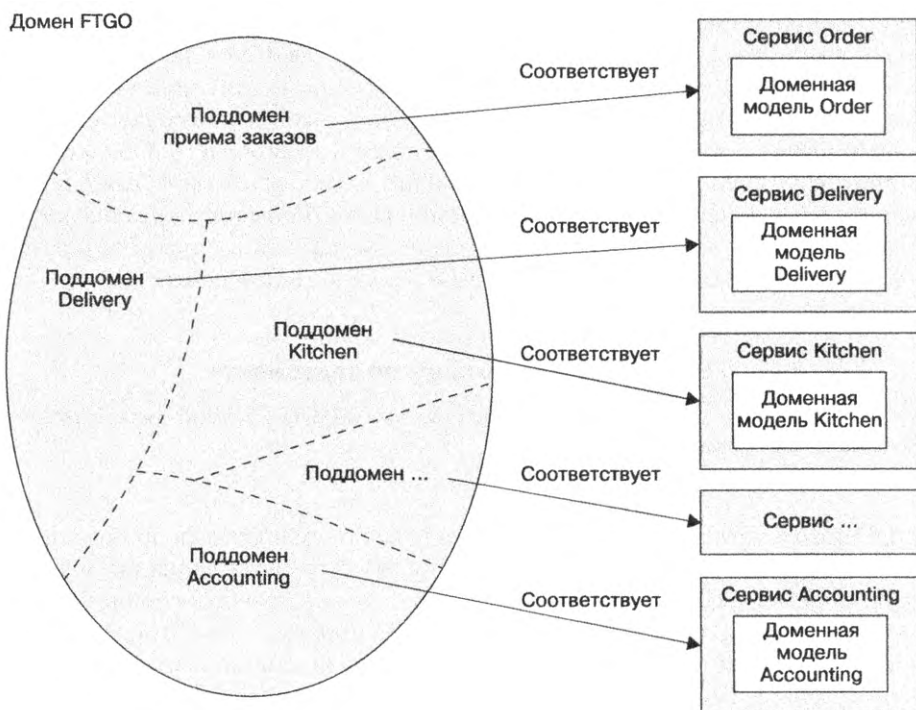


Рис. 2.9. От поддоменов к сервисам: каждый поддомен в домене приложения FTGO соответствует сервису с собственной доменной моделью

DDD почти идеально сочетается с микросервисной архитектурой. Концепции поддоменов и изолированных контекстов, применяемые в DDD, прекрасно соотносятся с микросервисами. К тому же концепция автономных команд, отвечающих за отдельные сервисы, полностью соответствует аналогичному принципу в DDD, согласно которому доменная модель отводится одной команде, которая ее разрабатывает. Но и это еще не все. Как вы увидите позже в этом разделе, поддомены с отдельными доменными моделями отлично борются с божественными классами и тем самым упрощают декомпозицию.

Разбиение по поддоменам и бизнес-возможностям — два основных подхода к описанию микросервисной архитектуры приложения. Однако некоторые полезные рекомендации относительно декомпозиции берут свое начало в объектно-ориентированном проектировании. Давайте их рассмотрим.

2.2.4. Методические рекомендации по декомпозиции

Мы уже обсудили в этой главе основные подходы к определению микросервисной архитектуры. Но при работе с микросервисами могут пригодиться и некоторые принципы, позаимствованные из объектно-ориентированного проектирования. Они созданы Робертом Мартином (Robert C. Martin) и описаны в его классической книге *Designing Object Oriented C++ Applications Using The Booch Method* (Prentice Hall, 1995). Во-первых, это принцип единственной ответственности (Single Responsibility Principle, SRP) для определения обязанностей класса. Во-вторых, принцип согласованного изменения (Common Closure Principle, CCP) для организации классов в виде пакетов. Рассмотрим каждый из них и попробуем понять, как их можно применить к микросервисной архитектуре.

Принцип единственной ответственности

Одной из основных задач проектирования программного обеспечения является определение обязанностей каждого программного элемента. Принцип единственной ответственности гласит следующее: *у класса должна быть только одна причина для изменения* (Роберт Мартин).

Каждая обязанность класса может нести в себе причину для его изменения. Если у класса несколько обязанностей, которые меняются независимо друг от друга, он не сможет оставаться стабильным. Согласно принципу SRP каждый класс должен иметь ровно одну обязанность и, следовательно, единственную причину для изменения.

Если применить SRP к микросервисной архитектуре, каждый сервис получится небольшим, согласованным и обладающим одной обязанностью. Это уменьшит размер сервисов и повысит их стабильность. Новая архитектура FTGO — живой пример применения SRP. Каждый аспект получения еды клиентом (принятие заказа, приготовление блюда и доставка) — это обязанность отдельного сервиса.

Принцип согласованного изменения

Еще одной полезной идеей является принцип согласованного изменения, гласящий: *причины изменения классов, входящих в один пакет, должны быть одинаковыми. Изменение пакета должно затрагивать все его классы* (Роберт Мартин).

Если два класса изменяются вместе по одной и той же причине, они должны входить в один пакет. Они могут, например, реализовывать разные аспекты определенного бизнес-правила. Суть в том, что, когда это бизнес-правило изменится, разработчикам нужно будет поменять код лишь в нескольких пакетах (в идеале только в одном). Соблюдение принципа ССР значительно упрощает поддержку приложения.

ССР можно применить при создании микросервисной архитектуры, объединяя компоненты, изменяющиеся по одной и той же причине, в единый сервис. Это позволит минимизировать количество сервисов, которые придется редактировать и заново развертывать при изменении какого-нибудь требования. В идеале такое изменение должно затрагивать лишь одну команду и один сервис. ССР — противоядие от антишаблона распределенного монолита.

SRP и ССР — это лишь два из 11 принципов, выработанных Бобом Мартином. Они особенно полезны при разработке микросервисной архитектуры. Остальные девять принципов используются при создании классов и пакетов. Больше информации об SRP, ССР и других аспектах объектно-ориентированного проектирования можно найти в статье *The Principles of Object Oriented Design* на сайте Боба Мартина (butunclebob.com/ArticleS.UncleBob).

Декомпозиция по бизнес-возможностям и поддоменам в сочетании с SRP и ССР хорошо подходит для разбиения приложения на сервисы. Но, чтобы применить эти методики и успешно разработать микросервисную архитектуру, вам придется решить некоторые проблемы с управлением транзакциями и межпроцессным взаимодействием.

2.2.5. Трудности при разбиении приложения на сервисы

На первый взгляд стратегия создания микросервисной архитектуры путем описания сервисов на основе бизнес-возможностей или поддоменов выглядит довольно простой. Тем не менее вы можете столкнуться со следующими проблемами:

- ☐ латентность сети;
- ☐ ухудшение доступности из-за синхронного взаимодействия;
- ☐ поддержание согласованности данных между сервисами;
- ☐ получение согласованного представления данных;
- ☐ божественные классы, препятствующие декомпозиции.

Пройдемся по всем этим проблемам, начиная с латентности сети.

Латентность сети

Латентность сети является вечной проблемой распределенных систем. Вы можете обнаружить, что определенный вид декомпозиции вынуждает сервисы часто обмениваться данными. Иногда латентность можно снизить до приемлемого уровня.

ня, реализовав пакетный API для извлечения нескольких объектов за один вызов. Но бывают ситуации, когда приходится объединять разные сервисы, отказываясь от IPC в пользу методов или функций уровня языка.

Синхронное межпроцессное взаимодействие ухудшает доступность

Еще одна проблема связана с необходимостью реализовать межсервисное взаимодействие таким образом, чтобы оно не сказывалось на доступности. Например, чтобы выполнить операцию `createOrder()`, проще всего сделать синхронный вызов из сервиса `Order`, используя REST. Недостатком таких протоколов, как REST, в данном случае является ухудшение доступности сервиса `Order`: он не сможет создать заказ, если любой из сервисов, к которым он обращается, окажется недоступным. Иногда это можно считать разумным компромиссом, но в главе 3 вы узнаете, что зачастую лучше применить асинхронный обмен сообщениями, который устраняет жесткую связанность и улучшает доступность.

Обеспечение согласованности данных между сервисами

Еще одним вызовом является поддержание согласованности данных между сервисами. Некоторым системным операциям нужно обновлять информацию в нескольких сервисах. Например, когда ресторан принимает заказ, обновления должны произойти в сервисах `Kitchen` и `Delivery`: первый меняет состояние объекта `Ticket`, а второй планирует доставку заказа. Эти обновления должны выполняться автоматически.

Традиционное решение этой задачи заключается в использовании двухэтапного механизма управления распределенными транзакциями, основанного на фиксации. Но, как вы убедитесь в главе 4, это не лучший выбор для современных приложений и для управления транзакциями лучше применять совсем другой подход — шаблон «Повествование». *Повествование* — это последовательность локальных транзакций, которые координируются путем обмена сообщениями. Повествования сложнее традиционных ACID-транзакций, но они хорошо подходят для многих ситуаций. У них есть одно ограничение — отложенная согласованность (*eventual consistency*). Если вам нужно, чтобы данные обновлялись автоматически, они должны находиться в пределах одного сервиса, что может помешать декомпозиции.

Получение согласованного представления данных

Еще одной трудностью на пути к декомпозиции является невозможность получения по-настоящему согласованного представления данных, расположенных в разных БД. В монолитных приложениях ACID-транзакции благодаря своим свойствам гарантируют, что запрос вернет согласованное представление базы данных. Для сравнения: микросервисная архитектура не позволяет получить глобально согласованное представление данных, несмотря на то что БД каждого отдельного сервиса согласована. Если вам нужно согласованное представление какой-то информации, она должна находиться в одном сервисе, что может нарушить декомпозицию. К счастью, в реальных условиях такая проблема возникает редко.

Божественные классы препятствуют декомпозиции

Еще одна трудность при композиции связана с существованием так называемых божественных классов. *Божественными* называют раздутые классы, которые используются в разных частях приложения (<http://wiki.c2.com/?GodClass>). Обычно они реализуют бизнес-логику для разных аспектов системы и содержат большое количество полей, привязанных к таблицам базы данных с множеством столбцов. У большинства приложений есть по меньшей мере один такой класс, представляющий центральную концепцию той или иной проблемной области: счета в банковской сфере, заказы в электронной торговле, полисы в страховании и т. д. Поскольку божественный класс вмещает в себе состояние и поведение множества разных аспектов приложения, он исключает разбиение на сервисы любой бизнес-логики, которая его применяет.

Отличный пример божественного класса в проекте FTGO — `Order`. Это и неудивительно — в конце концов, это приложение предназначено для доставки еды клиентам. Большинство участков системы связаны с заказами. Если бы мы использовали единую доменную модель, класс `Order` имел бы огромный размер. Его состояние и поведение пронизывали бы разные части кода. На рис. 2.10 показана структура класса, который получился бы при традиционных методах моделирования.

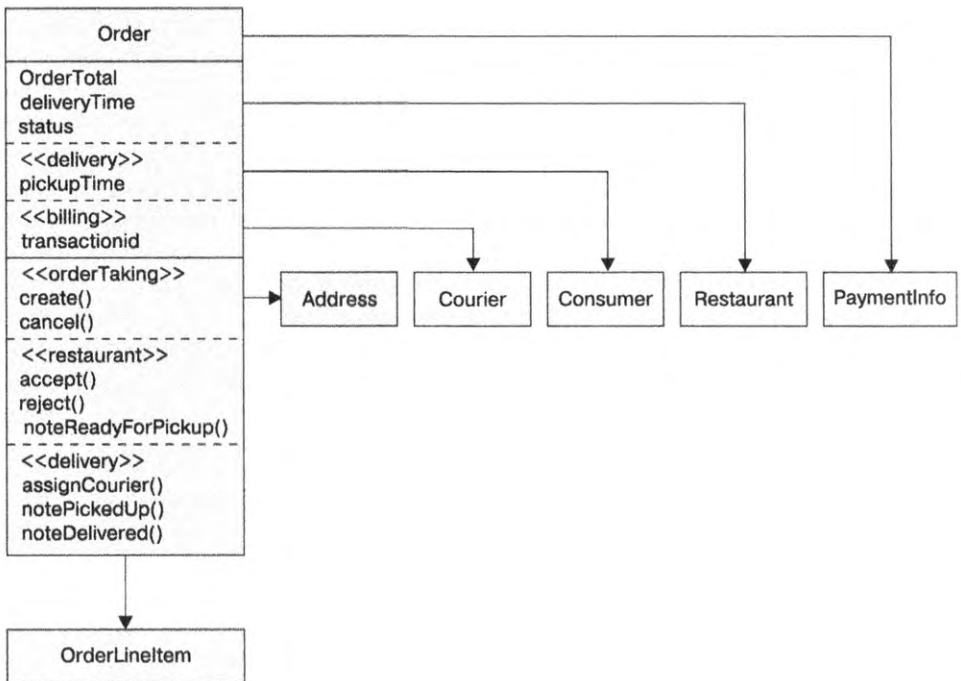


Рис. 2.10. Божественный класс `Order` слишком раздут и имеет многочисленные обязанности

Как видите, поля и методы класса `Order` связаны с обработкой заказов, управлением заказами в ресторанах, доставкой и платежами. Кроме того, усложнена модель данных, ведь она отвечает за описание переходов между состояниями в компонентах приложения, никак не связанных друг с другом. В своем теперешнем виде этот класс делает разделение кода на сервисы чрезвычайно трудным.

Одно из решений проблемы — упаковка класса `Order` в библиотеку и создание центральной базы данных `Order`. Все сервисы, обрабатывающие заказы, станут использовать эту библиотеку и обращаться к одноименной БД. Проблема с данным подходом состоит в том, что он нарушает ключевые принципы микросервисной архитектуры и приводит к нежелательному жесткому связыванию. Например, любое изменение структуры таблиц `Order` требует синхронного обновления кода со стороны других команд.

Еще одно решение связано с инкапсуляцией БД `Order` в одноименный сервис, который другие сервисы вызывают для получения и обновления заказов. Но в результате сервис `Order` отвечал бы только за данные и имел слабую доменную модель с минимальным количеством бизнес-логики или вовсе без нее. Ни один из этих вариантов не является привлекательным, но, к счастью, DDD предоставляет альтернативу.

Куда более удачным решением будет применение DDD и восприятие каждого сервиса как отдельного поддомена со своей доменной моделью. Это означает, что каждый сервис в приложении FTGO, имеющий какое-либо отношение к заказам, будет иметь собственную модель с отдельной версией класса `Order`. Отличная иллюстрация преимуществ множественных доменных моделей — сервис `Delivery`. Его представление класса `Order` (рис. 2.11) выглядит чрезвычайно просто: адрес получения, время получения, адрес доставки, время доставки. К тому же вместо `Order` используется более подходящее название — `Delivery`.

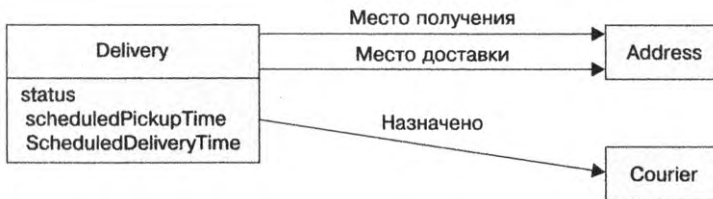


Рис. 2.11. Доменная модель сервиса `Delivery`

Сервис `Delivery` не заинтересован в остальных атрибутах `Order`.

Сервис `Kitchen` тоже имеет упрощенное представление заказа. Его версия класса `Order` называется `Ticket`. Он состоит из полей `status`, `requestedDeliveryTime` и `prepareByTime`, а также списка позиций, благодаря которому в ресторане знают, что нужно приготовить (рис. 2.12). Его не интересуют клиенты, оплата, доставка и т. д.

У сервиса `Order` самое сложное представление заказа (рис. 2.13). Но, несмотря на большое количество полей и методов, оно все равно намного проще исходной версии.



Рис. 2.12. Доменная модель сервиса Kitchen

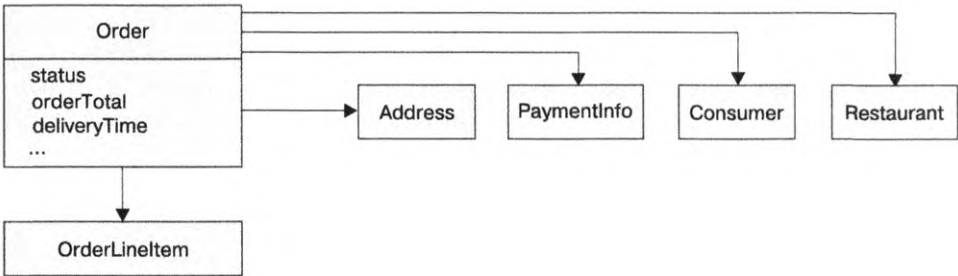


Рис. 2.13. Доменная модель сервиса Order

В каждой доменной модели класс `Order` представляет разные аспекты одной и той же бизнес-сущности. Приложение FTGO должно поддерживать согласованность между разными объектами в разных сервисах. Например, авторизовав банковскую карту клиента, сервис `Order` должен инициировать создание объекта `Ticket` в сервисе `Kitchen`. Аналогично, если ресторан отклонил заказ через сервис `Kitchen`, он должен быть отменен и в сервисе `Order`, а биллинговому сервису следует вернуть средства клиенту. Из главы 4 вы узнаете, как поддерживать связанность между сервисами, используя механизм повествований, основанный на событиях.

Помимо создания технических проблем, наличие множественных доменных моделей влияет на реализацию пользовательского интерфейса. Приложение должно наладить связь между интерфейсом пользователя, который сам по себе является отдельной доменной моделью, и доменными моделями каждого сервиса. Например, в приложении FTGO состояние заказа, которое выводится клиенту, извлекается из классов `Order`, принадлежащих нескольким сервисам. Эта связь часто реализуется посредством API-шлюза, описанного в главе 8. Несмотря на эти сложности, поиск и устранение божественных классов при создании микросервисной архитектуры имеет первостепенное значение.

Теперь посмотрим, как определяются API сервисов.

2.2.6. Определение API сервисов

На данном этапе у нас есть перечень системных операций и список потенциальных сервисов. Следующим шагом будет определение API каждого сервиса, его операций и событий. Операция в API может существовать по одной из двух причин: она либо соответствует системной операции и вызывается внешними клиентами (или, воз-

можно, другими сервисами), либо поддерживает взаимодействие между сервисами и вызывается только ими.

Сервис публикует события в основном для того, чтобы иметь возможность взаимодействовать с другими сервисами. В главе 4 рассказывается, как эти события можно использовать для реализации повествований, обеспечивающих согласованность данных между сервисами. А в главе 7 мы поговорим о том, как с помощью событий обновлять CQRS-представления с поддержкой эффективных запросов. Приложение также может задействовать события для уведомления внешних клиентов. Например, оно может оповещать браузер с помощью WebSocket.

Первое, что нужно сделать при определении API, — привязать системную операцию к сервису. После этого требуется решить, должен ли сервис взаимодействовать с другими сервисами для реализации системной операции. Если взаимодействие необходимо, следует определить, какие API эти сервисы должны предоставить для поддержки взаимодействия. Для начала посмотрим, как назначить сервису системную операцию.

Назначение сервисам системных операций

Первым делом нужно определить, какой сервис будет служить начальной входящей точкой для запроса. Многие системные операции легко соотнести с сервисами, но иногда этот процесс оказывается неочевидным. Возьмем, к примеру, операцию `noteUpdatedLocation()`, обновляющую местоположение курьера. С одной стороны, она относится к курьерам, поэтому ее следовало бы назначить сервису `Courier`. Но местоположение курьера нужно сервису `Delivery`. В данном случае назначение операции сервису, которому нужна информация, возвращаемая этой операцией, — наилучший выбор. В других ситуациях, возможно, имело бы смысл выбрать сервис, обладающий данными, необходимыми для выполнения операции.

В табл. 2.2 перечислены сервисы приложения FTGO и операции, за которые они отвечают.

Таблица 2.2. Привязка системных операций к сервисам в приложении FTGO

Сервис	Операция
Consumer	<code>createConsumer()</code>
Order	<code>createOrder()</code>
Restaurant	<code>findAvailableRestaurants()</code>
Kitchen	<code>acceptOrder()</code> <code>noteOrderReadyForPickup()</code>
Delivery	<code>noteUpdatedLocation()</code> <code>noteDeliveryPickedUp()</code> <code>noteDeliveryDelivered()</code>

Закончив с назначением системных операций, мы должны решить, какое взаимодействие между сервисами требуется для выполнения каждой из них.

Определение API, необходимых для поддержки взаимодействия между сервисами

Некоторые системные операции целиком выполняются одним сервисом. Например, в приложении FTGO сервис `Consumer` сам отвечает за операцию `createConsumer()`. В некоторых случаях для этого требуется несколько сервисов. Данные, необходимые для обработки одного из этих запросов, могут быть распределены по разным сервисам. Например, для реализации операции `createOrder()` сервис `Order` должен вызвать следующие сервисы, чтобы проверить предварительные условия и обеспечить выполнение окончательных.

- ❑ `Consumer` — проверяет, может ли клиент разместить заказ, и получает его платежную информацию.
- ❑ `Restaurant` — проверяет позиции заказа, нахождение адреса и времени доставки в пределах области обслуживания ресторана и набор минимальной суммы заказа, а также получает цены на заказанные блюда.
- ❑ `Kitchen` — создает объект `Ticket`.
- ❑ `Accounting` — авторизует банковскую карту клиента.

Точно так же, чтобы реализовать системную операцию `acceptOrder()`, сервис `Kitchen` должен обратиться к сервису `Delivery`, который назначит курьера для доставки заказа. В табл. 2.3 перечислены сервисы, их исправленные API и то, с чем они взаимодействуют. Чтобы полностью описать API сервиса, вы должны проанализировать каждую системную операцию и определить, какое взаимодействие для этого требуется.

Таблица 2.3. Сервисы, их исправленные API и то, с чем они взаимодействуют

Сервис	Операция	Связанный сервис
Consumer	verifyConsumerDetails()	—
Order	createOrder()	Consumer Service verifyConsumerDetails() Restaurant Service verifyOrderDetails() Kitchen Service createTicket() Accounting Service authorizeCard()
Restaurant	findAvailableRestaurants() verifyConsumerDetails()	—
Kitchen	createTicket() acceptOrder() noteOrderReadyForPickup()	Delivery ServicescheduleDelivery()
Delivery	scheduleDelivery() noteUpdatedLocation() noteDeliveryPickedUp() noteDeliveryDelivered()	—
Accounting	authorizeCard()	—

Итак, мы определили сервисы и операции, которые каждый из них реализует. Но важно помнить, что намеченная нами архитектура очень абстрактна. Мы не выбрали конкретной технологии IPC. Более того, несмотря на то что термин «*операция*» подразумевает некий IPC-механизм с синхронными запросами и ответами, вы увидите, что асинхронные сообщения играют здесь важную роль. Архитектурные концепции, описываемые на страницах книги, влияют на то, как эти сервисы будут взаимодействовать между собой.

В главе 3 описываются конкретные технологии IPC, включая такие механизмы синхронного взаимодействия, как REST и асинхронный обмен сообщениями с помощью брокера. Мы поговорим о том, как синхронное взаимодействие может сказаться на доступности, и познакомимся с концепцией автономных сервисов, которые не обращаются к другим компонентам приложения синхронно. Одним из методов реализации автономного сервиса является использование шаблона CQRS, рассматриваемого в главе 7. Сервис *Order*, например, мог бы хранить копию данных, принадлежащих сервису *Restaurant*, — это позволило бы ему устранить необходимость в синхронном вызове *Restaurant* для проверки заказа. Для поддержания копии в актуальном состоянии он может подписаться на события, которые сервис *Restaurant* публикует при каждом обновлении своих данных.

В главе 4 вы познакомитесь с концепцией повествований и узнаете, как сервисы, участвующие в повествовании, координируются с помощью асинхронных сообщений. Помимо надежного обновления информации, разбросанной по нескольким сервисам, повествование позволяет реализовать автономный сервис. Например, я покажу, как реализовать с помощью повествований операцию *createOrder()*, которая обращается к сервисам *Consumer*, *Kitchen* и *Accounting*, используя асинхронные сообщения.

В главе 8 описывается концепция API-шлюза, который делает API доступными для внешних клиентов. Вместо того чтобы просто направить запрос к сервису, API-шлюз может реализовать запрашивающую операцию, задействуя шаблон объединения API (это будет показано в главе 7). Логика API-шлюза собирает данные, необходимые запросу, обращаясь к нескольким сервисам и объединяя полученные результаты. В этом случае системная операция назначается не сервису, а API-шлюзу. Сервисы будут реализовывать операции-запросы, которые использует API-шлюз.

Резюме

- ❑ Архитектура определяет качественные характеристики приложения, влияющие непосредственно на темпы разработки: поддерживаемость, тестируемость и развертываемость.
- ❑ Микросервисная архитектура — это архитектурный стиль, который делает приложение хорошо поддерживаемым, тестируемым и развертываемым.
- ❑ Сервисы в микросервисной архитектуре организованы вокруг бизнес-аспектов (бизнес-возможностей или поддоменов), а не технических характеристик.

- ❑ Существует два вида декомпозиции.
 - Декомпозиция по бизнес-возможностям, которая берет начало в бизнес-архитектуре.
 - Декомпозиция по поддоменам, основанная на предметно-ориентированном проектировании.
- ❑ Вы можете избавиться от божественных классов, которые приводят к запутанным зависимостям и препятствуют декомпозиции, применяя DDD и определяя отдельную доменную модель для каждого сервиса.

Межпроцессное взаимодействие в микросервисной архитектуре

В этой главе

- Применение коммуникационных шаблонов: удаленный вызов процедур, предохранитель, обнаружение на стороне клиента, саморегистрация, обнаружение на стороне сервера, сторонняя регистрация, асинхронный обмен сообщениями, публикация событий, отслеживание транзакционного журнала, опрашивающий издатель.
- Роль межпроцессного взаимодействия в микросервисной архитектуре.
- Описание развивающихся API.
- Различные варианты межпроцессного взаимодействия, их плюсы и минусы.
- Преимущества от использования сервисов, которые взаимодействуют с помощью асинхронных сообщений.
- Надежная отправка сообщений в рамках транзакции в базе данных.

Мэри и ее команда, как и большинство других разработчиков, имеют определенный опыт использования механизмов межпроцессного взаимодействия (IPC). Приложение FTGO предоставляет интерфейс REST API, который применяется в мобильных клиентах и JavaScript на стороне браузера. Оно также задействует множество облачных сервисов, таких как Twilio (для обмена сообщениями) и Stripe (для платежей). Но, поскольку проект FTGO монолитный, его модули общаются друг с другом, вызывая методы и функции на уровне языка. Разработчикам FTGO обычно не нужно задумываться об IPC, разве что они имеют дело с REST API или модулями, которые интегрируются с облачными сервисами.

Для сравнения: как вы видели в главе 2, микросервисная архитектура структурирует приложение в виде набора сервисов. Чтобы обрабатывать запросы, этим сервисам часто приходится взаимодействовать между собой. Поскольку экземпляры сервисов обычно представляют собой процессы, запущенные на разных компьютерах, они должны общаться друг с другом с помощью IPC. В микросервисной архитектуре межпроцессное взаимодействие играет намного более важную роль, чем в монолитных приложениях. Таким образом, по мере разбиения своего монолита на микросервисы Мэри и остальным разработчикам FTGO нужно будет потратить много времени на обдумывание стратегий IPC.

Дефицита разнообразных механизмов IPC точно не наблюдается. Сейчас модным выбором является REST (с JSON). Хотя важно помнить, что идеальных решений, подходящих для любых ситуаций, не бывает. Вы должны тщательно проанализировать доступные варианты. В этой главе рассматриваются различные виды IPC, включая REST и обмен сообщениями, и взвешиваются все за и против.

Выбор механизма IPC — важное архитектурное решение, он может повлиять на уровень доступности приложения. Кроме того, как я продемонстрирую в этой и следующей главах, IPC пересекается даже с управлением транзакциями. Я отдаю предпочтение архитектуре, состоящей из слабо связанных сервисов, которые взаимодействуют между собой с помощью асинхронных сообщений. Синхронные протоколы, такие как REST, в основном используются для общения с другими приложениями.

Начну эту главу с обзора межпроцессного взаимодействия в микросервисной архитектуре. Затем опишу механизм IPC, основанный на удаленном вызове процедур, самый популярный пример которого — REST. Мы поговорим о том, как обнаружить сервисы и справиться с частичными отказами. После этого перейдем к IPC на основе асинхронных сообщений. Далее рассмотрим масштабирование клиентов с сохранением порядка следования сообщений, корректную обработку дубликатов и транзакционный обмен сообщениями. В конце пройдемся по концепции автономных сервисов, которые обрабатывают синхронные запросы без обращения к другим сервисам, чем улучшают доступность.

3.1. Обзор межпроцессного взаимодействия в микросервисной архитектуре

В вашем распоряжении множество разных технологий IPC. Сервисы могут использовать коммуникационные механизмы на основе запросов/ответов, такие как REST или gRPC, поверх HTTP. Альтернативным вариантом являются асинхронные механизмы коммуникации на основе сообщений, такие как AMQP или STOMP. Существует также множество других форматов сообщений — это могут быть как текстовые форматы, понятные человеку (JSON или XML), так и более эффективные двоичные, такие как Avro или Protocol Buffers.

Прежде чем погружаться в подробности тех или иных технологий, я хочу поговорить о нескольких проблемах проектирования, которые вы должны учитывать. Начнем со стилей взаимодействия — это способы описания взаимодействия между клиентами и сервисами, не привязанные к конкретным технологиям. Затем обсудим важность четкого определения API в микросервисной архитектуре и затронем концепцию проектирования, строящегося вокруг API. После этого перейдем к такой важной теме, как развитие API. В конце рассмотрим разные виды форматов сообщений и то, насколько они могут упростить развитие API.

3.1.1. Стили взаимодействия

Прежде чем выбирать механизм IPC для API сервиса, полезно будет подумать о стиле взаимодействия между сервисом и его клиентами. Это поможет сосредоточиться на требованиях и не увязнуть в деталях конкретной технологии IPC. К тому же выбор стиля взаимодействия влияет на уровень доступности вашего приложения (об этом будет говориться в разделе 3.4). Более того, как вы увидите в главах 9 и 10, это помогает выбрать подходящую стратегию интеграционного тестирования.

Существует много разных стилей взаимодействия между клиентом и сервисом. Как видно в табл. 3.1, их можно разделить на два уровня. Первый уровень определяет выбор между отношениями «один к одному» и «один ко многим»:

- ❑ «один к одному» — каждый клиентский запрос обрабатывается ровно одним сервисом;
- ❑ «один ко многим» — каждый запрос обрабатывается несколькими сервисами.

Второй уровень определяет выбор между синхронным и асинхронным взаимодействием:

- ❑ *синхронное* — клиент рассчитывает на своевременный ответ от сервиса и может даже заблокироваться на время ожидания;
- ❑ *асинхронное* — клиент не блокируется, а ответ, если таковой придет, может быть отправлен не сразу.

Таблица 3.1. Различные стили взаимодействия можно охарактеризовать на двух уровнях: «один к одному» или «один ко многим», а также синхронное или асинхронное

Взаимодействие	Один к одному	Один ко многим
Синхронное	Запрос/ответ	—
Асинхронное	Асинхронный запрос/ответ, однонаправленные уведомления	Издатель/подписчик, издатель/асинхронные ответы

Далее перечислены разные виды взаимодействия «один к одному».

- ❑ *Запрос/ответ* — клиент отправляет сервису запрос и ждет ответа. Он рассчитывает на то, что ответ придет своевременно, и может даже заблокироваться

на время ожидания. Этот стиль взаимодействия обычно приводит к жесткой связанности сервисов.

- ❑ *Асинхронный запрос/ответ* — клиент отправляет запрос, а сервис отвечает асинхронно. Клиент не блокируется на время ожидания, поскольку сервис может долго не отвечать.
- ❑ *Однонаправленные уведомления* — клиент отправляет сервису запрос, не ожидая (и не получая) ничего в ответ.

Важно помнить, что стиль взаимодействия с синхронными запросами/ответами очень опосредованно относится к технологиям IPC. Например, для взаимодействия сервисов в стиле «запрос/ответ» можно использовать как REST, так и обмен сообщениями. Даже если два сервиса задействуют брокер сообщений, клиентский сервис может блокироваться в ожидании ответа. Это вовсе не означает, что они слабо связаны. К этому мы вернемся позже, во время обсуждения того, как межсервисное взаимодействие влияет на уровень доступности.

Далее перечислены виды взаимодействия «один ко многим».

- ❑ *Издатель/подписчик* — клиент публикует сообщение с уведомлением, которое потребляется любым количеством заинтересованных сервисов.
- ❑ *Издатель/асинхронные ответы* — клиент публикует сообщение с запросом и ждет определенное время ответа от заинтересованных сервисов.

Сервисы обычно используют сочетание этих стилей взаимодействия. Многие сервисы в приложении FTGO предоставляют для выполнения операций как синхронные, так и асинхронные API, а также публикуют события.

Посмотрим, как описать API сервиса.

3.1.2. Описание API в микросервисной архитектуре

API являются одним из важнейших аспектов разработки программного обеспечения. Приложение состоит из модулей. У каждого модуля есть интерфейс, определяющий набор операций, которые могут вызываться клиентами модуля. Хорошо спроектированный интерфейс делает доступными полезные функции, скрывая при этом их реализацию. Благодаря этому внутренние изменения не влияют на клиентов.

В монолитном приложении интерфейсы обычно описываются с помощью конструкций языка программирования — например, в виде Java-интерфейса, который определяет набор методов, доступных клиенту. Класс-реализация от клиента спрятан. Более того, поскольку Java — статически типизированный язык, любая несовместимость между интерфейсом и клиентом будет мешать компиляции.

API и локальные интерфейсы играют одинаково важную роль в микросервисной архитектуре. API сервиса является контрактом между ним и его клиентами. Как говорилось в главе 2, API состоит из операций, которые клиент может вызывать, и событий, публикуемых сервисом. У операции есть имя, параметры и тип возвра-

щаемого значения. Событие имеет тип и набор полей, оно публикуется в канале сообщений (это описано в разделе 3.3).

Трудность заключается в том, что определение API сервиса не основано на простых конструкциях языка программирования. Как и полагается, сервис и его клиенты компилируются отдельно. Если будет развернута новая версия сервиса с несовместимым API, вы не получите ошибку компиляции — вместо этого возникнут сбои на этапе выполнения.

Независимо от того, какой механизм IPC вы выберете, важно создать четкое определение API сервиса, используя некий *язык описания интерфейсов* (interface definition language, IDL). Кроме того, существуют хорошие аргументы в пользу того, что описание сервиса должно начинаться с его API (см. www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10). Первым делом вы описываете интерфейс. Затем рассматриваете полученный результат с клиентскими разработчиками. И только после окончания работы над API реализуете сам сервис. Такой подход повышает шансы на то, что ваш сервис будет удовлетворять требованиям клиентов.

Первоочередность API в проектировании

Даже в небольших проектах я наблюдал возникновение проблем из-за того, что компоненты имели разногласия относительно API. Например, в одном из проектов разработчик на Java и клиентский разработчик на AngularJS объявили о завершении разработки. Но приложение оказалось нерабочим. Интерфейсы REST и WebSocket, которые применялись в клиентской части для взаимодействия с сервером, были плохо описаны. В итоге две части приложения не могли между собой общаться!

Характер определения API зависит от выбранного механизма IPC. Например, если вы используете обмен сообщениями, API будет состоять из каналов, типов и форматов сообщений. Если применяете HTTP, API будет основан на URL-адресах, HTTP-командах и форматах запроса и ответа. Позже в этой главе я покажу, как описываются API.

API сервиса редко оказывается неизменным. Скорее всего, он будет эволюционировать со временем. Посмотрим, как это происходит и с какими проблемами вы можете столкнуться.

3.1.3. Развивающиеся API

API непременно меняются со временем по мере добавления новых, изменения существующих и иногда удаления старых возможностей. В монолитном приложении изменение API и обновление кода, который его вызывает, — довольно простые процессы. Если вы применяете статически типизированный язык, компилятор выдаст вам список соответствующих ошибок. Единственной трудностью может

оказаться масштаб изменений: на обновление широко используемого API может уйти много времени.

В приложениях, основанных на микросервисах, изменение API сервиса куда сложнее. Клиентами сервиса выступают другие сервисы, которые часто разрабатываются другими командами, или даже посторонние приложения за пределами организации. Обычно синхронное обновление всех клиентов вместе с сервисом не представляется возможным. Кроме того, поскольку современные приложения, как правило, не останавливаются на время обслуживания, обновление, скорее всего, будет плавающим, то есть новая и старая версии сервиса станут работать параллельно.

Чтобы справиться с этими трудностями, необходимо иметь стратегию. То, как происходит изменение API, зависит от характера изменения.

Семантическая нумерация версий

Спецификация семантического версионирования (semver.org) — хорошее руководство по нумерации версий API. Это набор правил, регламентирующих, как использовать и увеличивать номера версий. Семантическое версионирование изначально создавалось для программных пакетов, но вы можете задействовать его для нумерации версий API в распределенных системах.

Согласно спецификации семантического версионирования (Semver) номер версии должен состоять из трех частей: **MAJOR.MINOR.PATCH**. Каждая часть должна инкрементироваться следующим образом:

- ❑ **MAJOR** — при внесении в API несовместимого изменения;
- ❑ **MINOR** — при изменении API с сохранением обратной совместимости;
- ❑ **PATCH** — при исправлении ошибки с сохранением обратной совместимости.

Номера версий в API можно использовать в нескольких местах. Если вы реализуете REST API, мажорную версию можно указать в качестве первого элемента URL-пути (как показано далее). Если сервис задействует обмен сообщениями, можете включать номер версии в публикуемые сообщения. Целями являются корректное версионирование API и их контролируемое развитие. Посмотрим, как вносятся минорные и мажорные изменения.

Внесение минорных изменений с обратной совместимостью

В идеале следует стремиться к тому, чтобы все ваши изменения были обратно совместимыми. Такие изменения API являются добавочными:

- ❑ добавление новых атрибутов к запросу;
- ❑ добавление атрибутов к ответу;
- ❑ добавление новых операций.

Если вы всегда будете делать только такие изменения, старые клиенты смогут работать с новыми сервисами. Но для этого необходимо соблюдать принцип устой-

чивости (en.wikipedia.org/wiki/Robustness_principle), который гласит: «Будь консервативен в собственных действиях и либерален к тому, что принимаешь от других». Сервисы должны предоставлять значения по умолчанию для пропущенных атрибутов запроса. В то же время клиентам следует игнорировать любые лишние атрибуты ответа. Чтобы это не вызвало проблем, клиенты и сервисы должны использовать формат запросов/ответов, который поддерживает принцип устойчивости. Позже в этом разделе вы увидите, как такие текстовые форматы, как JSON и XML в целом упрощают развитие API.

Внесение мажорных, ломающих изменений

Иногда в API приходится вносить мажорные, несовместимые изменения. Поскольку вы не можете сделать так, чтобы клиенты сразу же обновились, сервис должен некоторое время поддерживать одновременно старую и новую версии API. Если вы используете механизм IPC, основанный на HTTP (такой как REST), мажорную версию можно сделать частью URL. Например, пути к версии 1 будут иметь префикс `'/v1/...'`, а пути к версии 2 — `'/v2/...'`.

Еще один вариант: применить механизм согласования содержимого в HTTP и включить номер версии в MIME-тип. Например, чтобы обратиться к версии 1.x сервиса `Order`, клиент выполнит такой запрос:

```
GET /orders/xyz HTTP/1.1
Accept: application/vnd.example.resource+json; version=1
...
```

Этот запрос говорит сервису `Order` о том, что клиент ожидает получить ответ версии 1.x.

Для поддержки нескольких версий API у сервиса есть адаптеры, логика которых позволяет переходить между старой и новой версиями. К тому же, как описано в главе 8, API-шлюз практически всегда использует версионированные API. Он также может поддерживать множество старых версий интерфейса.

Теперь поговорим о форматах сообщений, выбор которых может повлиять на то, насколько легко будет развивать ваш API.

3.1.4. Форматы сообщений

Суть IPC состоит в обмене сообщениями. *Сообщения* обычно содержат данные, выбор формата для которых является важным архитектурным решением. Это может повлиять на эффективность IPC, удобство API и простоту его развития. Если вы применяете систему обмена сообщениями или протоколы вроде HTTP, выбор формата ложится на вас. Некоторые механизмы IPC, такие как gRPC (мы познакомимся с ним чуть позже), могут сами диктовать формат сообщений. В любом случае важно не привязываться к конкретному языку. Даже если сейчас вы используете только один язык программирования, в будущем ситуация может измениться. Например, не следует выбирать сериализацию Java.

Форматы сообщений делятся на две категории: текстовые и двоичные. Рассмотрим каждую из них.

Текстовые форматы сообщений

В первую категорию входят текстовые форматы, такие как JSON и XML. Они не нуждаются в дополнительном описании, помимо того что их может прочитать человек. JSON-сообщение представляет собой набор именованных свойств. Точно так же XML-сообщение, в сущности, является перечнем именованных элементов и значений. Этот формат позволяет потребителям выбирать интересующие их данные, игнорируя все остальное. Таким образом, многие изменения структуры сообщения могут быть обратно совместимыми.

Структура XML-документов определяется их спецификацией (www.w3.org/XML/Schema). Со временем сообщество разработчиков пришло к тому, что JSON тоже нуждается в подобном механизме. Одно из популярных решений состоит в использовании стандарта JSON Schema (json-schema.org). JSON schema определяет имена и типы свойств внутри сообщения и то, обязательны ли они. JSON schema может предоставлять не только полезную документацию, но и механизм для проверки входящих сообщений.

Недостаток применения текстовых форматов связан с тем, что сообщения получаются довольно объемными, особенно если речь идет об XML. Помимо самих значений, сообщение также предоставляет их имена. Еще одной отрицательной стороной являются накладные расходы на разбор текста, особенно в ходе работы с большими сообщениями. Следовательно, если вам важны эффективность и производительность, лучше подумать об использовании двоичных форматов.

Двоичные форматы сообщений

Вы можете выбрать из нескольких двоичных форматов. К самым популярным относятся Protocol Buffers (developers.google.com/protocol-buffers/docs/overview) и Avro (avro.apache.org). Оба эти формата предоставляют типизированный язык IDL для описания структуры ваших сообщений. Затем компилятор генерирует код, который их сериализует и десериализует. Вы обязаны начинать проектирование сервисов с API! Более того, если пишете клиентский код на статически типизированном языке, компилятор проверит корректность использования API.

Одно из различий между этими форматами состоит в том, что Protocol Buffers задействует маркированные поля, тогда как клиент Avro должен знать спецификацию сообщения, чтобы его интерпретировать. В итоге Protocol Buffers делает развитие API проще по сравнению с Avro. Отличное сравнение Thrift, Protocol Buffers и Avro выполнено в статье martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html.

Рассмотрев форматы сообщений, мы можем перейти к конкретным механизмам IPC, которые эти сообщения транспортируют. Начнем с удаленного вызова процедур.

3.2. Взаимодействие на основе удаленного вызова процедур

При помощи механизма удаленного вызова процедур (remote procedure invocation, RPI) клиент отправляет запрос сервису, а тот его обрабатывает и возвращает ответ. Некоторые клиенты могут блокироваться в ожидании ответа, а другие поддерживают реактивную, неблокирующую архитектуру. Но, в отличие от использования сообщений, клиент рассчитывает на своевременное получение ответа.

Принцип работы RPI показан на рис. 3.1. Клиентская бизнес-логика обращается к *прокси-интерфейсу*, реализованному классом-адаптером *RPI-прокси*. RPI-прокси выполняет запрос к сервису. Запрос обрабатывается классом-адаптером *RPI-сервер*, который вызывает бизнес-логику сервиса через интерфейс. Затем ответ передается обратно в RPI-прокси, который возвращает результат клиентской бизнес-логике.

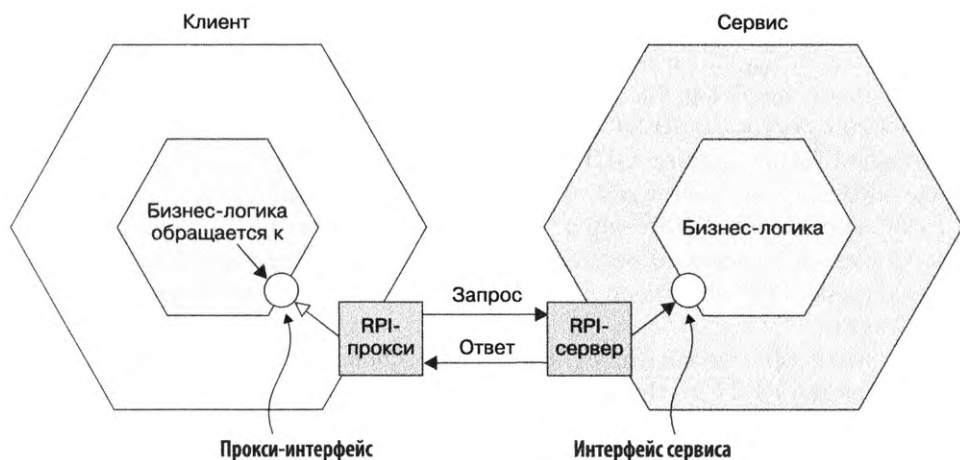


Рис. 3.1. Бизнес-логика клиента обращается к интерфейсу, реализованному классом-адаптером RPI-прокси. А он выполняет запрос к сервису. Класс-адаптер RPI-сервера обрабатывает запрос, вызывая бизнес-логику сервиса

Шаблон «Удаленный вызов процедур»

Клиент обращается к сервису по синхронному протоколу на основе удаленного вызова процедур, такому как REST (microservices.io/patterns/communication-style/messaging.html).

Прокси-интерфейс обычно инкапсулирует исходный коммуникационный протокол. Таких протоколов существует великое множество. В этом разделе я опишу REST и gRPC. Покажу, как повысить уровень доступности ваших сервисов

с помощью адекватной реакции на частичные сбои, и объясню, почему микросервисное приложение, задействующее RPI, должно иметь механизм обнаружения сервисов.

Сначала взглянем на REST.

3.2.1. Использование REST

В наши дни стало модным разрабатывать API в стиле RESTful (<https://ru.wikipedia.org/wiki/REST> и en.wikipedia.org/wiki/Representational_state_transfer). REST — это механизм IPC, который задействует HTTP (почти всегда). Рой Филдинг (Roy Fielding), создатель REST, дает следующее определение этой технологии: *REST предоставляет набор архитектурных ограничений, которые, если их применять как единое целое, делают акцент на масштабируемости взаимодействия между компонентами, обобщенности интерфейсов, независимом развертывании компонентов и промежуточных компонентах, чтобы снизить латентность взаимодействия, обеспечить безопасность и инкапсулировать устаревшие системы* (www.ics.uci.edu/~fielding/pubs/dissertation/top.htm).

Ресурс является ключевой концепцией в REST. Он представляет отдельный бизнес-объект, такой как Customer или Product, или коллекцию бизнес-объектов. Для работы с ресурсами REST использует HTTP-команды, которые указываются с помощью URL. Например, GET-запрос возвращает представление ресурса, часто в виде XML-документа или объекта JSON, хотя допускаются и другие форматы, в том числе двоичные. POST-запрос создает новый ресурс, а PUT-запрос обновляет существующий. У сервиса Order, к примеру, есть конечные точки POST /orders и GET /orders/{orderId} для создания нового и соответственно извлечения существующего заказа.

Многие разработчики считают, что их API, основанные на HTTP, соответствуют требованиям RESTful. Но, как утверждает Рой Филдинг в статье на roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven, это не всегда верно. Чтобы понять, почему так происходит, рассмотрим модель зрелости REST.

Модель зрелости REST

Леонард Ричардсон (Leonard Richardson) (однофамилец автора этой книги) предлагает крайне полезную модель зрелости для REST (martinfowler.com/articles/richardsonMaturityModel.html), которая состоит из следующих уровней.

- ❑ *Уровень 0.* Клиенты обращаются к сервису этого уровня путем выполнения HTTP-запроса типа POST к его единственной конечной точке (URL). Каждый запрос указывает выполняемое действие, цель этого действия (например, бизнес-объект) и различные параметры.
- ❑ *Уровень 1.* Сервисы этого уровня поддерживают концепцию ресурса. Чтобы выполнить какое-то действие с ресурсом, клиент выполняет POST-запрос, указывая действие и различные параметры.

- ❑ *Уровень 2.* Для выполнения действий сервисы второго уровня используют HTTP-команды: GET для извлечения, POST для создания и PUT — для обновления. Для задания параметров действия служат параметры и тело запроса, если таковые имеются. Это позволяет сервисам задействовать инфраструктуру протокола HTTP, включая кэширование GET-запросов.
- ❑ *Уровень 3.* Архитектура сервисов третьего уровня основана на принципе с ужасным названием — HATEOAS (Hypertext as the Engine of Application State — гипертекст в качестве ядра для состояния приложения). Основной его смысл в том, что представление ресурса, возвращаемое GET-запросом, содержит ссылки для выполнения действий с этим ресурсом. Например, клиент может отменить заказ с помощью ссылки в представлении, полученном в результате GET-запроса, который этот заказ извлек. В число преимуществ HATEOAS входит то, что вам больше не нужно встраивать URL-адреса в клиентский код (www.infoq.com/news/2009/04/hateoas-restful-api-advantages).

Я советую вам просмотреть интерфейсы REST API в своей организации и подумать над тем, каким уровням они соответствуют.

Описание REST API

Как упоминалось в разделе 3.1, для определения API необходимо использовать язык описания интерфейсов (interface definition language, IDL). В отличие от старых коммуникационных протоколов, таких как CORBA и SOAP, у REST изначально не было IDL. К счастью, сообщество разработчиков заново открыло для себя ценность такого языка в контексте RESTful API. Самым популярным REST IDL является спецификация Open API Specification (www.openapis.org). Она берет начало в открытом проекте Swagger, который представляет собой набор инструментов для разработки и документирования интерфейсов REST API. Он включает в себя утилиты для генерации клиентских заглушек и серверных каркасов на основе определения интерфейса.

Трудности извлечения нескольких ресурсов за один запрос

REST-ресурсы обычно ориентируются на бизнес-объекты, такие как *Consumer* или *Order*. Следовательно, при проектировании REST API часто возникает проблема с тем, как позволить клиенту извлекать несколько родственных объектов за один запрос. Представьте, например, что REST-клиент хочет извлечь информацию о заказе и его заказчике. Если строго следовать стандарту REST API, для этого потребуется как минимум два запроса: один для *Order*, другой — для *Consumer*. В более сложном сценарии нам пришлось бы передавать данные туда и обратно больше двух раз, что сказалось бы на времени отклика.

Чтобы решить эту проблему, можно позволить клиенту извлекать не только сам ресурс, но и все объекты, которые с ним связаны. Например, клиент мог бы

извлечь заказ и информацию о заказчике с помощью запроса `GET /orders/order-id-1345?expand=consumer`. В параметре запроса указан ресурс, который нужно вернуть вместе с заказом. Этот подход хорош во многих сценариях, но в более сложных ситуациях его обычно не хватает. К тому же на его реализацию может потребоваться слишком много времени. Все это сделало популярными альтернативные технологии построения API, такие как GraphQL (graphql.org) и Netflix Falcor (netflix.github.io/falcor/), изначально рассчитанные на эффективное извлечение данных.

Трудности с привязкой операций к HTTP-командам

Еще одна распространенная проблема проектирования REST API связана с тем, как привязать операции, которые вы хотите выполнять с бизнес-объектом, к HTTP-команде. REST API должен использовать команду PUT для обновления, но обновить заказ можно разными способами, например отменить, отредактировать и т. д. К тому же обновление может оказаться неидемпотентным, чем нарушит правила применения PUT. Одно из решений состоит в определении подресурса для обновления отдельных аспектов объекта. Например, у сервиса Order может существовать конечная точка `POST /orders/{orderId}/cancel` для отмены заказов и `POST /orders/{orderId}/revise` для их редактирования. Еще одним решением может стать задание команды в параметре запроса. К сожалению, ни одно из них до конца не отвечает принципам RESTful.

Трудности с привязкой операций к HTTP-командам играют на руку альтернативам REST, таким как gRPC (подробнее о них — в подразделе 3.2.2). Но прежде, чем перейти к ним, рассмотрим преимущества и недостатки REST.

Преимущества и недостатки REST

Стандарт REST обладает множеством положительных качеств.

- ❑ Он простой и привычный.
- ❑ API на основе HTTP можно тестировать в браузере, используя, к примеру, расширение Postman, или в командной строке с помощью `curl` (при условии, что вы применяете JSON или другой текстовый формат).
- ❑ Он имеет встроенную поддержку стиля взаимодействия вида «запрос/ответ».
- ❑ Протокол HTTP, естественно, дружелюбен к брандмауэрам.
- ❑ Он не нуждается в промежуточном брокере, что упрощает архитектуру системы.

Однако использование REST имеет и недостатки.

- ❑ Он поддерживает лишь стиль взаимодействия вида «запрос/ответ».
- ❑ Степень доступности снижена. Поскольку клиент и сервис взаимодействуют между собой напрямую, без промежуточного звена для буферизации сообщений, они оба должны работать на протяжении всего обмена данными.
- ❑ Клиенты должны знать местонахождение (URL) экземпляра (-ов) сервиса. Как описывается в подразделе 3.2.4, это нетривиальная проблема для современ-

ных приложений. Для определения местонахождения экземпляров сервисов клиентам приходится использовать так называемый *механизм обнаружения сервисов*.

- ❑ Извлечение нескольких ресурсов за один запрос связано с определенными трудностями.
- ❑ Иногда непросто привязать к HTTP-командам несколько операций обновления.

Несмотря на эти недостатки, REST считается стандартом де-факто для построения API, хотя у него есть несколько любопытных альтернатив. GraphQL, к примеру, реализует гибкое и эффективное извлечение данных (этот стандарт обсуждается в главе 8, в которой речь пойдет также о шаблоне API-шлюза).

Еще одна альтернатива REST — технология gRPC. Посмотрим, как она работает.

3.2.2. Использование gRPC

Как упоминалось в предыдущем разделе, одна из трудностей применения REST связана с тем, что HTTP поддерживает ограниченный набор команд, из-за чего процесс проектирования REST API с поддержкой нескольких операций обновления не всегда оказывается простым. Одна из технологий, которой удастся избежать этой проблемы, — gRPC (www.grpc.io). Это фреймворк для написания многоязычных клиентов и серверов (см. ru.wikipedia.org/wiki/Удалённый_вызов_процедур). gRPC представляет собой двоичный протокол на основе сообщений. Как вы помните из обсуждения двоичных форматов, это означает, что проектирование сервиса должно начинаться с его API. API в gRPC описывается с помощью языка IDL на основе Protocol Buffers — многоязычного механизма сериализации структурированных данных от компании Google. Компилятор Protocol Buffer генерирует клиентские заглушки и серверные каркасы. Он поддерживает разные языки, включая Java, C#, NodeJS и GoLang. Клиенты и серверы обмениваются сообщениями в формате Protocol Buffers, используя HTTP/2.

gRPC API состоит из одного или нескольких определений сервисов и сообщений вида «запрос/ответ». *Определение сервиса* является аналогом интерфейса в Java и представляет собой набор строго типизированных методов. Помимо простых вызовов, состоящих из запроса и ответа, gRPC поддерживает поточный RPC. Сервер может вернуть клиенту поток сообщений. В то же время клиент может сам отправить поток сообщений на сервер.

В качестве формата сообщений gRPC использует Protocol Buffers. Как уже упоминалось, это эффективный и компактный двоичный формат. Он является маркируемым. Каждое поле сообщения нумеруется и содержит код типа. Получатель сообщения может извлечь нужные ему поля, а остальные, которые не распознает, — проигнорировать. В итоге gRPC позволяет развивать API с сохранением обратной совместимости.

В листинге 3.1 показан фрагмент интерфейса gRPC API для сервиса `Order`. Он описывает несколько методов, включая `createOrder()`. Этот метод принимает `CreateOrderRequest` в качестве параметра и возвращает `CreateOrderReply`.

Листинг 3.1. Фрагмент интерфейса gRPC API для сервиса Order

```

service OrderService {
  rpc createOrder(CreateOrderRequest) returns (CreateOrderReply) {}
  rpc cancelOrder(CancelOrderRequest) returns (CancelOrderReply) {}
  rpc reviseOrder(ReviseOrderRequest) returns (ReviseOrderReply) {}
  ...
}

message CreateOrderRequest {
  int64 restaurantId = 1;
  int64 consumerId = 2;
  repeated LineItem lineItems = 3;
  ...
}

message LineItem {
  string menuItemId = 1;
  int32 quantity = 2;
}

message CreateOrderReply {
  int64 orderId = 1;
}
...

```

Сообщения `CreateOrderRequest` и `CreateOrderReply` являются типизированными. Например, `CreateOrderRequest` содержит поле `restaurantId` типа `int64`, а значение его метки равно 1.

Протокол gRPC обладает несколькими преимуществами.

- ❑ Он позволяет легко спроектировать API с богатым набором операций обновления.
- ❑ Он имеет эффективный компактный механизм IPC, что особенно явно проявляется при обмене крупными сообщениями.
- ❑ Поддержка двунаправленных потоков делает возможными стили взаимодействия на основе RPI и обмена сообщениями.
- ❑ Он позволяет сохранять совместимость между клиентами и сервисами, написанными на совершенно разных языках.

У gRPC есть и несколько недостатков.

- ❑ Процесс работы с API, основанным на gRPC, оказывается для JavaScript-клиентов более трудоемким, чем с API, основанным на REST/JSON.
- ❑ Старые брандмауэры могут не поддерживать HTTP/2.

Протокол gRPC — это полноценная альтернатива REST, хотя оба они представляют собой синхронные коммуникационные механизмы и, как следствие, страдают из-за проблем с частичным отказом. Давайте подробнее поговорим об этом недостатке и посмотрим, как с ним справиться.

3.2.3. Работа в условиях частичного отказа с применением шаблона «Предохранитель»

Каждый раз, когда сервис в распределенной системе делает синхронный запрос к другому сервису, возникает риск частичного отказа. Поскольку сервис является отдельным процессом, он может не ответить вовремя на запрос клиента. Или оказаться недоступным из-за сбоя, или находиться в процессе технического обслуживания. Кроме того, сервис может быть перегруженным и отвечать на запросы чрезвычайно медленно.

Поскольку клиент блокируется в ожидании ответа, существует опасность того, что его собственные клиенты тоже окажутся заблокированными и так по цепочке откажет вся система.

Шаблон «Предохранитель»

RPI-прокси, который в случае достижения определенного лимита последовательных отказов начинает отклонять все вызовы, пока не истечет определенное время. См. microservices.io/patterns/reliability/circuit-breaker.html.

Представьте: сервис Order перестал отвечать (рис. 3.2). Мобильный клиент делает REST-запрос к API-шлюзу, который, как вы увидите в главе 8, служит для клиентов API входной точкой в приложение. API-шлюз проксирует запрос к недоступному сервису Order.



Рис. 3.2. API-шлюз должен защищаться от неотзывчивых сервисов, таких как Order

Оптимистичная реализация `OrderServiceProxy` блокировалась бы до бесконечности в ожидании ответа. Это плохо сказалось бы не только на удобстве использования, но и во многих случаях на потреблении ценных ресурсов, таких как потоки. Рано или поздно ресурсы закончились бы, и API-шлюз не смог бы обслуживать запросы. Весь API стал бы недоступным.

При проектировании сервисов необходимо позаботиться о том, чтобы частичный отказ не мог распространиться по всему приложению. Решение этой задачи состоит из двух частей.

- ❑ Вы должны использовать RPI-прокси наподобие `OrderServiceProxy`, чтобы справляться с недоступными удаленными сервисами.
- ❑ Вам нужно решить, как восстановиться после отказа удаленного сервиса.

Для начала посмотрим, как написать надежный RPI-прокси.

Разработка надежных RPI-прокси

Каждый раз, когда один сервис вызывает другой, он должен защитить себя с помощью метода, предложенного компанией Netflix (techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html). Этот подход сочетает в себе следующие механизмы.

- ❑ *Сетевое время ожидания.* Никогда не блокируйтесь бессрочно, всегда отсчитывайте время ожидания запроса. Это гарантирует, что когда-нибудь ресурсы освободятся.
- ❑ *Ограничение количества неудачных запросов от клиента к сервису.* Установите лимит максимального количества неудавшихся запросов, которые клиент может послать определенному сервису. При исчерпании этого лимита выполнение дальнейших запросов, скорее всего, будет бессмысленным, поэтому такие попытки должны сразу завершаться ошибкой.
- ❑ *Шаблон «Предохранитель».* Отслеживайте количество успешных и неудавшихся запросов. Если частота ошибок превысит некий порог, разомкните предохранитель, чтобы дальнейшие попытки сразу же завершались. Большое количество неудачных запросов говорит о том, что сервис недоступен и обращаться к нему не имеет смысла. По истечении какого-то периода клиент должен предпринять новую попытку и, если она окажется успешной, замкнуть предохранитель.

Netflix Hystrix (github.com/Netflix/Hystrix) — это библиотека с открытым исходным кодом, которая реализует эти и другие шаблоны. Если вы работаете с JVM, вам определенно стоит подумать об использовании Hystrix при реализации RPI-прокси. Если же имеете дело со средой, не основанной на JVM, следует применять аналогичную библиотеку. Например, в сообществе .NET популярен проект Polly (github.com/App-vNext/Polly).

Восстановление после отказа сервиса

Работа с такой библиотекой, как Hystrix, — лишь часть решения. Помимо этого, вы должны определиться с тем, как каждый отдельный сервис должен реагировать на недоступность удаленного сервиса. Один из вариантов — простое возвращение ошибки своему клиенту. Такой подход, к примеру, имеет смысл в ситуации, представленной на рис. 3.2, где запрос на создание заказа оказывается неудачным. В этом случае API-шлюз может сделать лишь одно — вернуть ошибку мобильному клиенту.

В других сценариях может оказаться предпочтительнее вернуть резервное значение, то есть либо значение по умолчанию, либо закешированный ответ. Например, в главе 7 описывается реализация API-шлюзом запрашивающей операции `findOrder()` с использованием шаблона объединения API. Как видно на рис. 3.3, реализованная таким образом конечная точка `GET /orders/{orderId}` вызывает несколько сервисов, включая `Order`, `Kitchen` и `Delivery`, а затем объединяет их ответы.



Рис. 3.3. API-шлюз реализует конечную точку `GET /orders/{orderId}` путем объединения API. Он вызывает несколько сервисов, агрегирует их ответы и возвращает результат мобильному приложению. Код, реализующий конечную точку, должен иметь стратегию на случай отказа любого из сервисов, которые он вызывает

Вполне вероятно, что для клиента данные разных сервисов важны в разной степени. Данные сервиса `Order` оказываются необходимыми. Если этот сервис недоступен, API-шлюз должен вернуть либо закешированную версию его ответа, либо ошибку. Данные других сервисов не так важны. Например, клиент сможет вывести пользователю полезную информацию даже в том случае, если состояние доставки окажется недоступным. Если сервис `Delivery` не отвечает, API-шлюз должен вернуть закешированную версию его данных или вовсе исключить его из ответа.

Умение справляться с частичными отказами должно быть неотъемлемой способностью ваших сервисов, но это не единственная проблема, которую необходимо решить при использовании RPI. Чтобы один сервис мог удаленно вызывать процедуры из другого, он должен знать местоположение конкретного экземпляра сервиса в сети. На первый взгляд здесь нет ничего сложного, но на самом деле это довольно непростая проблема. Вы должны задействовать механизм обнаружения сервисов. Посмотрим, как это работает.

3.2.4. Обнаружение сервисов

Представьте, что вы пишете код, который вызывает сервис через REST API. Для выполнения запроса этому коду необходимо знать местоположение экземпляра сервиса в сети (IP-адрес и порт). В традиционных приложениях, работающих на физическом оборудовании, сетевое местоположение экземпляров сервисов обычно статическое. Ваш код, к примеру, мог бы извлечь сетевые адреса из конфигурационного файла, который время от времени обновляется. Но в современных микросервисных приложениях, основанных на облачных технологиях, все может быть не так просто. Современная система куда более динамичная (рис. 3.4).



Рис. 3.4. Экземпляры сервисов имеют динамически назначаемые IP-адреса

Сетевое местоположение назначается экземплярам сервисов динамически. Более того, набор этих экземпляров постоянно меняется из-за автоматического масштабирования, отказов и обновлений. Из-за этого ваш клиент должен использовать обнаружение сервисов.

Обзор механизмов обнаружения сервисов

Как вы только что видели, клиент нельзя сконфигурировать статически, предоставив ему IP-адреса сервисов. Приложение должно задействовать механизм динамического обнаружения. По своей сути обнаружение сервисов является довольно простым: его ключевым компонентом выступает реестр сервисов — база данных с информацией о том, где находятся экземпляры сервисов приложения.

Когда экземпляры сервисов запускаются и останавливаются, механизм обнаружения обновляет реестр. Когда клиент обращается к сервису, механизм обнаружения получает список его доступных экземпляров, запрашивая реестр, и направляет запрос одному из них.

Есть два основных способа реализации механизма обнаружения сервисов.

- ❑ Сервисы и их клиенты напрямую взаимодействуют с реестром.
- ❑ За обнаружение сервисов отвечает инфраструктура развертывания (подробнее об этом — в главе 12).

Рассмотрим оба эти варианта.

Применение шаблонов обнаружения сервисов на уровне приложения

Один из способов реализации обнаружения сервисов заключается в том, что сервисы приложения и их клиенты взаимодействуют с реестром сервисов. На рис. 3.5 показано, как это работает. Для вызова сервиса клиент сначала обращается к реестру, чтобы получить список его экземпляров, а затем шлет запрос одному из них.

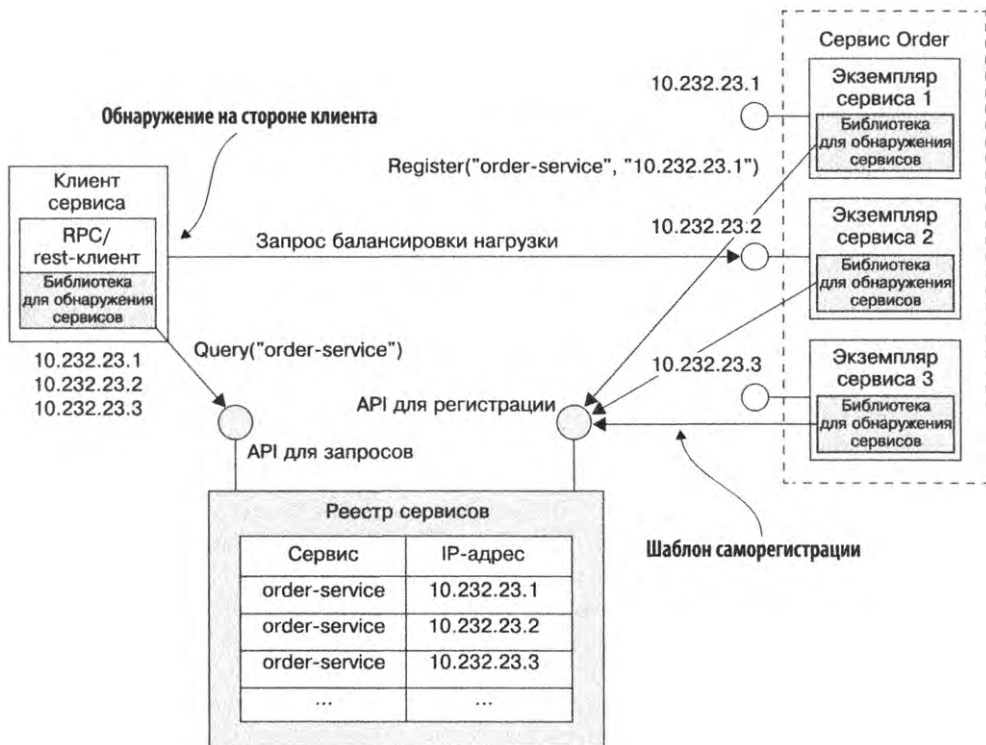


Рис. 3.5. Реестр сервисов отслеживает их экземпляры. Клиенты обращаются к реестру, чтобы найти сетевое местоположение доступных экземпляров сервиса

Данный подход к обнаружению сервисов сочетает в себе два шаблона. Первый — это саморегистрация. Экземпляр сервиса обращается к API реестра, чтобы зарегистрировать свое сетевое местоположение. Он также может предоставить URL-адрес для *проверки работоспособности* (подробнее об этом — в главе 11). Этот URL-адрес является конечной точкой API, которую реестр периодически запрашивает, чтобы убедиться

в том, что сервис работает в нормальном режиме и доступен для обработки запросов. Реестр может требовать, чтобы экземпляр сервиса периодически вызывал API «сердцебиения», который предотвращает истечение срока действия его регистрации.

Шаблон «Саморегистрация»

Экземпляр сервиса регистрирует себя в реестре. См. microservices.io/patterns/self-registration.html.

Вторым шаблоном является обнаружение на клиентской стороне. Когда клиент хочет обратиться к сервису, он обращается к реестру, чтобы получить список его экземпляров. Для улучшения производительности клиент может кэшировать экземпляры сервиса. После этого он использует алгоритм балансирования нагрузки, циклический или случайный, чтобы выбрать конкретный экземпляр и отправить ему запрос.

Шаблон «Обнаружение на клиентской стороне»

Клиент извлекает из реестра список доступных экземпляров сервиса и выбирает один из них с учетом балансирования нагрузки. См. <http://microservices.io/patterns/server-side-discovery.html>.

Обнаружение сервисов на уровне приложения популяризируют компании Netflix и Pivotal. Например, в Netflix были разработаны и выпущены под открытой лицензией несколько компонентов, таких как Eureka (высокодоступный реестр сервисов), Java-клиент для Eureka и Ribbon (сложный HTTP-клиент с поддержкой клиента для Eureka). Компания Pivotal разработала Spring Cloud — фреймворк на основе Spring, который делает использование компонентов Netflix на удивление простым. Сервисы, построенные с помощью Spring Cloud, автоматически регистрируются с помощью Eureka, а клиенты, основанные на Spring Cloud, автоматически применяют Eureka для обнаружения сервисов.

Одно из преимуществ обнаружения сервисов на уровне приложения — то, что в нем предусмотрена возможность развертывания сервисов на разных платформах. Представьте, к примеру, что лишь часть ваших сервисов развернута на Kubernetes (см. главу 12), а остальные работают в устаревшей среде. Обнаружение сервисов на уровне приложения с использованием Eureka будет охватывать обе среды, тогда как решение на базе Kubernetes совместимо лишь с Kubernetes.

Один из недостатков этого подхода связан с тем, что он требует наличия библиотеки обнаружения сервисов для каждого языка, а возможно, и фреймворка, который вы применяете. Spring Cloud может помочь лишь разработчикам на Spring. Если вы используете какой-то другой Java-фреймворк или язык вне платформы JVM, такой как NodeJS или GoLang, вам придется поискать другую библиотеку. Еще одной отрицательной стороной обнаружения сервисов на уровне приложения является то, что настройка и обслуживание реестра ложатся на вас — это будет вас отвлекать.

В связи с этим обычно предпочтительно задействовать механизмы обнаружения, предоставляемые инфраструктурой развертывания.

Применение шаблонов обнаружения сервисов, предоставляемых платформой

Из главы 12 вы узнаете, что многие современные платформы развертывания, такие как Docker и Kubernetes, имеют встроенные реестр и механизм обнаружения сервисов. Платформа развертывания выдает каждому сервису DNS-имя, виртуальный IP-адрес (VIP) и привязанное к нему доменное имя. Клиент делает запрос к DNS-имени/VIP, а платформа развертывания автоматически направляет его к одному из доступных экземпляров сервиса. В итоге регистрация и обнаружение сервисов, а также маршрутизация запросов выполняются самой платформой. Как это работает, показано на рис. 3.6.

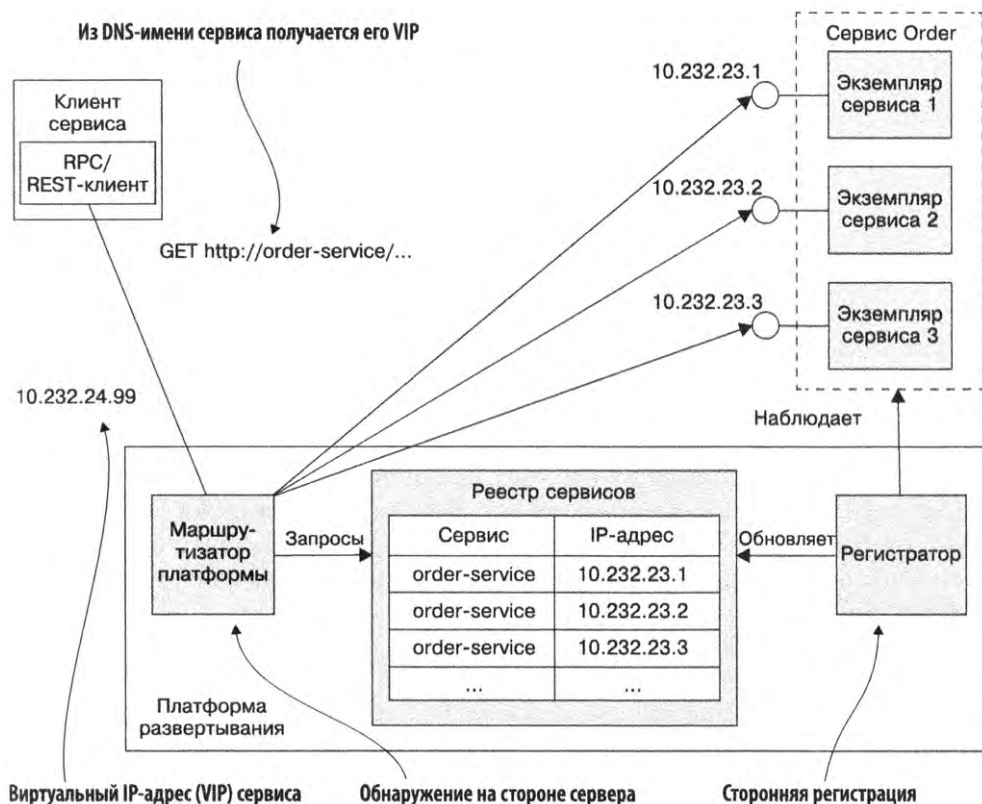


Рис. 3.6. Платформа ответственна за регистрацию сервисов, обнаружение и маршрутизацию запросов. Экземпляры сервисов заносятся в реестр регистратором. У каждого сервиса есть сетевое местоположение, DNS-имя или виртуальный IP-адрес. Клиент выполняет запрос к сетевому местоположению сервиса. Маршрутизатор обращается к реестру и распределяет запросы между всеми доступными сервисами

Платформа развертывания включает в себя реестр с информацией об IP-адресах развернутых сервисов. В данном примере клиент обращается к сервису `Order` по DNS-имени `order-service`, которое направляет к виртуальному IP-адресу `10.1.3.4`. Платформа автоматически распределяет запросы между экземплярами сервиса `Order`.

Этот подход сочетает в себе два шаблона:

- ❑ *сторонней регистрации* — сервис не прописывает себя в реестре сам, за него это делает компонент *регистратор*, который обычно является частью платформы развертывания и отвечает за регистрацию;
- ❑ *обнаружения на стороне сервера* — вместо того чтобы обращаться к реестру, клиент делает запрос к DNS-имени, которое направляет его к маршрутизатору запросов, а тот в свою очередь идет в реестр и балансирует нагрузку.

Шаблон «Сторонняя регистрация»

Экземпляры сервиса автоматически регистрируются в реестре сторонним компонентом. См. microservices.io/patterns/3rd-party-registration.html.

Шаблон «Обнаружение на стороне сервера»

Клиент делает запрос к маршрутизатору, который отвечает за обнаружение сервисов. См. microservices.io/patterns/server-side-discovery.html.

Ключевое преимущество данного подхода состоит в том, что всеми аспектами обнаружения сервисов занимается сама платформа. Ни клиенты, ни сервисы не содержат никакого кода для этой цели. Благодаря этому механизм обнаружения доступен для всех сервисов и клиентов независимо от языка или фреймворка, на которых они написаны.

Один из недостатков метода связан с тем, что он поддерживает обнаружение только тех сервисов, которые были развернуты на данной платформе. Например, как отмечалось при описании обнаружения на уровне приложения, платформа Kubernetes может обнаружить только те сервисы, которые на ней запущены. Несмотря на это ограничение, рекомендую использовать обнаружение сервисов на уровне платформы везде, где это возможно.

Итак, мы рассмотрели синхронную модель IPC на примере REST и gRPC. Теперь поговорим об альтернативе — асинхронном взаимодействии на основе сообщений.

3.3. Взаимодействие с помощью асинхронного обмена сообщениями

Сервисы могут взаимодействовать путем асинхронного обмена сообщениями. Приложения, основанные на этом подходе, обычно используют *брокер сообщений*, который играет роль промежуточного звена между сервисами. Возможна и архитектура без брокера, когда сервисы взаимодействуют между собой напрямую. Клиент шлет сервису запрос в виде сообщения. Если экземпляр сервиса должен ответить, он сделает это, отправив отдельное сообщение клиенту. Поскольку взаимодействие является асинхронным, клиент не блокируется в ожидании ответа — он написан с расчетом на то, что ответ может прийти не сразу.

Шаблон «Обмен сообщениями»

Клиент общается с сервисом посредством асинхронных сообщений. См. microservices.io/patterns/communication-style/messaging.html.

Начну этот раздел с обзора механизмов обмена сообщениями. Я покажу, как сделать архитектуру обмена сообщениями независимой от конкретной технологии. Далее сравню архитектуры с брокером и без, выделю их основные отличия и опишу критерии выбора брокера сообщений. После этого мы обсудим несколько важных тем, таких как масштабирование потребителей с сохранением упорядоченности сообщений, определение и отклонение дубликатов, а также отправка и прием сообщений в рамках транзакции базы данных. Вначале посмотрим, как работает обмен сообщениями.

3.3.1. Обзор механизмов обмена сообщениями

Практичная модель обмена сообщениями описывается в книге Грегора Хона (Gregor Hohpe) и Бобби Вульфа (Bobby Woolf) *Enterprise Integration Patterns* (Addison-Wesley Professional, 2003)¹. В этой модели сообщения передаются по каналам. Отправитель (приложение или сервис) пишет сообщение в канал, а получатель (приложение или сервис) считывает его из этого канала. Начнем с сообщений, а затем перейдем к каналам.

О сообщениях

Сообщение состоит из заголовка и тела (www.enterpriseintegrationpatterns.com/Message.html). *Заголовок* — это набор пар «ключ — значение», метаданные, которые описывают

¹ Хон Г., Вульф Б. Шаблоны интеграции корпоративных приложений. — М.: Вильямс, 2016.

отправляемую информацию. Помимо ключей и значений, предоставляемых отправителем, заголовок содержит такие данные, как *идентификатор сообщения* (предоставляется либо отправителем, либо инфраструктурой) и необязательный *обратный адрес*, в котором указан канал, куда следует записывать ответ. *Тело сообщения* — это отправляемые данные. Они могут иметь текстовый или двоичный формат.

Существует несколько разных видов сообщений.

- ❑ *Документ* — обобщенное сообщение, содержащее только данные. Получатель сам решает, как его интерпретировать. Пример документного сообщения — ответ на команду.
- ❑ *Команда* — сообщение, эквивалентное RPC-запросу. В нем указываются вызываемая операция и ее параметры.
- ❑ *Событие* — сообщение о том, что с отправителем произошло нечто заслуживающее внимания. Событие часто принадлежит к домену и представляет изменение состояния доменного объекта, такого как *Order* или *Customer*.

Подход к микросервисной архитектуре, описанный в этой книге, подразумевает активное использование команд и событий.

Теперь рассмотрим каналы — механизм, посредством которого взаимодействуют сервисы.

О каналах сообщений

Как видно на рис. 3.7, сообщения передаются по каналам (www.enterpriseintegrationpatterns.com/MessageChannel.html). Бизнес-логика отправителя обращается к интерфейсу *исходящего порта*, который инкапсулирует внутренний механизм взаимодействия. *Исходящий порт* реализуется классом-адаптером *отправителя*, который передает сообщение получателю через канал. *Канал сообщений* — это инфраструктурная абстракция. Для обработки сообщения вызывается класс-адаптер *обработчика сообщений*, который обращается к интерфейсу *входящего порта*, реализованному бизнес-логикой потребителя. Записывать сообщения в канал может любое количество отправителей. Точно так же любое количество получателей может их оттуда читать.

Существует два вида каналов: «точка — точка» (www.enterpriseintegrationpatterns.com/PointToPointChannel.html) и «издатель — подписчик» (www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html).

- ❑ Канал типа «точка — точка» доставляет сообщения ровно одному потребителю, который считывает их оттуда. Сервисы используют такие каналы для взаимодействия вида «один к одному», описанного ранее. Например, по каналам «точка — точка» часто передают командные сообщения.
- ❑ Канал типа «издатель — подписчик» доставляет каждое сообщение всем подключенным потребителям. Сервисы применяют такие каналы для взаимодействия вида «один ко многим», описанного ранее. Например, по каналам «издатель — подписчик» обычно рассылают события.

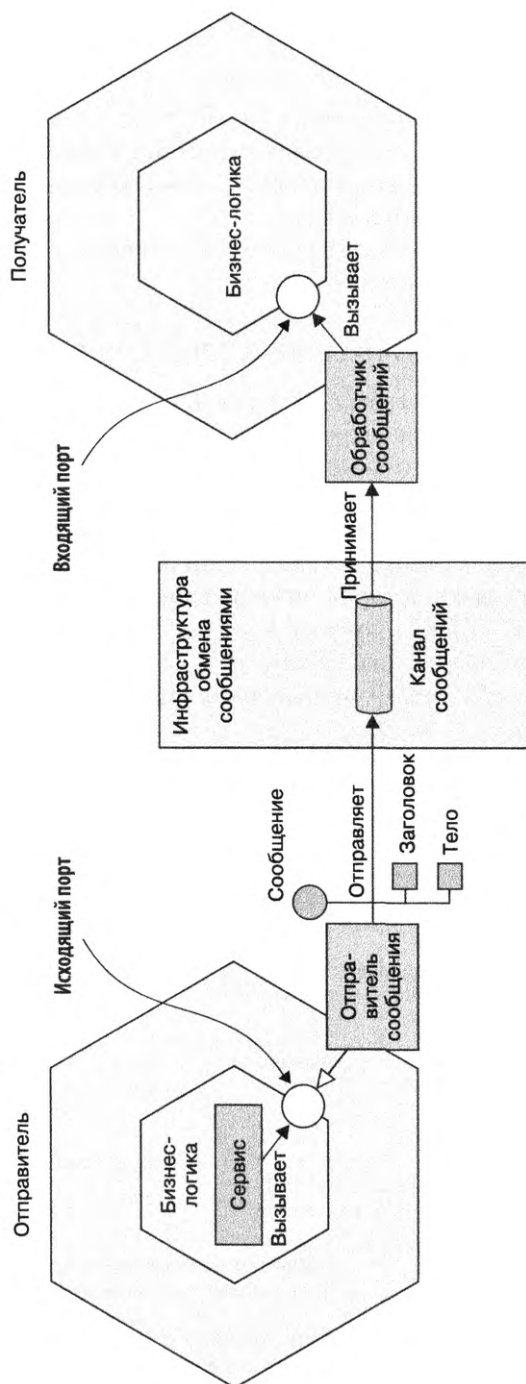


Рис. 3.7. Бизнес-логика отправителя обращается к интерфейсу исходящего порта, который реализуется адаптером отправителя сообщения. Отправитель шлет сообщение получателю через канал. Канал сообщений — это абстракция инфраструктуры обмена сообщениями. Адаптер обработчика сообщений в получателе вызывается для обработки сообщения. Он обращается к интерфейсу входящего порта, который реализуется бизнес-логикой получателя

3.3.2. Реализация стилей взаимодействия с помощью сообщений

Одно из полезных свойств механизмов обмена сообщениями — их гибкость, которой достаточно для поддержки всех стилей взаимодействия, описанных в подразделе 3.1.1. Некоторые из этих стилей реализованы напрямую в виде обмена сообщениями, другие строятся поверх этого подхода.

Посмотрим, как реализовать каждый из стилей взаимодействия. Начнем с синхронных и асинхронных запросов/ответов.

Реализация синхронных и асинхронных запросов/ответов

Клиент и сервис могут взаимодействовать между собой, отправляя запросы и принимая ответы. Если это происходит синхронно, клиент ожидает немедленного ответа от сервиса, а если это асинхронное взаимодействие — не ждет. Обмен сообщениями по своей природе асинхронен, поэтому он предоставляет лишь асинхронные запросы/ответы. Но клиент может блокироваться, пока ответ не будет получен.

Чтобы реализовать стиль взаимодействия с асинхронными запросами/ответами, клиент и сервис обмениваются парными сообщениями. Как видно на рис. 3.8, клиент отправляет в канал «точка — точка», принадлежащий сервису, командное сообщение с указанием операции и ее параметров. Сервис обрабатывает запрос и возвращает в канал «точка — точка», принадлежащий клиенту, ответное сообщение с результатом.

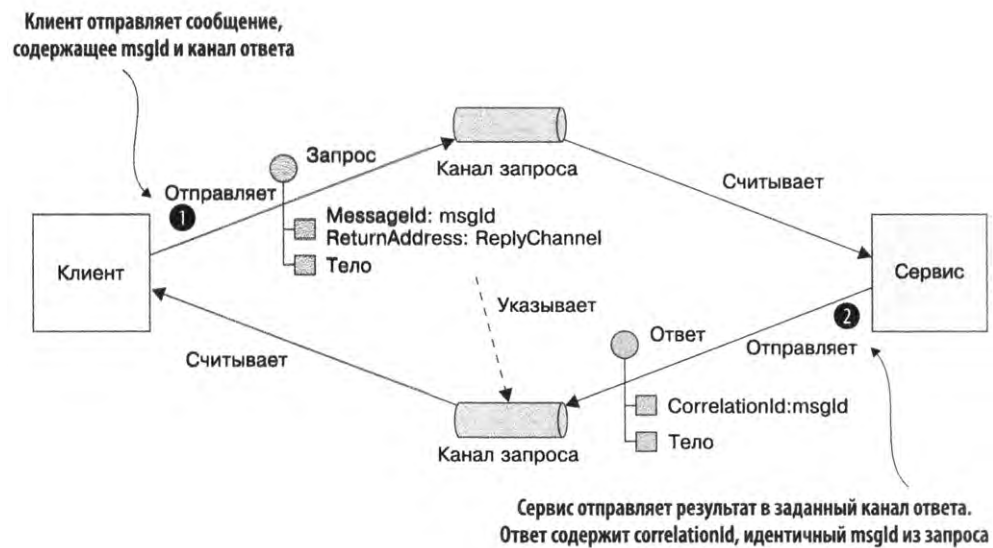


Рис. 3.8. Реализация асинхронных запросов/ответов путем включения ответного канала и идентификатора в исходное сообщение. Получатель обрабатывает сообщение и шлет ответ в заданный канал

Клиент должен сообщить сервису, куда тому следует вернуть результат, и сопоставить ответное сообщение со своим запросом. К счастью, решить эти две задачи несложно. Клиент отправляет командное сообщение с *каналом ответа* в заголовке. Сервер записывает в этот канал свой ответ, содержащий *идентификатор соответствия* с тем же значением, что и *идентификатор запроса*. Клиент использует *идентификатор соответствия*, чтобы сопоставить свое сообщение с ответом.

Поскольку клиент и сервис взаимодействуют с помощью сообщений, их общение изначально асинхронно. Теоретически клиент может заблокироваться, пока не получит ответ, но на практике обработка ответов происходит асинхронно. Кроме того, ответы обычно могут обрабатываться любым экземпляром клиента.

Реализация однонаправленных уведомлений

Реализация однонаправленных уведомлений с помощью асинхронных сообщений — довольно простой процесс. Клиент шлет сообщение (обычно командное) в канал типа «точка — точка», принадлежащий сервису. Сервис подписывается на этот канал и обрабатывает сообщения. Он не возвращает ничего в ответ.

Реализация шаблона «издатель/подписчик»

Обмен сообщениями имеет встроенную поддержку стиля взаимодействия «издатель/подписчик». Клиент публикует в канале типа «издатель — подписчик» сообщение, которое считывается несколькими потребителями. Как описывается в главах 4 и 5, сервисы используют этот стиль взаимодействия для публикации доменных событий, которые представляют изменения в доменных объектах. Сервис, публикующий доменное событие, владеет каналом типа «издатель — подписчик», чье название основано на доменном классе. Например, сервис **Order** публикует события **Order** в канале **Order**, а сервис **Delivery** публикует события **Delivery** в канале **Delivery**. Сервису, который заинтересован в конкретном доменном объекте, достаточно подписаться на соответствующий канал.

Реализация издателя/асинхронных ответов

«Издатель/асинхронные ответы» — это высокоуровневый стиль взаимодействия, который сочетает в себе элементы шаблонов «издатель/подписчик» и «запрос/ответ». Клиент публикует в канале типа «издатель — подписчик» сообщение с *каналом ответа* в заголовке. Потребитель записывает ответное сообщение с *идентификатором соответствия* в канал ответа. Клиент принимает ответы и сверяет их с запросом с помощью *идентификатора соответствия*.

Каждый сервис в вашем приложении, который имеет асинхронный API, будет использовать одну или несколько таких методик реализации. Если у сервиса есть асинхронный API для вызова операций, у него должен быть также канал сообщений для приема запросов. Аналогично сервис, публикующий события, будет задействовать для этого канал событий.

Как отмечалось в подразделе 3.1.2, у API сервиса должна быть спецификация. Посмотрим, как ее можно создать для асинхронного API.

3.3.3. Создание спецификации для API сервиса на основе сообщений

Как показано на рис. 3.9, спецификация асинхронного API сервиса должна содержать имена каналов, а также типы и форматы сообщений, которыми обмениваются в каждом из них. Форматы сообщений должны быть описаны с помощью стандартов JSON, XML или Protobuf. Хотя, в отличие от REST и Open API, здесь нет широко распространенного стандарта для документирования каналов и типов сообщений. Вам придется написать неформальный документ.

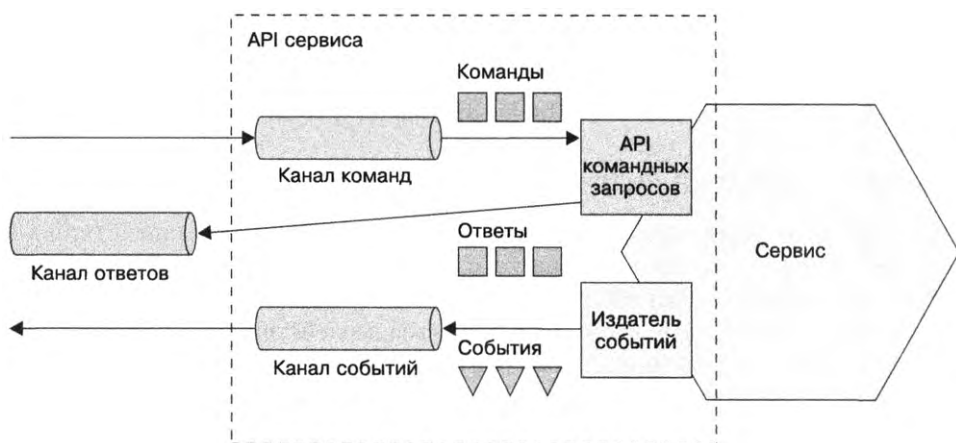


Рис. 3.9. Асинхронный API сервиса состоит из каналов, а также таких типов сообщений, как команды, ответы и события

Асинхронный API состоит из операций, вызываемых клиентами, и событий, которые публикуют сервисы. Эти два аспекта документируются по-разному. Рассмотрим каждый из них, начав с операций.

Документирование асинхронных операций

Операции сервиса можно вызывать с помощью одного из двух стилей взаимодействия.

- ❑ *API в стиле «запрос/асинхронный ответ»* — состоит из канала команд сервиса, типов и форматов командных сообщений, которые сервис принимает, а также типов и форматов ответных сообщений, отправляемых сервисом.
- ❑ *API в стиле однонаправленных уведомлений* — состоит из канала команд сервиса, а также типов и форматов командных сообщений, которые принимает сервис.

Сервис может использовать один и тот же канал как для асинхронных запросов/ответов, так и для однонаправленных уведомлений.

Документирование публикуемых событий

Сервис может публиковать события, используя стиль взаимодействия «издатель/подписчик». Спецификация API такого стиля состоит из канала событий, а также типов и форматов сообщений, которые сервис в нем публикует.

Эта модель обмена сообщениями — отличная абстракция и хороший метод проектирования асинхронных API. Но для создания сервиса вам необходимо выбрать технологию обмена сообщениями и определить, как с помощью ее возможностей реализовать свою архитектуру. Посмотрим, как выглядит этот процесс.

3.3.4. Использование брокера сообщений

Приложения, основанные на сообщениях, обычно применяют *брокер сообщений* — инфраструктурный компонент, через который сервисы общаются друг с другом. Архитектура обмена сообщениями может и не иметь брокера, в этом случае сервисы взаимодействуют между собой напрямую. Оба эти подхода приводятся на рис. 3.10. У них есть разные достоинства и недостатки, но обычно решения с участием брокера оказываются более удачными.

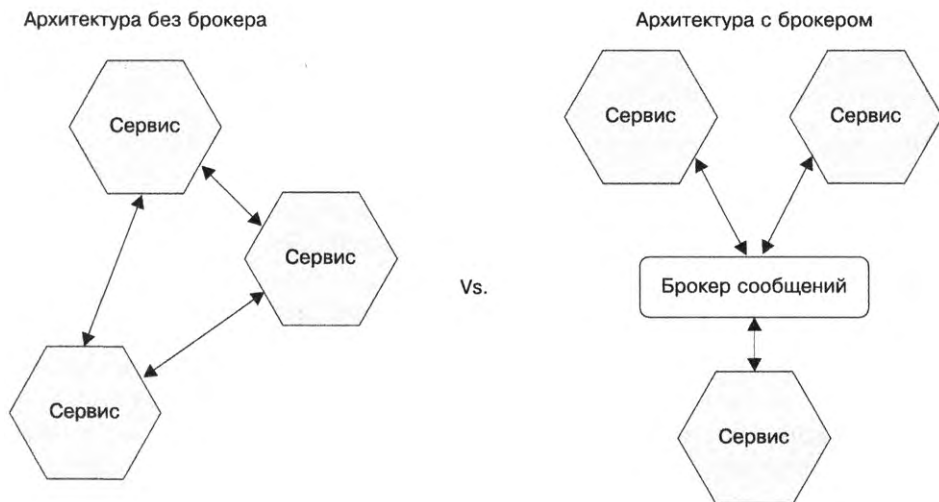


Рис. 3.10. В одной архитектуре сервисы взаимодействуют напрямую, а в другой — через брокер сообщений

В этой книге основное внимание уделяется архитектуре с применением брокера, но будет нелишним взглянуть и на альтернативный подход — в некоторых ситуациях он может вам пригодиться.

Обмен сообщениями без брокера

В архитектуре без брокера сервисы могут обмениваться сообщениями напрямую. Проект ZeroMQ (zeromq.org) — популярная реализация этого подхода. Он представляет собой как спецификацию, так и набор библиотек для разных языков. Проект поддерживает разнообразные механизмы передачи данных, включая TCP, доменные сокеты в стиле UNIX и многоадресное вещание.

Отсутствие брокера дает несколько преимуществ.

- ❑ Более легковесный сетевой трафик и меньшие задержки, поскольку сообщения передаются напрямую от отправителя к получателю и не должны проходить через брокер.
- ❑ Брокер сообщений не станет узким местом или единой точкой отказа.
- ❑ Более простое администрирование, так как вам не нужно настраивать и обслуживать брокер сообщений.

Но какими бы заманчивыми ни были эти преимущества, отсутствие брокера сообщений чревато существенными недостатками.

- ❑ Сервисы должны знать о местонахождении друг друга и, следовательно, использовать один из механизмов обнаружения, описанных в подразделе 3.2.4.
- ❑ Снижена степень доступности, поскольку отправитель и получатель должны оставаться доступными на время передачи сообщения.
- ❑ Возникают дополнительные трудности с реализацией таких механизмов, как гарантированная доставка.

Пониженная доступность и потребность в обнаружении сервисов совпадают с недостатками, присущими синхронным запросам/ответам.

Из-за этих ограничений в большинстве промышленных приложений используется архитектура с брокером. Посмотрим, как это работает.

Обзор обмена сообщениями на основе брокера

Брокер — это промежуточное звено, через которое проходят все сообщения. Отправитель передает сообщение брокеру, а тот доставляет его получателю. Важным преимуществом этого подхода является то, что отправителю не нужно знать сетевое местонахождение потребителя. Кроме того, брокер буферизирует сообщения, пока у получателя не появится возможность их обработать.

Существует много разных брокеров сообщений. Далее перечислены популярные проекты с открытым исходным кодом:

- ❑ ActiveMQ (activemq.apache.org);
- ❑ RabbitMQ (www.rabbitmq.com);
- ❑ Apache Kafka (kafka.apache.org).

Есть также облачные решения для обмена сообщениями, такие как AWS Kinesis (<https://aws.amazon.com/ru/kinesis/>) и AWS SQS (<https://aws.amazon.com/ru/sqs/>).

При выборе брокера сообщений следует учитывать различные факторы, включая следующие.

- ❑ *Поддерживаемые языки программирования.* Лучше выбрать брокер с поддержкой широкого диапазона языков программирования.
- ❑ *Поддерживаемые стандарты обмена сообщениями.* Поддерживает ли брокер сообщений стандарт вроде AMQP или STOMP? Использует ли он свой закрытый протокол?
- ❑ *Порядок следования сообщений.* Сохраняет ли брокер порядок следования сообщений?
- ❑ *Гарантии доставки.* Какие гарантии доставки дает брокер сообщений?
- ❑ *Постоянное хранение.* Сохраняются ли сообщения на диск? Могут ли они пережить сбой брокера?
- ❑ *Устойчивость.* Если потребитель переподключится к брокеру, получит ли он сообщения, отправленные, пока он был отключен?
- ❑ *Масштабируемость.* Насколько масштабируем брокер сообщений?
- ❑ *Латентность.* Какова сквозная латентность?
- ❑ *Конкурирующие потребители.* Поддерживает ли брокер сообщений конкурирующих потребителей?

У каждого брокера есть свои плюсы и минусы. Например, брокер с низкой латентностью может не сохранять порядок следования сообщений, не гарантировать их доставку и хранить их исключительно в оперативной памяти. Гарантированная доставка и надежное хранение сообщений на диске, скорее всего, будут стоить вам повышенной латентности. То, какой брокер подходит лучше всего, зависит от требований вашего приложения. Возможно даже, что у разных частей системы разные требования к обмену сообщениями.

Хотя, наверное, важнейшими свойствами являются порядок следования и масштабируемость. Давайте поговорим о том, как реализовать каналы сообщений с помощью брокера.

Реализация каналов сообщений с помощью брокера

Все брокеры сообщений по-своему реализуют концепцию каналов (табл. 3.2). JMS-брокеры, такие как ActiveMQ, имеют очереди и темы. Брокеры, основанные на AMQP, как RabbitMQ, поддерживают обмены и очереди. У Apache Kafka есть темы, у AWS Kinesis — потоки, а у AWS SQS — те же очереди. Более того, некоторые из них обеспечивают более гибкий обмен сообщениями по сравнению с абстрактной концепцией каналов, описанной в этой главе.

Таблица 3.2. Брокеры сообщений реализуют концепцию каналов по-разному

Брокер сообщений	Канал типа «точка — точка»	Канал типа «издатель — подписчик»
JMS	Очередь	Тема
Apache Kafka	Тема	Тема
Брокеры на основе AMQP, такие как RabbitMQ	Обмен + Очередь	Обмен типа fanout и отдельная очередь для каждого потребителя
AWS Kinesis	Поток	Поток
AWS SQS	Очередь	—

Почти все брокеры сообщений, описанные здесь, одновременно поддерживают каналы типа «точка — точка» и «издатель — подписчик». Исключение — проект AWS SQS, который поддерживает только каналы «точка — точка».

Теперь пройдемся по положительным и отрицательным сторонам обмена сообщениями на основе брокера.

Преимущества и недостатки обмена сообщениями на основе брокера

Обмен сообщениями на основе брокера имеет множество преимуществ.

- ❑ *Слабая связанность.* Для выполнения запроса клиенту нужно лишь отправить сообщение в подходящий канал. Клиенту ничего не известно об экземплярах сервиса, и ему не нужно использовать механизм обнаружения, чтобы определить их местонахождение.
- ❑ *Буферизация сообщений.* Брокер буферизирует сообщения до тех пор, пока их не смогут обработать. В протоколах с синхронными запросами/ответами, таких как HTTP, и клиент, и сервис должны быть доступны на протяжении всего обмена данными. Сообщения же накапливаются в очереди, пока потребитель не будет готов их принять. Это означает, что, к примеру, онлайн-магазин может принимать заказы от посетителей, даже если система выполнения заказов слишком медленная или недоступна. Сообщения просто будут ожидать в очереди, пока их не смогут обработать.
- ❑ *Гибкое взаимодействие.* Обмен сообщениями поддерживает все стили взаимодействия, описанные ранее.
- ❑ *Явное межпроцессное взаимодействие.* Механизмы, основанные на RPC, пытаются сделать так, чтобы обращение к удаленному сервису выглядело словно вызов локальной процедуры. Но законы физики и вероятность частичных отказов делают эти два вида взаимодействия очень разными. Обмен сообщениями делает различия явными, чтобы у разработчиков не возникало ложное чувство безопасности.

У обмена сообщениями есть и некоторые недостатки.

- ❑ *Потенциальное узкое место производительности.* Существует риск того, что брокер сообщений может стать узким местом производительности. Хорошо, что многие современные брокеры спроектированы с поддержкой высокой масштабируемости.
- ❑ *Потенциальная единая точка отказа.* Крайне важно, чтобы брокер сообщений был высокодоступным, иначе может пострадать надежность системы. К счастью, большинство современных брокеров спроектированы с поддержкой высокой доступности.
- ❑ *Дополнительная сложность в администрировании.* Механизм обмена сообщениями — это еще один системный компонент, который нужно устанавливать, конфигурировать и администрировать.

Рассмотрим некоторые архитектурные проблемы, с которыми вы можете столкнуться.

3.3.5. Конкурирующие получатели и порядок следования сообщений

Одна из трудностей связана с тем, как масштабировать получателей и сохранить при этом порядок следования сообщений. Наличие нескольких экземпляров сервиса для параллельной обработки сообщений является распространенным требованием. Более того, даже один экземпляр, вероятно, будет использовать потоки, чтобы обрабатывать сообщения параллельно. Применение нескольких потоков и экземпляров сервиса повышает пропускную способность приложения. Но из-за конкурентной работы сложно гарантировать, что каждое сообщение будет обработано лишь один раз и в правильном порядке.

Представьте, например, что у вас есть три экземпляра сервиса, которые читают из одного канала типа «точка — точка», и издатель последовательно публикует события `Order Created`, `Order Updated` и `Order Cancelled`. Примитивная реализация могла бы параллельно доставить каждое сообщение своему получателю. Но по причине задержек, связанных с проблемами в сети или со сборкой мусора, сообщения могут быть обработаны не в том порядке, что обусловит неожиданное поведение. Теоретически экземпляр сервиса может обработать сообщения `Order Cancelled` до того, как другой экземпляр обработает `Order Created`!

Распространенное решение, применяемое в таких современных брокерах сообщений, как Apache Kafka и AWS Kinesis, состоит в использовании *сегментированных* (разделенных) каналов. На рис. 3.11 показано, как это работает. Решение состоит из трех частей.

1. Сегментированный канал включает в себя два и более сегмента, каждый из которых сам ведет себя как канал.

2. Отправитель указывает в заголовке сообщения ключ сегмента, который обычно представляет собой произвольную строку или последовательность байтов. Брокер использует этот ключ, чтобы привязать сообщение к определенному сегменту/разделу. Например, он может выбрать сегмент взятием остатка от целочисленного деления хеша сегментного ключа на количество сегментов.
3. Брокер сообщений группирует экземпляры получателя и обращается с ними как с одним логическим получателем. В Apache Kafka, например, применяется термин «*группа потребителей*». Брокер сообщений назначает каждый сегмент отдельному получателю. При запуске и остановке получателей эта процедура повторяется.

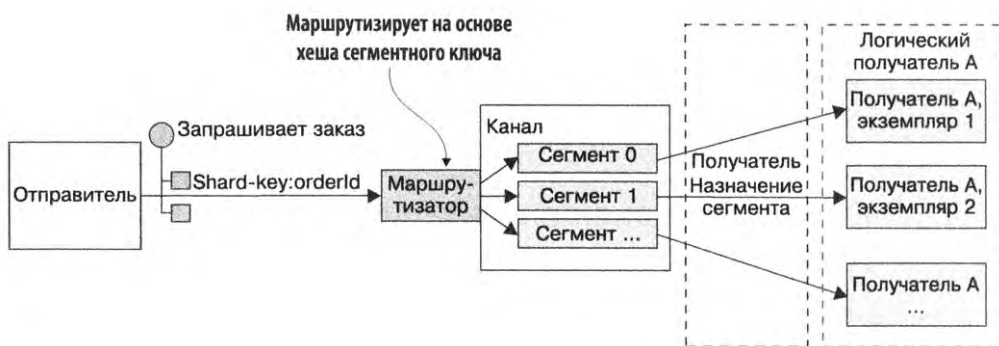


Рис. 3.11. Масштабирование потребителей с использованием сегментированного (разделенного) канала, чтобы сохранить порядок следования сообщений. Отправитель включает в сообщение ключ сегмента. Брокер записывает сообщение в сегмент, определенный на основе этого ключа, и назначает каждый раздел экземпляру реплицированного получателя

В этом примере каждое событие `Order` содержит ключ сегмента в качестве `orderId`. Каждое событие, связанное с конкретным заказом, публикуется в один и тот же сегмент, который считывается одним экземпляром потребителя. В итоге гарантируется упорядоченная обработка этих сообщений.

3.3.6. Дублирование сообщений

Еще одна проблема, которую необходимо решить при обмене сообщениями, связана с дубликатами. В идеале брокер должен доставлять каждое сообщение ровно один раз, но обеспечение таких гарантий обычно оказывается слишком затратным. Вместо этого большинство брокеров обещают доставить сообщение *как минимум* один раз.

Когда система работает в нормальном режиме, каждое сообщение доставляется по одному разу. Но отказ клиента, сети или брокера сообщений может вызывать множественную доставку. Допустим, клиент отказывает после обработки сообщения и обновления базы данных, но перед подтверждением сообщения. Тогда брокер

доставит неподтвержденное сообщение повторно — либо тому же клиенту, когда он перезапустится, либо копии другого клиента.

В идеале вы должны использовать брокер сообщений, который сохраняет порядок следования при повторной доставке. Представьте, что клиент последовательно обрабатывает события `Order Created` и `Order Cancelled` для одного и того же заказа, но по какой-то причине событие `Order Created` не было подтверждено. В этом случае брокер должен заново доставить оба сообщения, `Order Created` и `Order Cancelled`. Если он повторно отправит лишь `Order Created`, клиент может проигнорировать отмену заказа.

Существует несколько методов работы с повторяющимися сообщениями:

- ☐ добавление в сообщения идемпотентных дескрипторов;
- ☐ отслеживание и отклонение дубликатов.

Рассмотрим оба варианта.

Добавление в сообщения идемпотентных дескрипторов

Если программная логика, обрабатывающая сообщения, является идемпотентной, дубликаты не несут в себе никакой опасности. Логика считается *идемпотентной*, если ее многократное выполнение с идентичными входными значениями не имеет дополнительных эффектов. Например, отмена уже отмененного заказа — это идемпотентная операция. То же самое относится к созданию заказа с ID, который предоставляется клиентом. Идемпотентный обработчик сообщений можно безопасно вызвать несколько раз при условии, что брокер сохраняет порядок следования во время повторной доставки.

К сожалению, программная логика не всегда оказывается идемпотентной. Или же ваш брокер сообщений не сохраняет порядок следования при повторной доставке. Дубликаты и сообщения, доставленные не в том порядке, могут привести к ошибкам. В таких случаях ваши обработчики должны отслеживать сообщения и отклонять дубликаты.

Отслеживание сообщений и отклонение дубликатов

Возьмем, к примеру, обработчик сообщений, который авторизует банковскую карту клиента. Для каждого заказа он должен выполнить ровно одну авторизацию. Такая программная логика будет иметь разный эффект при каждом последующем вызове. Если за-за продублированных сообщений она выполнится несколько раз, приложение поведет себя некорректно. Обработчик сообщений, реализующий эту логику, обязан быть идемпотентным, для этого он должен определять и отклонять повторяющиеся сообщения.

В качестве простого решения можно сделать так, чтобы потребитель отслеживал обработанные сообщения с помощью идентификаторов и отклонял любые дубликаты. Например, он может сохранять в базе данных идентификатор каждого

сообщения, которое обработал. Как это сделать с использованием отдельной таблицы БД, показано на рис. 3.12.

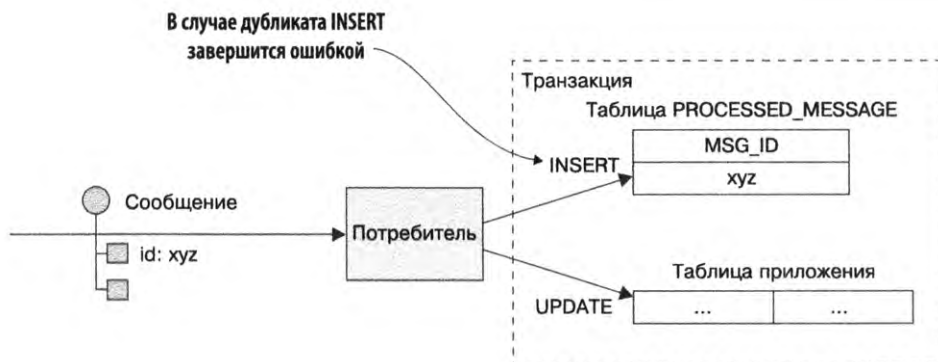


Рис. 3.12. Потребитель определяет и отклоняет повторяющиеся сообщения, записывая в таблицу базы данных идентификаторы уже обработанных. Если сообщение было обработано прежде, операция INSERT в таблице PROCESSED_MESSAGES завершится неудачно

Когда потребитель обрабатывает сообщение, он записывает его ID в таблицу базы данных как часть транзакции по созданию и обновлению бизнес-объектов. В этом примере потребитель вставляет строку с идентификатором сообщения в таблицу PROCESSED_MESSAGES. Если происходит дублирование, операция INSERT завершится неудачно и потребитель сможет отклонить сообщение.

Еще один вариант состоит в том, что обработчик записывает идентификаторы сообщений не в отдельную таблицу, а в таблицу приложения. Этот подход особенно полезен при использовании баз данных NoSQL, которые имеют ограниченную транзакционную модель и не поддерживают обновление двух таблиц в рамках одной транзакции. Пример приведен в главе 7.

3.3.7. Транзакционный обмен сообщениями

Сервису часто нужно публиковать сообщения в рамках транзакции, обновляющей базу данных. На страницах этой книги вы найдете примеры того, как сервисы публикуют доменные события при каждом обновлении или создании бизнес-объектов. Обновление базы данных и отправка сообщения должны происходить в пределах одной транзакции, иначе сервис может обновить БД и, например, отказать до того, как сообщение будет отправлено. Если не выполнять эти две операции атомарно, сбой может оставить систему в несогласованном состоянии.

Транзакционное решение подразумевает использование распределенных транзакций, которые охватывают как БД, так и брокер сообщений. Но, как вы увидите в главе 4, распределенные транзакции — неудачное решение для современных приложений. Кроме того, они не поддерживаются во многих современных брокерах, таких как Apache Kafka.

В связи с этим приложения должны задействовать другой механизм для надежной публикации сообщений. Рассмотрим его.

Использование таблицы базы данных в качестве очереди сообщений

Представьте, что ваше приложение применяет реляционную базу данных. Простой способ надежной публикации сообщений — с помощью шаблона «Публикация событий». Он использует таблицу БД в качестве временной очереди сообщений. У сервиса, отправляющего сообщения, есть таблица **OUTBOX** (рис. 3.13). В рамках транзакции, которая создает, обновляет и удаляет бизнес-объекты, сервис шлет сообщения, вставляя их в эту таблицу. Поскольку это локальная ACID-транзакция, атомарность гарантируется.

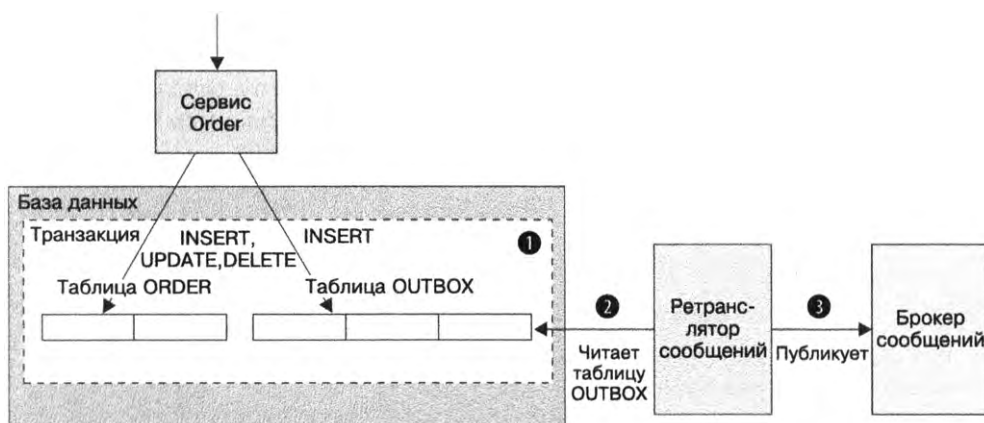


Рис. 3.13. Для надежной публикации сообщения сервис вставляет его в таблицу **OUTBOX** в рамках транзакции по обновлению базы данных. Ретранслятор сообщений считывает таблицу **OUTBOX** и передает сообщения брокеру

Таблица **OUTBOX** играет роль временной очереди сообщений. *Ретранслятор* (**MessageRelay**) — это компонент, который читает таблицу **OUTBOX** и передает сообщения брокеру.

Шаблон «Публикация событий»

Публикует событие или сообщение в рамках транзакции БД, сохраняя его в таблицу **OUTBOX**. См. microservices.io/patterns/data/transactional-outbox.html.

Аналогичный подход можно применять и к некоторым базам данных NoSQL. Каждый бизнес-объект, хранящийся в виде записи внутри БД, имеет атрибут со

списком сообщений, которые нужно опубликовать. Обновляя этот объект, сервис добавляет в список новое сообщение. Это атомарная операция, поскольку она выполняется за один запрос к базе данных. Трудность данного подхода связана с эффективным поиском бизнес-объектов, содержащих события, и их публикацией.

Существует несколько способов доставки сообщений от базы данных к брокеру. Рассмотрим каждый из них.

Публикация событий с помощью шаблона «Опрашивающий издатель»

Если приложение использует реляционную базу данных, сообщения, вставленные в таблицу **OUTBOX**, можно опубликовать очень простым способом: ретранслятору достаточно запросить из таблицы неопубликованные записи. Таблица периодически опрашивается:

```
SELECT * FROM OUTBOX ORDERED BY ... ASC
```

Затем ретранслятор публикует эти сообщения брокеру, отправляя их по соответствующим каналам. В конце он удаляет сообщения из таблицы **OUTBOX**:

```
BEGIN  
DELETE FROM OUTBOX WHERE ID in (...)  
COMMIT
```

Шаблон «Опрашивающий издатель»

Публикует сообщения, опрашивая таблицу **OUTBOX** в базе данных. См. microservices.io/patterns/data/polling-publisher.html.

Опрос базы данных — это простой подход, который неплохо работает в небольших масштабах. Его недостатком является то, что частое обращение к БД может оказаться затратным. К тому же его можно использовать только с теми базами данных NoSQL, которые поддерживают соответствующие запросы. Это связано с тем, что вместо таблицы **OUTBOX** приложению приходится запрашивать бизнес-объекты, а это не всегда можно сделать эффективно. Часто из-за этих недостатков и ограничений предпочтительнее (а в некоторых случаях необходимо) задействовать более тонкий и производительный метод — отслеживание транзакционного журнала.

Публикация событий с применением шаблона «Отслеживание транзакционного журнала»

Менее тривиальное решение заключается в том, что ретранслятор *отслеживает* транзакционный журнал базы данных, который называют еще журналом фиксации. Каждое зафиксированное обновление, выполненное приложением, представлено в виде записи в журнале транзакций БД. Вы можете прочитать этот журнал и опу-

бликовать каждое изменение в качестве сообщения для брокера. Принцип работы этого подхода показан на рис. 3.14.

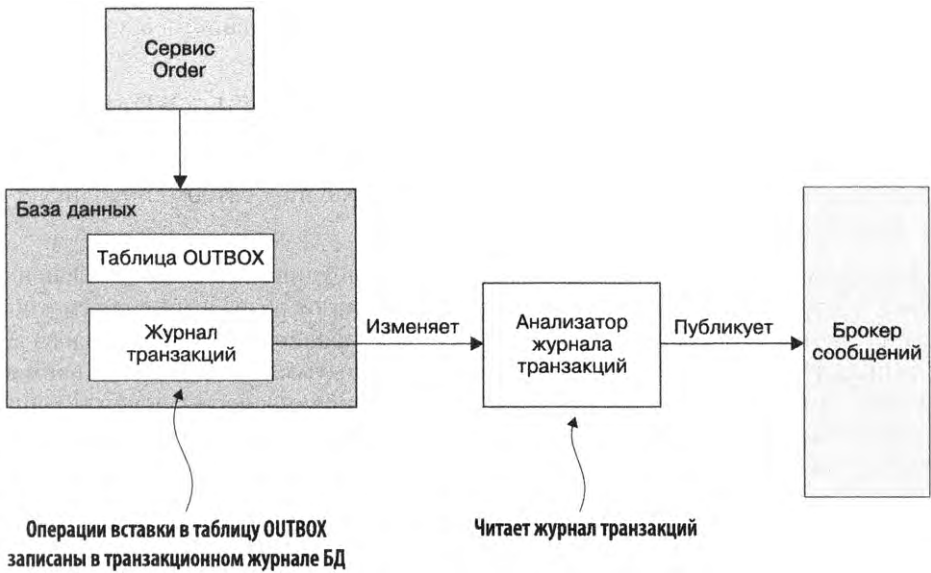


Рис. 3.14. Сервис публикует сообщения, вставленные в таблицу OUTBOX, извлекая информацию из транзакционного журнала БД

Анализатор журнала транзакций читает записи в транзакционном журнале. Каждую подходящую запись, которая соответствует добавлению сообщения, он преобразует в событие и публикует его для брокера. Этот подход годится для публикации сообщений, записанных в таблицу СУБД или добавленных в список записей в базе данных NoSQL.

Шаблон «Отслеживание транзакционного журнала»

Публикует изменения, внесенные в базу данных, отслеживая журнал транзакций.
См. microservices.io/patterns/data/transaction-log-tailing.html.

Существует несколько примеров реализации этого подхода.

- ❑ *Debezium* (debezium.io) — проект с открытым исходным кодом, который публикует изменения базы данных для брокера сообщений Apache Kafka.
- ❑ *LinkedIn Databus* (github.com/linkedin/databus) — проект с открытым исходным кодом, который анализирует журнал транзакций Oracle и публикует изменения в виде событий. Компания LinkedIn использует Databus для синхронизации различных производных источников данных с системой записей.

- ❑ *DynamoDB streams* (docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html) — создает поток упорядоченных по времени изменений (создание, обновление и удаление), произошедших с записями своих таблиц за последние 24 часа. Приложение может читать эти изменения из потока и, например, публиковать их в виде событий.
- ❑ *Eventuate Tram* (github.com/eventuate-tram/eventuate-tram-core) — моя собственная библиотека с открытым исходным кодом для транзакционного обмена сообщениями, которая использует протокол двоичного журнала MySQL, Postgres WAL или просто проверяет изменения, внесенные в таблицу `OUTBOX`, и публикует их в Apache Kafka.

Это не самый простой подход, но работает он на удивление хорошо. Основная трудность состоит в том, что его реализация требует от разработчиков некоторых усилий. Вы могли бы, к примеру, написать низкоуровневый код для вызова API баз данных. Или, как вариант, воспользоваться открытым фреймворком, таким как Debezium, который публикует в Apache Kafka изменения, вносимые приложением в MySQL, Postgres или MongoDB. Недостатком Debezium является то, что он предназначен для перехвата изменений на уровне БД и API для отправки и получения сообщений находятся вне его зоны ответственности. В связи с этим я создал фреймворк Eventuate Tram, который предоставляет API для обмена сообщениями, а также отслеживания и опрашивания транзакционного журнала.

3.3.8. Библиотеки и фреймворки для обмена сообщениями

Для отправки и получения сообщений сервис должен использовать какую-то библиотеку. Это может быть клиентская библиотека брокера сообщений, хотя непосредственное ее применение чревато несколькими проблемами.

- ❑ Клиентская библиотека привязывает бизнес-логику, публикующую сообщения, к API брокера.
- ❑ Клиентские библиотеки брокеров обычно низкоуровневые, поэтому на отправку и получение сообщений потребуется много строчек кода. Разработчики предпочитают не писать один и тот же шаблонный код по многу раз. К тому же, как автор этой книги, я не хочу засорять свои примеры низкоуровневым кодом.
- ❑ Клиентские библиотеки обычно предоставляют лишь базовый механизм отправки и получения сообщений, без поддержки высокоуровневых стилей взаимодействия.

Лучше было бы использовать более высокоуровневые библиотеки или фреймворки, которые скрывают низкоуровневые детали и непосредственно поддерживают высокоуровневые стили взаимодействия. Для простоты в примерах в этой книге задействуется мой фреймворк Eventuate Tram. Он имеет простой и понятный API, который скрывает от вас сложности использования брокера сообщений. Помимо

интерфейса для отправки и получения сообщений Eventuate Tram поддерживает также высокоуровневые стили взаимодействия, такие как асинхронные запросы/ответы и публикация доменных событий.

Что?! Почему именно фреймворки Eventuate?

Примеры кода в этой книге используют разработанные мной фреймворки Eventuate для транзакционного обмена сообщениями, порождения событий и повествований. Я решил взять собственный проект, поскольку, в отличие от, скажем, внедрения зависимостей или фреймворка Spring, многие возможности микросервисной архитектуры не имеют общепринятых реализаций. Без фреймворка Eventuate Tram во многих примерах пришлось бы напрямую применять низкоуровневые API для обмена сообщениями, что сделало бы код намного более сложным и отвлекло от важных концепций. Я мог бы также воспользоваться малоизвестным фреймворком, что тоже вызвало бы критику.

Вместо этого в примерах используется фреймворк Eventuate Tram, который имеет простой и понятный API, скрывающий подробности реализации. Вы можете применять Eventuate Tram в своих приложениях. Также можете изучить содержимое этого фреймворка и самостоятельно реализовать те же концепции.

Eventuate Tram реализует два важных механизма.

- ❑ *Транзакционный обмен сообщениями* — публикует сообщения в рамках транзакции базы данных.
- ❑ *Обнаружение дубликатов* — потребитель сообщений в Eventuate Tram обнаруживает и отклоняет повторяющиеся сообщения. Это очень важно, потому что, как обсуждалось в подразделе 3.3.6, потребитель должен обработать каждое сообщение ровно один раз.

Рассмотрим API Eventuate Tram.

Простой обмен сообщениями

Для простого обмена сообщениями предусмотрено два Java-интерфейса: `MessageProducer` и `MessageConsumer`. Сервис-отправитель использует `MessageProducer` для публикации сообщений в канале. Вот пример работы с этим интерфейсом:

```
MessageProducer messageProducer = ...;
String channel = ...;
String payload = ...;
messageProducer.send(destination, MessageBuilder.withPayload(payload).build())
```

Сервис-потребитель применяет интерфейс `MessageConsumer`, чтобы подписаться на сообщения:

```
MessageConsumer messageConsumer;
messageConsumer.subscribe(subscriberId, Collections.singleton(destination),
    message -> { ... })
```


`MessageProducer` и `MessageConsumer` являются фундаментом высокоуровневых API для асинхронных запросов/ответов и публикации доменных событий.

Поговорим о том, как публикуются события и как на них подписаться.

Публикация доменных событий

У Eventuate Tram есть API для публикации и потребления доменных событий. В главе 5 объясняется, что доменными являются события, которые генерируются *агрегатом* (бизнес-объектом) при его создании, обновлении или удалении. Сервис публикует доменные события с помощью интерфейса `DomainEventPublisher`, например:

```
DomainEventPublisher domainEventPublisher;  
  
String accountId = ...;  
  
DomainEvent domainEvent = new AccountDebited(...);  
  
domainEventPublisher.publish("Account", accountId, Collections.singletonList(  
    domainEvent));
```

Сервис потребляет доменные события с помощью класса `DomainEventDispatcher`. Пример показан далее:

```
DomainEventHandlers domainEventHandlers = DomainEventHandlersBuilder  
    .forAggregateType("Order")  
    .onEvent(AccountDebited.class, domainEvent -> { ... })  
    .build();  
  
new DomainEventDispatcher("eventDispatcherId",  
    domainEventHandlers,  
    messageConsumer);
```

События не единственный высокоуровневый шаблон обмена сообщениями, который поддерживает Eventuate Tram. Можно использовать также сообщения вида «команда/ответ».

Сообщения вида «команда/ответ»

Клиент может послать сервису командное сообщение, используя интерфейс `CommandProducer`, например:

```
CommandProducer commandProducer = ...;  
  
Map<String, String> extraMessageHeaders = Collections.emptyMap();  
  
String commandId = commandProducer.send("CustomerCommandChannel",  
    new DoSomethingCommand(),  
    "ReplyToChannel",  
    extraMessageHeaders);
```

Для потребления командных сообщений сервис использует класс `CommandDispatcher`, который подписывается на определенные события с помощью интерфейса `MessageConsumer`. Он передает каждое командное сообщение подходящему методу-обработчику. Далее приведен пример:

```
CommandHandlers commandHandlers =CommandHandlersBuilder
    .fromChannel(commandChannel)
    .onMessage(DoSomethingCommand.class, (command) -
        > { ... ; return withSuccess(); })
    .build();
CommandDispatcher dispatcher = new CommandDispatcher("subscribeId",
    commandHandlers, messageConsumer, messageProducer);
```

По мере чтения книги вы будете встречать примеры кода, которые используют эти API для отправки и получения сообщений.

Как вы можете увидеть сами, фреймворк Eventuate Tram реализует транзакционный обмен сообщениями для Java-приложений. Он предоставляет два вида интерфейсов: низкоуровневый для отправки и получения сообщений транзакционным способом и высокоуровневые для публикации и потребления доменных событий, а также обработки команд.

Теперь обсудим подход к проектированию сервисов, который улучшает доступность за счет асинхронного обмена сообщениями.

3.4. Использование асинхронного обмена сообщениями для улучшения доступности

Как вы видели, разнообразные механизмы IPC подталкивают вас к различным компромиссам. Один из них связан с тем, как механизм IPC влияет на доступность. В этом разделе вы узнаете, что синхронное взаимодействие с другими сервисами в рамках обработки запросов снижает степень доступности приложения. В связи с этим при проектировании своих сервисов вы должны по возможности использовать асинхронный обмен сообщениями.

Сначала посмотрим, какие проблемы создает синхронное взаимодействие и как это сказывается на доступности.

3.4.1. Синхронное взаимодействие снижает степень доступности

REST — это чрезвычайно популярный механизм IPC. У вас может возникнуть соблазн использовать его для межсервисного взаимодействия. Но проблема REST заключается в том, что это синхронный протокол: HTTP-клиенту приходится ждать, пока сервис не вернет ответ. Каждый раз, когда сервисы общаются между собой по синхронному протоколу, это снижает доступность приложения.

Чтобы понять, почему так происходит, рассмотрим сценарий, представленный на рис. 3.15. У сервиса `Order` есть интерфейс REST API для создания заказов. Для проверки заказа он обращается к сервисам `Consumer` и `Restaurant`, которые тоже имеют REST API.

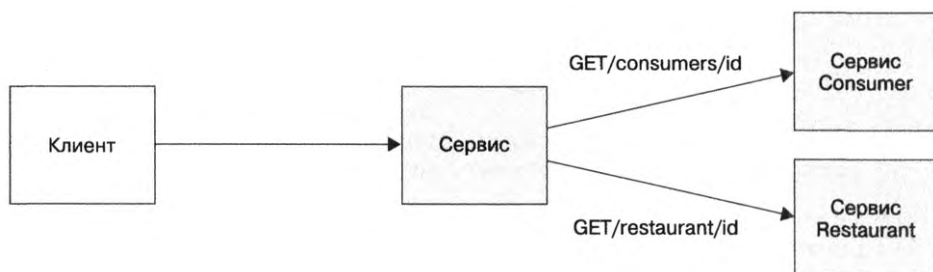


Рис. 3.15. Сервис `Order` обращается к другим сервисам по REST. Это простой способ, но он требует, чтобы все сервисы были доступны одновременно, а это понижает доступность API

Создание заказа состоит из такой последовательности шагов.

1. Клиент делает HTTP-запрос `POST /orders` к сервису `Order`.
2. Сервис `Order` извлекает информацию о заказчике, выполняя HTTP-запрос `GET /consumers/id` к сервису `Consumer`.
3. Сервис `Order` извлекает информацию о ресторане, выполняя HTTP-запрос `GET /restaurant/id` к сервису `Restaurant`.
4. `Order Taking` проверяет запрос, задействуя информацию о заказчике и ресторане.
5. `Order Taking` создает заказ.
6. `Order Taking` отправляет HTTP-ответ клиенту.

Поскольку эти сервисы используют HTTP, все они должны быть доступны, чтобы приложение FTGO смогло обработать запрос `CreateOrder`. Оно не сможет создать заказ, если хотя бы один из сервисов недоступен. С математической точки зрения доступность системной операции является произведением доступности сервисов, которые в нее вовлечены. Если сервис `Order` и те два сервиса, которые он вызывает, имеют доступность 99,5 %, то их общая доступность будет $99,5 \%^3 = 98,5 \%$, что намного ниже. Каждый последующий сервис, участвующий в запросе, делает операцию менее доступной.

Эта проблема не уникальна для взаимодействия на основе REST. Доступность снижается всякий раз, когда для ответа клиенту сервис должен получить ответы от других сервисов. Здесь не поможет даже переход к стилю взаимодействия «запрос/ответ» поверх асинхронных сообщений. Например, если сервис `Order` пошлет сервису `Consumer` сообщение через брокер и примется ждать ответа, его доступность ухудшится.

Если вы хотите максимально повысить уровень доступности, минимизируйте объем синхронного взаимодействия. Посмотрим, как это сделать.

3.4.2. Избавление от синхронного взаимодействия

Существует несколько способов уменьшения объема синхронного взаимодействия с другими сервисами при обработке синхронных запросов. Во-первых, чтобы полностью избежать этой проблемы, все сервисы можно снабдить исключительно асинхронными API. Но это не всегда возможно. Например, публичные API обычно придерживаются стандарта REST. Поэтому некоторые сервисы обязаны иметь синхронные API.

К счастью, чтобы обрабатывать синхронные запросы, вовсе не обязательно выполнять их самому. Поговорим о таких вариантах.

Использование асинхронных стилей взаимодействия

В идеале все взаимодействие должно происходить в асинхронном стиле, описанном ранее в этой главе. Представьте, к примеру, что клиент приложения FTGO применяет для создания заказов асинхронный стиль взаимодействия вида «запрос/асинхронный ответ». Чтобы создать заказ, он отправляет сообщение с запросом сервису Order. Затем этот сервис асинхронно обменивается сообщениями с другими сервисами и в итоге возвращает клиенту ответ (рис. 3.16).

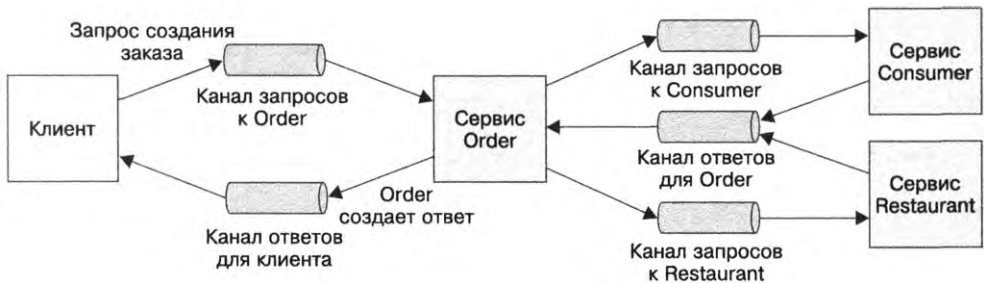


Рис. 3.16. Приложение FTGO окажется более доступным, если его сервисы будут общаться с помощью асинхронных сообщений вместо синхронных вызовов

Клиент и сервис общаются асинхронно, отправляя сообщения через каналы. Ни один из участников этого взаимодействия не блокируется в ожидании ответа.

Такая архитектура была бы чрезвычайно устойчивой, потому что брокер буферизирует сообщения до тех пор, пока их потребление не станет возможным. Но проблема в том, что у сервисов часто есть внешний API, который использует синхронный протокол вроде REST и, как следствие, обязан немедленно отвечать на запросы.

Если у сервиса есть синхронный API, доступность можно улучшить за счет репликации данных. Посмотрим, как это работает.

Репликация данных

Одним из способов минимизации синхронного взаимодействия во время обработки запросов является репликация данных. Сервис хранит копию (реплику) данных, которые ему нужны для обработки запросов. Чтобы поддерживать реплику в актуальном состоянии, он подписывается на события, публикуемые сервисами, которым эти данные принадлежат. Например, сервис `Order` может хранить копию данных, принадлежащих сервисам `Consumer` и `Restaurant`. Это позволит ему обрабатывать запросы на создание заказов, не обращаясь к этим сервисам. Такая архитектура показана на рис. 3.17.

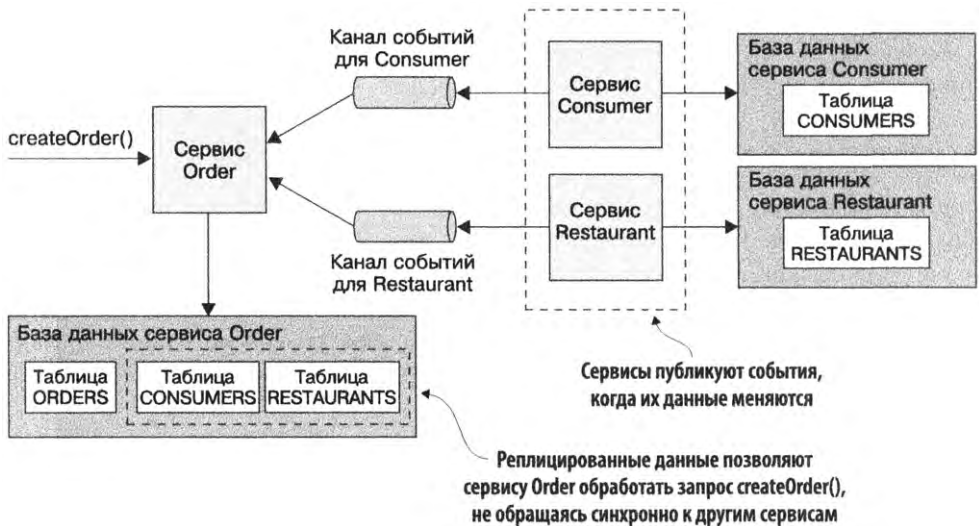


Рис. 3.17. Сервис `Order` автономный, потому что у него есть копии данных о заказчиках и ресторанах

Сервисы `Consumer` и `Restaurant` публикуют события всякий раз, когда их данные меняются. Сервис `Order` подписывается на эти события и обновляет свою реплику.

В некоторых случаях репликация данных — это хорошее решение. Например, в главе 5 описывается, как сервис `Order` реплицирует данные сервиса `Restaurant`, чтобы иметь возможность проверять элементы меню. Один из недостатков этого подхода связан с тем, что иногда он требует копирования больших объемов данных, что неэффективно. Например, если у нас много заказчиков, хранить реплику данных, принадлежащих сервису `Consumer`, может оказаться непрактично. Еще один недостаток репликации кроется в том, что она не решает проблему обновления данных, принадлежащих другим сервисам.

Чтобы решить эту проблему, сервис может отсрочить взаимодействие с другими сервисами до тех пор, пока он не ответит своему клиенту. Речь об этом пойдет далее.

Завершение обработки после возвращения ответа

Еще один способ устранения синхронного взаимодействия во время обработки запросов состоит в том, чтобы выполнять эту обработку в виде следующих этапов.

1. Сервис проверяет запрос только с помощью данных, доступных локально.
2. Он обновляет свою базу данных, в том числе добавляет сообщения в таблицу **OUTBOX**.
3. Возвращает ответ своему клиенту.

Во время обработки запроса сервис не обращается синхронно ни к каким другим сервисам. Вместо этого он шлет им асинхронные сообщения. Данный подход обеспечивает слабую связанность сервисов. Как вы увидите в следующей главе, этот процесс часто реализуется в виде *повествования*.

Представьте, что сервис **Order** действует таким образом. Он создает заказ с состоянием **PENDING** и затем проверяет его, обмениваясь асинхронными сообщениями с другими сервисами. На рис. 3.18 показано, что происходит при вызове операции `createOrder()`. Цепочка событий выглядит так.

1. Сервис **Order** создает заказ с состоянием **PENDING**.
2. Сервис **Order** возвращает своему клиенту ответ с ID заказа.
3. Сервис **Order** шлет сообщение `ValidateConsumerInfo` сервису **Consumer**.
4. Сервис **Order** шлет сообщение `ValidateOrderDetails` сервису **Restaurant**.
5. Сервис **Consumer** получает сообщение `ValidateConsumerInfo`, проверяет, может ли заказчик размещать заказ, и отправляет сообщение `ConsumerValidated` сервису **Order**.
6. Сервис **Restaurant** получает сообщение `ValidateOrderDetails`, проверяет корректность элементов меню и способность ресторана доставить заказ по заданному адресу и отправляет сообщение `OrderDetailsValidated` сервису **Order**.
7. Сервис **Order** получает сообщения `ConsumerValidated` и `OrderDetailsValidated` и меняет состояние заказа на **VALIDATED**.

И так далее...

Сервис **Order** может получить сообщения `ConsumerValidated` и `OrderDetailsValidated` в любом порядке. Чтобы знать, какое из них он получил первым, он меняет состояние заказа. Если первым пришло сообщение `ConsumerValidated`, состояние заказа меняется на **CONSUMER_VALIDATED**, а если `OrderDetailsValidated` — на **ORDER_DETAILS_VALIDATED**. Получив второе сообщение, сервис **Order** присваивает заказу состояние **VALIDATED**.

После проверки заказа сервис **Order** выполняет оставшиеся шаги по его созданию, о которых мы поговорим в следующей главе. Замечательной стороной этого подхода является то, что сервис **Order** сможет создать заказ и ответить клиенту, даже если сервис **Consumer** окажется недоступным. Рано или поздно сервис **Consumer** восстановится и обработает все отложенные сообщения, что позволит завершить проверку заказов.

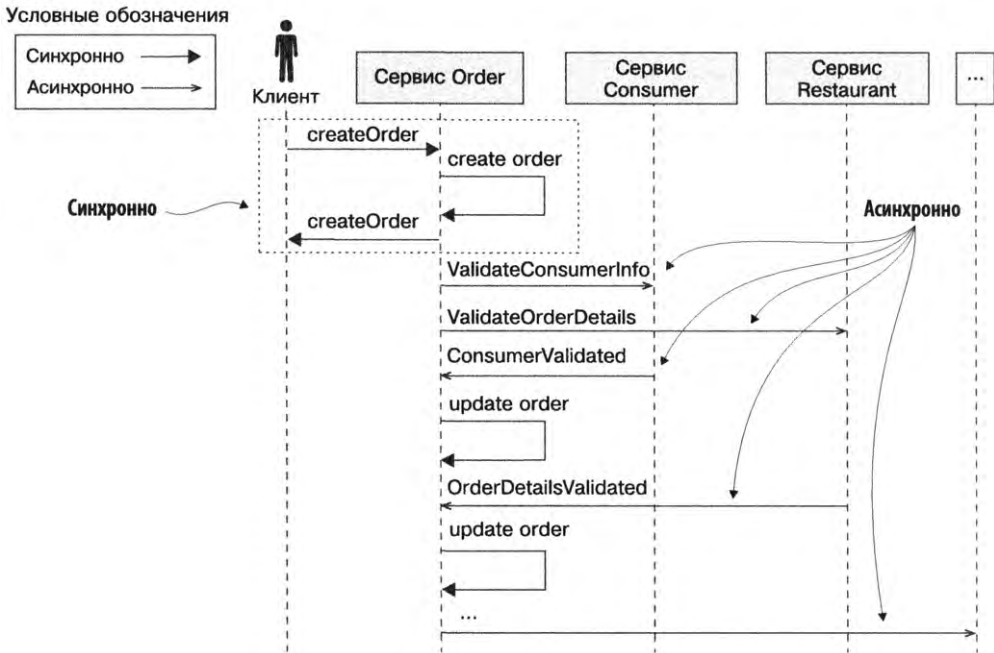


Рис. 3.18. Сервис Order создает заказ, не вызывая ни один из прочих сервисов. Затем он асинхронно проверяет только что созданный заказ, обмениваясь сообщениями с другими сервисами, Consumer и Restaurant

Недостаток возвращения ответа до полной обработки запроса связан с тем, что это делает клиент более сложным. Например, когда сервис Order возвращает ответ, он дает минимальные гарантии по поводу состояния только что созданного заказа. Он отвечает немедленно, еще до проверки заказа и авторизации банковской карты клиента. Таким образом, чтобы узнать о том, успешно ли создан заказ, клиент должен периодически запрашивать информацию или же сервис Order должен послать ему уведомительное сообщение. Несмотря на всю сложность этого подхода, во многих случаях стоит предпочесть его, особенно из-за того, что он учитывает проблемы с управлением распределенными транзакциями, которые мы обсудим в главе 4. В главах 4 и 5 я продемонстрирую эту методику на примере сервиса Order.

Резюме

- ❑ Микросервисная архитектура является распределенной, поэтому межпроцессное взаимодействие играет в ней ключевую роль.
- ❑ К развитию API сервиса необходимо подходить тщательно и осторожно. Легче всего вносить обратно совместимые изменения, поскольку они не влияют на работу клиентов. При внесении ломающих изменений в API сервиса обычно приходится поддерживать как старую, так и новую версию, пока клиенты не обновятся.

- ❑ Существует множество технологий IPC, каждая со своими достоинствами и недостатками. Ключевое решение на стадии проектирования — выбор между синхронным удаленным вызовом процедур и асинхронными сообщениями. Самыми простыми в использовании являются синхронные протоколы вроде REST, основанные на вызове удаленных процедур. Но в идеале, чтобы повысить уровень доступности, сервисы должны взаимодействовать с помощью асинхронного обмена сообщениями.
- ❑ Чтобы предотвратить лавинообразное накопление сбоев в системе, клиент, использующий синхронный протокол, должен быть способен справиться с частичными отказами — тем, что вызываемый сервис либо недоступен, либо проявляет высокую латентность. В частности, при выполнении запросов следует отсчитывать время ожидания, ограничивать количество просроченных запросов и применять шаблон «Предохранитель», чтобы избежать обращений к неисправному сервису.
- ❑ Архитектура, использующая синхронные протоколы, должна содержать механизм обнаружения, чтобы клиенты могли определить сетевое местонахождение экземпляров сервиса. Проще всего остановиться на механизме обнаружения, который предоставляет платформа развертывания: на шаблонах «Обнаружение на стороне сервера» и «Сторонняя регистрация». Альтернативный подход — реализация обнаружения сервисов на уровне приложения: шаблоны «Обнаружение на стороне клиента» и «Саморегистрация». Этот способ требует больших усилий, но подходит для ситуаций, когда сервисы выполняются на нескольких платформах развертывания.
- ❑ Модель сообщений и каналов инкапсулирует детали реализации системы обмена сообщениями и становится хорошим выбором при проектировании архитектуры этого вида. Позже вы сможете привязать свою архитектуру к конкретной инфраструктуре обмена сообщениями, в которой обычно используется брокер.
- ❑ Ключевая трудность при обмене сообщениями связана с их публикацией и обновлением базы данных. Удачным решением является применение шаблона «Публикация событий»: сообщение в самом начале записывается в базу данных в рамках транзакции. Затем отдельный процесс извлекает сообщение из базы данных, используя шаблон «Опрашивающий издатель» или «Отслеживание транзакционного журнала», и передает его брокеру.

Управление транзакциями с помощью повествований

В этой главе

- Почему распределенные транзакции не подходят для современных приложений.
- Использование шаблона «Повествование» для поддержания согласованности данных в микросервисной архитектуре.
- Координация повествований хореографии и оркестрации.
- Контрмеры при нехватке изолированности.

Когда Мэри начала исследовать микросервисную архитектуру, то поняла, что один из аспектов, которые беспокоили ее больше всего, связан с реализацией транзакций, охватывающих несколько сервисов. Транзакции — незаменимый компонент любого промышленного приложения. Без них невозможно поддерживать согласованность данных.

Транзакции типа ACID (Atomicity, Consistency, Isolation, Durability — «атомарность, согласованность, изолированность, долговечность») значительно упрощают жизнь разработчиков, создавая иллюзию того, что каждая из них имеет эксклюзивный доступ к данным. В микросервисной архитектуре ACID-транзакции могут использовать даже запросы, выполняемые в рамках одного сервиса. Однако основная трудность состоит в реализации операций, которые обновляют данные, принадлежащие разным сервисам. Например, как упоминалось в главе 2, операция `createOrder()` охватывает множество сервисов, включая `Order`, `Kitchen` и `Accounting`. Для подобных операций нужен механизм управления транзакциями, который не ограничен отдельным сервисом.

Мэри обнаружила, что традиционный подход к управлению распределенными транзакциями не очень подходит для современных приложений (об этом говорилось

в главе 2). Вместо ACID-транзакций операция, охватывающая несколько сервисов и стремящаяся поддерживать согласованность данных, должна использовать то, что называется *повествованием* (или *сагой*), — последовательность локальных транзакций на основе сообщений. Одна из проблем повествований связана с тем, что по своей природе они являются ACD (Atomicity, Consistency, Durability — «атомарность, согласованность, долговечность»). Им не хватает поддержки изолированности, которая есть в ACID-транзакциях. В итоге приложение должно использовать так называемые *контрмеры* — методики проектирования, которые устраняют или снижают влияние аномалий конкурентности, вызванных нехваткой изолированности.

Скорее всего, самым большим препятствием, с которым столкнутся Мэри и разработчики FTGO при внедрении микросервисов, будет переход от единой базы данных с ACID-транзакциями к архитектуре с множеством баз данных и работе с ACD-повествованиями. Они привыкли к простоте модели ACID-транзакций. Но в реальности даже такие монолитные приложения, как FTGO, не используют классические ACID-транзакции. Во многих проектах установлен пониженный уровень изолированности, чтобы улучшить производительность. Кроме того, множество важных бизнес-процессов, таких как перевод средств между счетами в разных банках, имеют отложенную согласованность. Даже в Starbucks не задействуется двухэтапная фиксация (www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html).

Я начну эту главу с рассмотрения трудностей в работе с транзакциями в микросервисной архитектуре и объясню, почему здесь не годится традиционный подход к распределенным транзакциям. Далее расскажу, как поддерживать согласованность данных с помощью повествований. После этого мы поговорим о двух способах координации повествований: *хореографии*, когда участники обмениваются событиями без централизованной точки управления, и *оркестрации*, когда централизованный контроллер говорит участникам повествования, какие операции нужно выполнить. Я покажу, как использовать контрмеры, чтобы устранить или снизить влияние аномалий конкурентности, вызванных нехваткой изолированности между повествованиями. В конце будет представлен пример реализации повествования.

Для начала посмотрим, какие трудности сопровождают работу с транзакциями в микросервисной архитектуре.

4.1. Управление транзакциями в микросервисной архитектуре

Почти любой запрос, обрабатываемый промышленным приложением, выполняется в рамках транзакции базы данных. Разработчики таких приложений используют фреймворки и библиотеки, которые упрощают работу с транзакциями. Некоторые инструменты предоставляют императивный API для выполняемого вручную начала, фиксации и отката транзакций. А такие фреймворки, как Spring, имеют декларативный механизм. Spring поддерживает аннотацию `@Transactional`, которая автоматически вызывает метод в рамках транзакции. Благодаря этому написание транзакционной бизнес-логики становится довольно простым.

Если быть более точным, управлять транзакциями просто в монолитных приложениях, которые обращаются к единой базе данных. Если же приложение задействует несколько БД и брокеров сообщений, этот процесс затрудняется. Ну а в микросервисной архитектуре транзакции охватывают несколько сервисов, каждый из которых имеет свою БД. В таких условиях приложение должно использовать более продуманный механизм работы с транзакциями. Как вы вскоре увидите, традиционный подход к распределенным транзакциям нежизнеспособен в современных приложениях. Вместо него системы на основе микросервисов должны применять повествования.

Но прежде, чем переходить к повествованиям, посмотрим, почему управление транзакциями создает столько сложностей в микросервисной архитектуре.

4.1.1. Микросервисная архитектура и необходимость в распределенных транзакциях

Представьте, что вы — разработчик в компании FTGO и отвечаете за реализацию системной операции `createOrder()`. Как было написано в главе 2, эта операция должна убедиться в том, что заказчик может размещать заказы, проверить детали заказа, авторизовать банковскую карту заказчика и создать запись `Order` в базе данных. Реализация этих действий была бы относительно простой в монолитном приложении. Все данные, необходимые для проверки заказа, уже готовы и доступны. Кроме того, для обеспечения согласованности данных можно было бы использовать ACID-транзакции. Вы могли бы просто указать аннотацию `@Transactional` для метода сервиса `createOrder()`.

Однако выполнить эту операцию в микросервисной архитектуре гораздо сложнее. Как видно на рис. 4.1, данные, необходимые операции `createOrder()`, разбросаны по нескольким сервисам. `createOrder()` считывает информацию из сервиса `Consumer` и обновляет содержимое сервисов `Order`, `Kitchen` и `Accounting`.

Поскольку у каждого сервиса есть своя БД, вы должны использовать механизм для согласования данных между ними.

4.1.2. Проблемы с распределенными транзакциями

Традиционный подход к обеспечению согласованности данных между несколькими сервисами, БД или брокерами сообщений заключается в применении распределенных транзакций. Стандартом де-факто для управления распределенными транзакциями является X/Open XA (см. <https://ru.wikipedia.org/wiki/XA>). Модель XA использует *двух-этапную фиксацию* (two-phase commit, 2PC), чтобы гарантировать сохранение или откат всех изменений в транзакции. Для этого требуется, чтобы базы данных, брокеры сообщений, драйверы БД и API обмена сообщениями соответствовали стандарту XA, необходим также механизм межпроцессного взаимодействия, который распространяет глобальные идентификаторы XA-транзакций. Большинство реляционных БД совместимы с XA, равно как и некоторые брокеры сообщений. Например, приложение на основе Java EE может выполнять распределенные транзакции с помощью JTA.

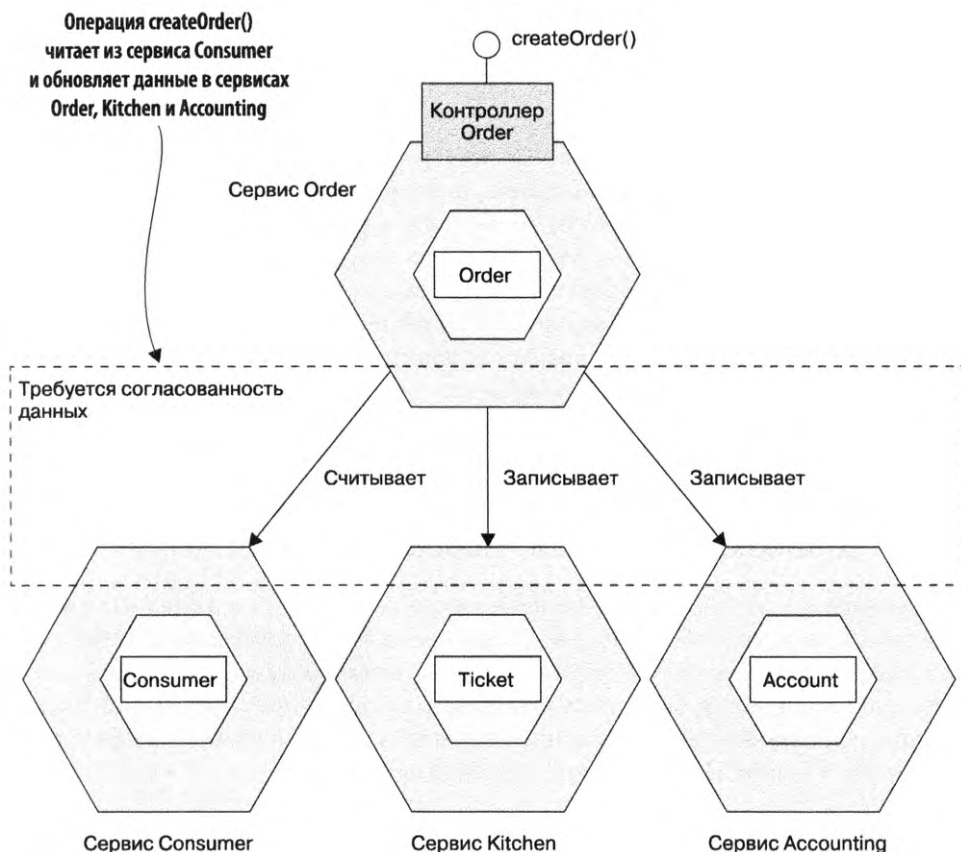


Рис. 4.1. Операция createOrder() обновляет данные в нескольких сервисах. Чтобы обеспечить согласованность между ними, она должна задействовать специальный механизм

Несмотря на внешнюю простоту, распределенные транзакции имеют ряд проблем. Многие современные технологии, включая такие базы данных NoSQL, как MongoDB и Cassandra, их не поддерживают. Распределенные транзакции не поддерживаются и некоторыми современными брокерами сообщений вроде RabbitMQ и Apache Kafka. Так что, если вы решите использовать распределенные транзакции, многие современные инструменты будут вам недоступны.

Еще одна проблема распределенных транзакций связана с тем, что они представляют собой разновидность синхронного IPC, а это ухудшает доступность. Чтобы распределенную транзакцию можно было зафиксировать, доступными должны быть все вовлеченные в нее сервисы. Как описывалось в главе 3, доступность системы — это произведение доступности всех участников транзакции. Если в распределенной транзакции участвуют два сервиса с доступностью 99,5 %, общая доступность будет 99 %, что намного меньше. Каждый дополнительный сервис понижает степень доступности. Эрик Брюер (Eric Brewer) сформулировал CAP-теорему, которая гласит,

что система может обладать лишь двумя из следующих трех свойств: согласованность, доступность и устойчивость к разделению (ru.wikipedia.org/wiki/Теорема_CAP). В наши дни архитекторы отдают предпочтение доступным системам, жертвуя согласованностью.

На первый взгляд распределенные транзакции могут показаться привлекательными. С точки зрения разработчика, они имеют ту же программную модель, что и локальные транзакции. Но из-за проблем, описанных ранее, эта технология оказывается нежизнеспособной в современных приложениях. В главе 3 было показано, как отправлять сообщения в рамках транзакции базы данных, не используя при этом распределенные транзакции. Для решения более сложной проблемы, связанной с обеспечением согласованности данных в микросервисной архитектуре, приложение должно применять другой механизм, основанный на концепции слабо связанных асинхронных сервисов. И здесь пригодятся повествования.

4.1.3. Использование шаблона «Повествование» для сохранения согласованности данных

Повествования — это механизм, обеспечивающий согласованность данных в микросервисной архитектуре без применения распределенных транзакций. Повествование создается для каждой системной команды, которой нужно обновлять данные в нескольких сервисах. Это последовательность локальных транзакций, каждая из которых обновляет данные в одном сервисе, задействуя знакомые фреймворки и библиотеки для ACID-транзакций, упомянутые ранее.

Шаблон «Повествование»

Обеспечивает согласованность данных между сервисами, используя последовательность локальных транзакций, которые координируются с помощью асинхронных сообщений. См. <http://microservices.io/patterns/data/saga.html>.

Системная операция инициирует первый этап повествования. Завершение одной локальной транзакции приводит к выполнению следующей. В разделе 4.2 вы увидите, как координация этих этапов реализуется с помощью асинхронных сообщений. Важным преимуществом асинхронного обмена сообщениями является то, что он гарантирует выполнение всех этапов повествования, даже если один или несколько участников оказываются недоступными.

Повествования имеют несколько важных отличий от ACID-транзакций. Прежде всего, им не хватает изолированности (подробно об этом — в разделе 4.3). К тому же, поскольку каждая локальная транзакция фиксирует свои изменения, для отката повествования необходимо использовать компенсирующие транзакции, о которых мы поговорим позже в этом разделе. Рассмотрим пример повествования.

Пример повествования: создание заказа

В этой главе в качестве примера используем повествование `Create Order` (рис. 4.2). Оно реализует операцию `createOrder()`. Первая локальная транзакция инициируется внешним запросом создания заказа. Остальные пять транзакций срабатывают одна за другой.

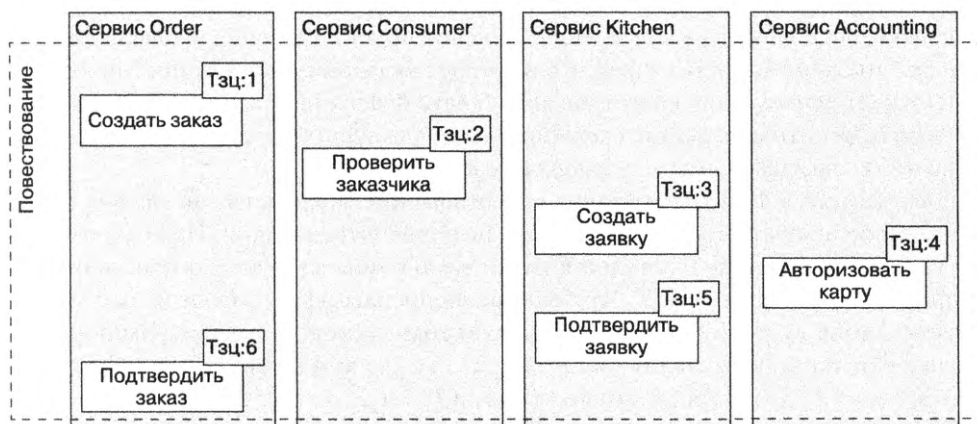


Рис. 4.2. Создание заказа с помощью повествования. Операция `createOrder()` реализуется повествованием, которое состоит из локальных транзакций в нескольких сервисах

Это повествование состоит из следующих локальных транзакций.

1. Сервис **Order**. Создает заказ с состоянием **APPROVAL_PENDING**.
2. Сервис **Consumer**. Проверяет, может ли заказчик размещать заказы.
3. Сервис **Kitchen**. Проверяет детали заказа и создает заявку с состоянием **CREATE_PENDING**.
4. Сервис **Accounting**. Авторизует банковскую карту заказчика.
5. Сервис **Kitchen**. Меняет состояние заявки на **AWAITING_ACCEPTANCE**.
6. Сервис **Order**. Меняет состояние заказа на **APPROVED**.

В разделе 4.2 я покажу, как сервисы, участвующие в повествовании, взаимодействуют между собой с помощью асинхронных сообщений. Сервис публикует сообщение по завершении локальной транзакции. Это инициирует следующий этап повествования и позволяет не только добиться слабой связанности участников, но и гарантировать полное выполнение повествования. Даже если получатель временно недоступен, брокер буферизирует сообщение до того момента, когда его можно будет доставить.

Повествования выглядят простыми, но их использование связано с некоторыми трудностями, прежде всего с нехваткой изолированности между ними. Решение проблемы описано в разделе 4.3. Еще один нетривиальный аспект связан с откатом изменений при возникновении ошибки. Посмотрим, как это делается.

Повествования применяют компенсирующие транзакции для отката изменений

У традиционных ACID-транзакций есть одно прекрасное свойство: бизнес-логика может легко откатить транзакцию, если обнаружится нарушение бизнес-правила. Она просто выполняет команду ROLLBACK, а база данных отменяет все изменения, внесенные до этого момента. К сожалению, повествование нельзя откатить автоматически, поскольку на каждом этапе оно фиксирует изменения в локальной базе данных. Это, к примеру, означает, что в случае неудачной авторизации банковской карты на четвертом этапе повествования Create Order приложение FTGO должно вручную отменить изменения, сделанные на предыдущих трех этапах. Вы должны написать так называемые *компенсирующие транзакции*.

Допустим, $(n + 1)$ -я транзакция в повествовании завершилась неудачно. Необходимо нивелировать последствия от предыдущих n транзакций. На концептуальном уровне каждый из этих этапов T_i имеет свою компенсирующую транзакцию C_i , которая отменяет эффект от T_i . Чтобы компенсировать эффект от первых n этапов, повествование должно выполнить каждую транзакцию C_i в обратном порядке. Последовательность выглядит так: $T_1 \dots T_n, C_n \dots C_1$ (рис. 4.3). В данном примере отказывает этап T_{n+1} , что требует отмены шагов $T_1 \dots T_n$.

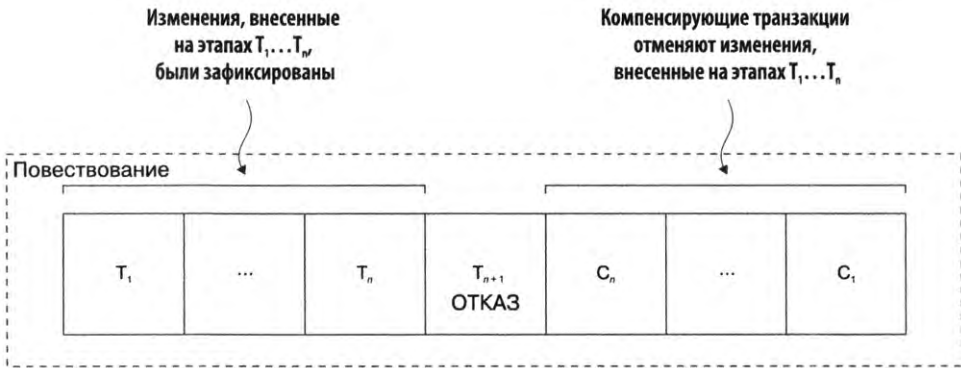


Рис. 4.3. Когда этап повествования завершается неудачей в результате нарушения бизнес-правила, повествование должно вручную отменить все обновления, сделанные на предыдущих этапах, выполнив компенсирующие транзакции

Повествование выполняет компенсирующие транзакции в обратном порядке по отношению к исходным: $C_n \dots C_1$. Здесь действует тот же механизм последовательного выполнения, что и в случае с T_i . Завершение C_i должно инициировать C_{i-1} .

Возьмем, к примеру, повествование Create Order. Оно может отказать по целому ряду причин.

- 1. Некорректная информация о заказчике, или заказчику не позволено создавать заказы.

2. Некорректная информация о ресторане, или ресторан не в состоянии принять заказ.
3. Невозможность авторизовать банковскую карту заказчика.

В случае сбоя в локальной транзакции механизм координации повествования должен выполнить компенсирующие шаги, которые отклоняют заказ и, возможно, заявку. В табл. 4.1 собраны компенсирующие транзакции для каждого этапа повествования `Create Order`. Следует отметить, что не всякий этап требует компенсирующей транзакции. Это относится, например, к операциям чтения, таким как `verifyConsumerDetails()`, или к операции `authorizeCreditCard()`, все шаги после которой всегда завершаются успешно.

Таблица 4.1. Компенсирующие транзакции для повествования `Create Order`

Этап	Сервис	Транзакция	Компенсирующая транзакция
1	Order	<code>createOrder()</code>	<code>rejectOrder()</code>
2	Consumer	<code>verifyConsumerDetails()</code>	—
3	Kitchen	<code>createTicket()</code>	<code>rejectTicket()</code>
4	Accounting	<code>authorizeCreditCard()</code>	—
5	Kitchen	<code>approveTicket()</code>	—
6	Order	<code>approveOrder()</code>	—

В разделе 4.3 вы узнаете, что первые три этапа повествования `Create Order` называются *доступными для компенсации транзакциями*, потому что шаги, следующие за ними, могут отказать. Четвертый этап называется *поворотной транзакцией*, потому что дальнейшие шаги никогда не отказывают. Последние два этапа называются *доступными для повторения транзакциями*, потому что они всегда заканчиваются успешно.

Чтобы понять, как используются компенсирующие транзакции, представьте ситуацию, в которой авторизация банковской карты заказчика проходит неудачно. В этом случае повествование выполняет следующие локальные транзакции.

1. Сервис `Order`. Создает заказ с состоянием `APPROVAL_PENDING`.
2. Сервис `Consumer`. Проверяет, может ли заказчик размещать заказы.
3. Сервис `Kitchen`. Проверяет детали заказа и создает заявку с состоянием `CREATE_PENDING`.
4. Сервис `Accounting`. Делает неудачную попытку авторизовать банковскую карту заказчика.
5. Сервис `Kitchen`. Меняет состояние заявки на `CREATE_REJECTED`.
6. Сервис `Order`. Меняет состояние заказа на `REJECTED`.

Пятый и шестой этапы — это компенсирующие транзакции, которые отменяют обновления, внесенные сервисами `Kitchen` и соответственно `Order`. Координирующая логика повествования отвечает за последовательность выполнения прямых и компенсирующих транзакций. Посмотрим, как это работает.

4.2. Координация повествований

Реализация повествования состоит из логики, которая координирует его этапы. Когда повествование инициируется системной командой, координирующая логика должна выбрать первого участника и сделать так, чтобы тот выполнил локальную транзакцию. Когда транзакция завершится, механизм координации выберет и вызовет следующего участника. Этот процесс продолжается до тех пор, пока повествование не выполнит все свои этапы. Если какая-либо локальная транзакция завершится неудачно, повествование должно выполнить компенсирующие транзакции в обратном порядке. Координирующую логику можно структурировать следующими способами.

- ❑ *Хореография* — распределение принятия решений и упорядочения действий между участниками повествования, которые в основном общаются, обмениваясь событиями.
- ❑ *Оркестрация* — централизация координирующей логики повествования в виде класса-оркестратора. *Оркестратор* отправляет участникам повествования командные сообщения с инструкциями, какие операции нужно выполнить.

Обсудим оба варианта. Начнем с хореографии.

4.2.1. Повествования, основанные на хореографии

Хореография — это один из способов реализации повествований. Она не предусматривает центрального координатора, который выдает участникам команды. Вместо этого участники подписываются на события друг друга и реагируют соответствующим образом. Чтобы показать, как работают повествования на основе хореографии, я сначала опишу пример. После этого мы обсудим несколько архитектурных проблем, которые вы должны учитывать. Затем будут представлены плюсы и минусы использования хореографии.

Реализация повествования `Create Order` с помощью хореографии

На рис. 4.4 представлена архитектура повествования `Create Order`, основанного на хореографии. Участники взаимодействуют, обмениваясь сообщениями. Каждый участник, начиная с сервиса `Order`, обновляет свою базу данных и публикует событие, благодаря которому срабатывает следующий участник.

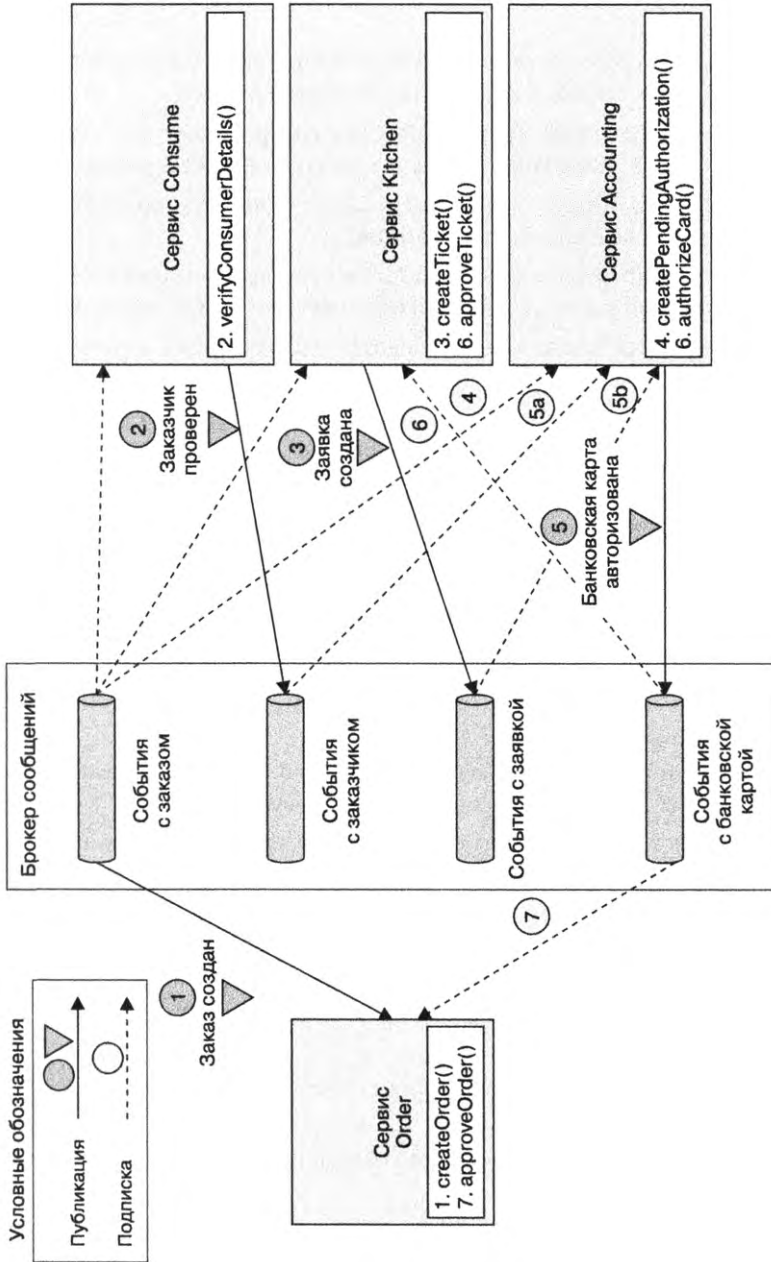


Рис. 4.4. Реализация повествования Create Order с помощью хореографии. Участники повествования общаются, обмениваясь событиями

Оптимистичный путь через это повествование выглядит так.

1. Сервис **Order** создает заказ с состоянием **APPROVAL_PENDING** и публикует событие **OrderCreated**.
2. Сервис **Consumer** потребляет событие **OrderCreated**, проверяет, может ли заказчик размещать заказы, и публикует событие **ConsumerVerified**.
3. Сервис **Kitchen** потребляет событие **OrderCreated**, проверяет заказ, создает заявку с состоянием **CREATE_PENDING** и публикует событие **TicketCreated**.
4. Сервис **Accounting** потребляет событие **OrderCreated** и создает объект **CreditCardAuthorization** с состоянием **PENDING**.
5. Сервис **Accounting** потребляет события **TicketCreated** и **ConsumerVerified**, выставляет счет банковской карте заказчика и публикует событие **CreditCardAuthorized**.
6. Сервис **Kitchen** потребляет событие **CreditCardAuthorized** и меняет состояние заявки на **AWAITING_ACCEPTANCE**.
7. Сервис **Order** принимает события **CreditCardAuthorized**, меняет состояние заказа на **APPROVED** и публикует событие **OrderApproved**.

Повествование **Create Order** должно также предусматривать сценарий, в котором участник отклоняет заказ и публикует некое неудачное событие. Например, неудачей может завершиться авторизация банковской карты заказчика. Повествование должно выполнить компенсирующие транзакции, чтобы отменить то, что уже было сделано. На рис. 4.5 показана последовательность событий, когда сервису **Accounting** не удастся авторизовать банковскую карту заказчика. Это выглядит так.

1. Сервис **Order** создает заказ с состоянием **APPROVAL_PENDING** и публикует событие **OrderCreated**.
2. Сервис **Consumer** потребляет событие **OrderCreated**, проверяет, может ли заказчик размещать заказы, и публикует событие **ConsumerVerified**.
3. Сервис **Kitchen** потребляет событие **OrderCreated**, проверяет заказ, создает заявку с состоянием **CREATE_PENDING** и публикует событие **TicketCreated**.
4. Сервис **Accounting** потребляет событие **OrderCreated** и создает объект **CreditCardAuthorization** с состоянием **PENDING**.
5. Сервис **Accounting** потребляет события **TicketCreated** и **ConsumerVerified**, выставляет счет банковской карте заказчика и публикует событие **CreditCardAuthorizationFailed**.
6. Сервис **Kitchen** потребляет событие **CreditCardAuthorizationFailed** и меняет состояние заявки на **REJECTED**.
7. Сервис **Order** потребляет событие **CreditCardAuthorizationFailed** и меняет состояние заказа на **REJECTED**.

Как видите, участники повествования, основанного на хореографии, взаимодействуют в стиле «издатель/подписчик». Давайте подробнее поговорим о некоторых проблемах, которые следует учитывать при реализации взаимодействия вида «издатель/подписчик» в ваших повествованиях.

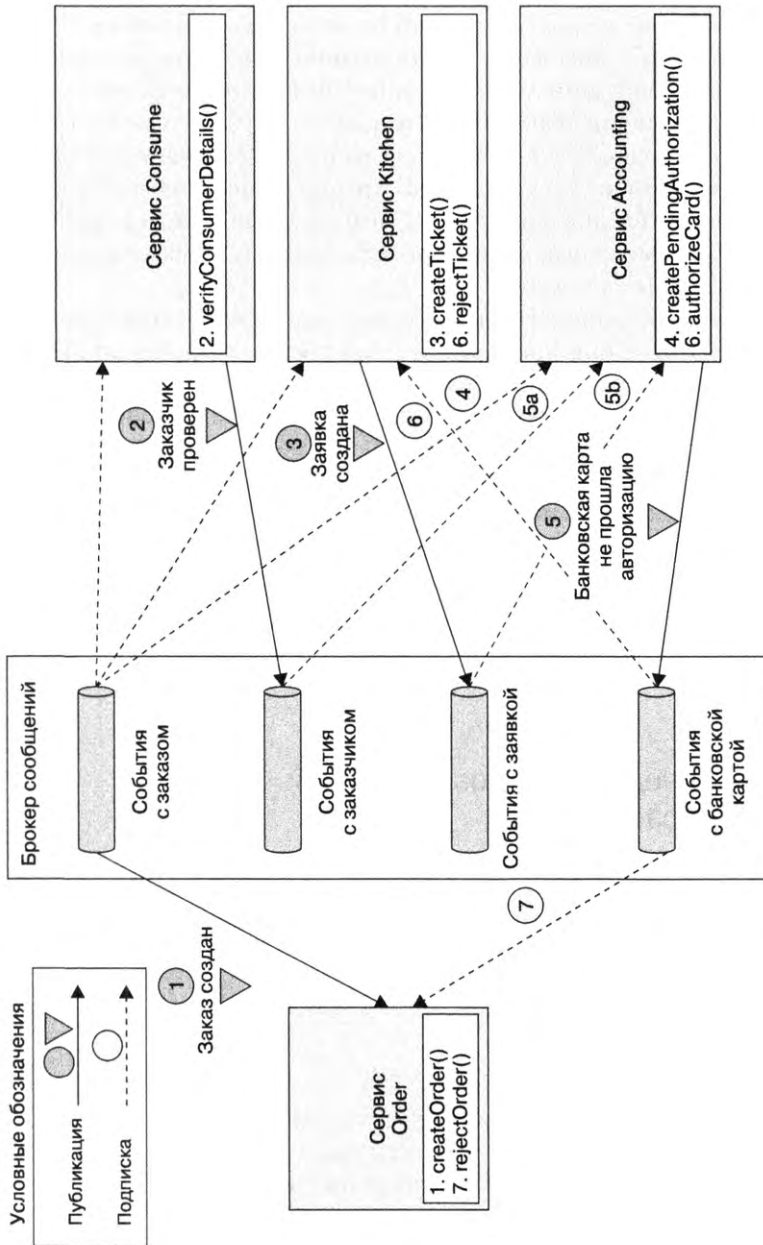


Рис. 4.5. Последовательность событий в повествовании Create Order, когда авторизация банковской карты заказчика проходит неудачно. Сервис Accounting публикует событие CreditCardAuthorizationFailed, которое заставляет сервис Kitchen отклонить заявку, а сервис Order — отклонить заказ

Надежное взаимодействие на основе событий

У межсервисного взаимодействия есть несколько проблем, которые необходимо учитывать при реализации повествований на основе хореографии. Во-первых, следует сделать так, чтобы участники повествования обновляли свои базы данных и публиковали события в рамках транзакций БД. Каждый этап повествования, основанного на хореографии, обновляет базу данных и публикует событие. Например, в `Create Order` сервис `Kitchen` получает событие `ConsumerVerified`, создаст заявку и публикует событие `TicketCreated`. Крайне важно, чтобы обновление БД и публикация события были атомарными. Следовательно, для надежного взаимодействия участники повествования должны использовать транзакционный обмен сообщениями, описанный в главе 3.

Во-вторых, участники повествования должны иметь возможность сопоставить каждое событие, которое они принимают, с собственными данными. Например, когда сервис `Order` получает событие `CreditCardAuthorized`, он должен уметь найти соответствующий заказ. Решением здесь является публикация событий с *идентификаторами соответствия*, благодаря которым другие участники могут выполнить сопоставление.

Например, участники повествования `Create Order` могут использовать параметр `orderId` в качестве ID соответствия, он будет передаваться от одного участника к другому. Сервис `Accounting` публикует событие `CreditCardAuthorized` с `orderId` из события `TicketCreated`. Когда сервис `Order` получает событие `CreditCardAuthorized`, он задействует `orderId` для извлечения соответствующего заказа. Аналогичным образом сервис `Kitchen` задействует `orderId` из того же события, чтобы извлечь подходящую заявку.

Преимущества и недостатки повествований на основе хореографии

У повествований, основанных на хореографии, есть несколько преимуществ.

- ❑ *Простота.* Сервисы публикуют события при создании, обновлении и удалении бизнес-объектов.
- ❑ *Слабая связанность.* Участники подписываются на события, не владея непосредственной информацией друг о друге.

Но существуют и определенные недостатки.

- ❑ *Они сложнее для понимания.* В отличие от оркестрации хореография не описывает повествование на каком-то одном участке кода — его реализация разбросана между сервисами. Из-за этого разработчикам иногда трудно понять, как работает то или иное повествование.
- ❑ *Возникают циклические зависимости между сервисами.* Участники повествования подписываются на события друг друга, что часто создает циклические зависимости. Например, если внимательно присмотреться к рис. 4.4, можно за-

метить такие циклические зависимости, как `Order` → `Accounting` → `Order`. Они не всегда являются проблемой, но, как принято считать, их наличие — признак плохого тона.

- ❑ *Существует риск жесткого связывания.* Каждый участник повествования должен подписаться на все события, которые на него влияют. Например, сервис `Accounting` интересуют все события, приводящие к выставлению счета или возмещению средств на банковской карте. В итоге возникает риск того, что ему придется обновляться синхронно с жизненным циклом заказа, который реализован сервисом `Order`.

Хореография может хорошо работать с простыми повествованиями, но, учитывая ее недостатки, в более сложных случаях лучше использовать оркестрацию. Об этом мы поговорим далее.

4.2.2. Повествования на основе оркестрации

Оркестрация — это еще один способ реализации повествований. Она подразумевает определение класса-оркстратора, единственной задачей которого является рассылка инструкций участникам. Оркстратор взаимодействует с участниками в стиле «команда/асинхронный ответ». Чтобы выполнить этап повествования, он шлет участнику командное сообщение, объясняя, какую операцию тот должен выполнить. После выполнения операции участник возвращает оркстратору сообщение с ответом. Оркстратор обрабатывает это сообщение и решает, какой этап повествования нужно выполнить дальше.

Чтобы показать, как работают повествования на основе оркестрации, я сначала опишу пример. Затем покажу, как моделировать такие повествования в виде конечного автомата. И объясню, как обеспечить надежное взаимодействие между оркстратором и участниками с помощью транзакционного обмена сообщениями. В конце мы обсудим преимущества и недостатки повествований на основе оркестрации.

Реализация повествования `Create Order` с помощью оркестрации

Архитектура повествования `Create Order`, основанного на оркестрации, представлена на рис. 4.6. Повествование управляется с помощью класса `CreateOrderSaga`, который общается с участниками с помощью асинхронных запросов/ответов. Этот класс отслеживает весь процесс и шлет командные сообщения участникам, таким как сервисы `Kitchen` и `Consumer`. Класс `CreateOrderSaga` читает сообщения из канала с ответами и определяет следующий шаг повествования (если таковой имеется).

Вначале сервис `Order` создает заказ и оркстратор повествования `Create Order`. После этого оптимистичный путь выглядит так.

1. Оркстратор повествования отправляет сервису `Consumer` команду `VerifyConsumer`.
2. Сервис `Consumer` возвращает в ответ сообщение `ConsumerVerified`.

3. Оркестратор отправляет сервису Kitchen команду CreateTicket.
4. Сервис Kitchen возвращает в ответ сообщение TicketCreated.
5. Оркестратор отправляет сервису Accounting сообщение AuthorizeCard.
6. Сервис Accounting возвращает в ответ сообщение CardAuthorized.
7. Оркестратор отправляет сервису Kitchen команду ApproveTicket.
8. Оркестратор отправляет сервису Order команду ApproveOrder.

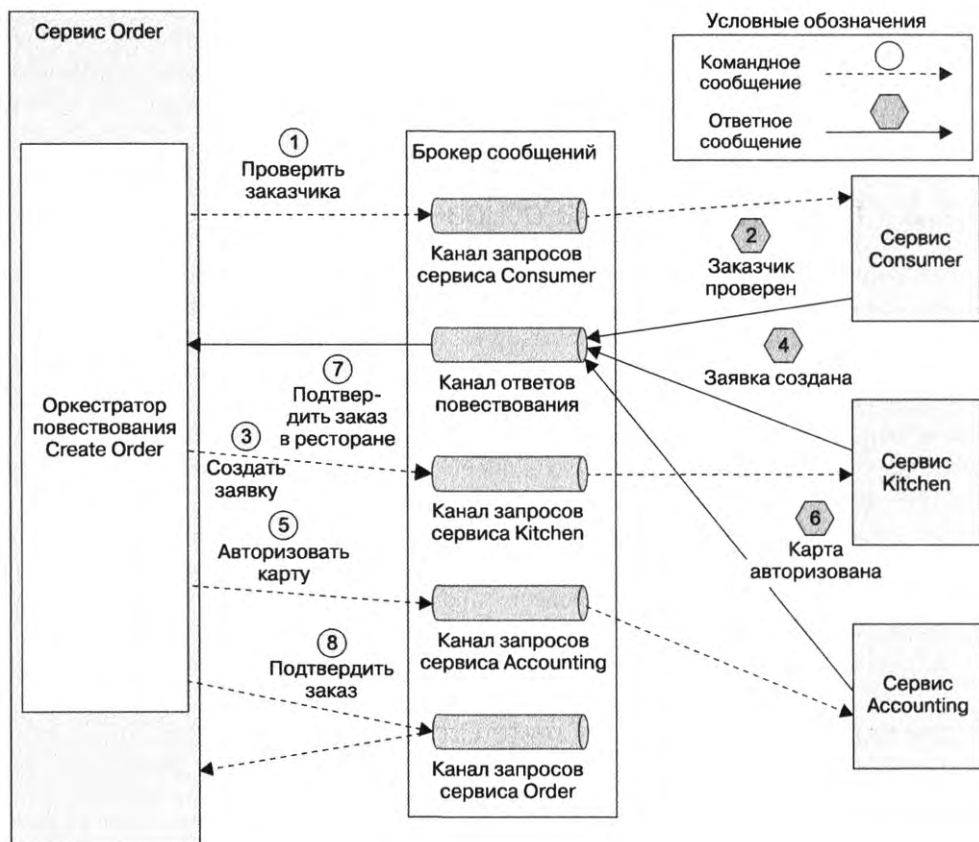


Рис. 4.6. Реализация повествования Create Order с помощью оркестрации. Сервис Order реализует оркестратор, который вызывает участников повествования с помощью асинхронных запросов/ответов

Обратите внимание на то, что на последнем этапе оркестратор шлет командное сообщение сервису Order, компонентом которого он сам является. В принципе, повествование Create Order могло бы подтвердить заказ, обновив его напрямую. Но, чтобы оставаться последовательным, оно обращается с сервисом Order просто как с еще одним участником.

Такие схемы, как на рис. 4.6, описывают лишь один из множества потенциальных сценариев повествования. Например, у повествования `Create Order` есть четыре сценария. Помимо оптимистичного пути, процесс может завершиться неудачно из-за сбоя в сервисах `Consumer`, `Kitchen` или `Accounting`. В связи с этим имеет смысл смоделировать повествование в виде конечного автомата, который описывает все возможные сценарии.

Моделирование оркестраторов повествований в виде конечных автоматов

Конечный автомат — это хорошая модель для оркестратора повествования. Он состоит из набора состояний и переходов между ними, которые инициируются с помощью событий. У каждого перехода может быть какое-то действие, которое в контексте повествования означает вызов участника. Переходы между состояниями инициируются завершением локального перехода, выполненного участником повествования. Текущее состояние и конкретный результат локального перехода определяют последующий переход и действие, которое нужно выполнить (если таковое имеется). Конечный автомат имеет эффективные стратегии тестирования. Благодаря этому использование данной модели упрощает проектирование, реализацию и тестирование повествований.

На рис. 4.7 показана модель конечного автомата для повествования `Create Order`. Она включает в себя следующие состояния.

- ❑ *Проверка заказчика* — начальное состояние. Повествование ждет, когда сервис `Consumer` подтвердит, что заказчик может размещать заказы.
- ❑ *Создание заявки* — повествование ждет ответа на команду `CreateTicket`.
- ❑ *Авторизация карты* — ожидание авторизации банковской карты заказчика сервисом `Accounting`.
- ❑ *Заказ подтвержден* — финальный этап, свидетельствующий об успешном завершении повествования.
- ❑ *Заказ отклонен* — финальный этап, свидетельствующий об отклонении заказа одним из участников.

Конечный автомат также определяет множество переходов состояний. Например, состояние *создание заявки* может перейти в одно из двух состояний: *авторизация карты* или *заказ отклонен*. В первом случае нужно получить успешный ответ на команду `CreateTicket`, а во втором сервису `Kitchen` не удастся создать заявку.

В самом начале конечный автомат отправляет команду `VerifyConsumer` сервису `Consumer`. Ответ от этого сервиса инициирует переход к следующему состоянию. Если заказчик успешно прошел проверку, повествование создает заявку и переходит к состоянию «создание заявки». Но если проверка завершается неудачно, повествование отклоняет заказ и переходит к состоянию «отклонение заказа». Конечный автомат выполняет множество других переходов, которые инициируются ответами участников повествования, пока не дойдет до финального состояния: «заказ подтвержден» или «заказ отклонен».

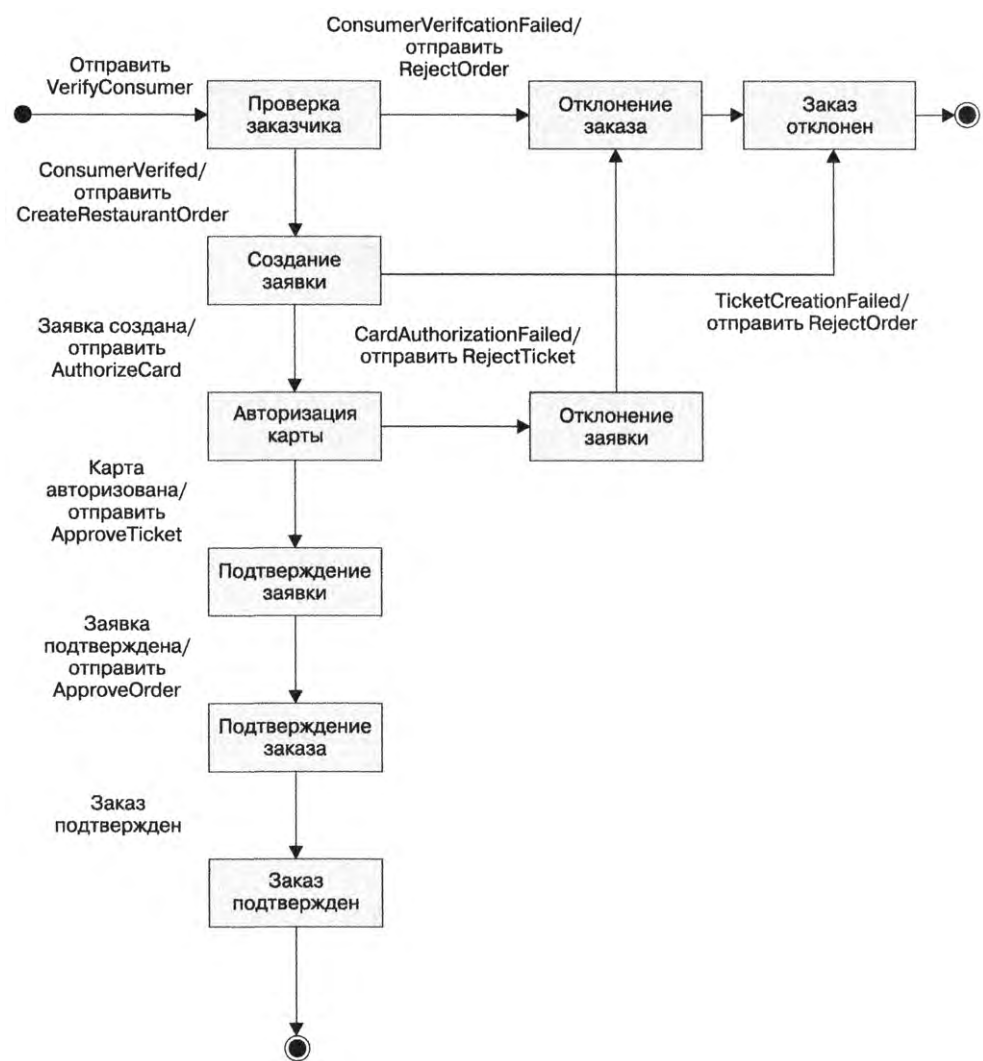


Рис. 4.7. Модель конечного автомата для повествования Create Order

Оркестрация повествований и транзакционный обмен сообщениями

На каждом этапе повествования, основанного на оркестрации, какой-то сервис обновляет базу данных и публикует сообщение. Например, сервис `Order` сохраняет заказ, а оркестратор шлет сообщение первому участнику повествования. Участник (например, сервис `Kitchen`) обрабатывает команду, обновляя свою базу данных и отправляя ответное сообщение. Сервис `Order` обрабатывает ответ участника, обновляя

состояние оркестратора и отправляя командное сообщение следующему участнику. Как описано в главе 3, для атомарного обновления БД и публикации сообщений сервис должен использовать транзакции. В разделе 4.4 вы подробнее познакомитесь с реализацией оркестратора повествования `Create Order`, в том числе с тем, как в нем применяется транзакционный обмен сообщениями.

Рассмотрим преимущества и недостатки использования оркестрации в повествованиях.

Преимущества и недостатки повествований, основанных на оркестрации

Повествования, основанные на оркестрации, имеют несколько преимуществ.

- ❑ *Упрощенные зависимости.* Одной из положительных сторон оркестрации является то, что она не создает циклических зависимостей. Оркестратор вызывает участников повествования, но участники не вызывают оркестратора. В результате оркестратор зависит от участников, но не наоборот, поэтому циклических зависимостей нет.
- ❑ *Меньше связывания.* Каждый сервис реализует API, который вызывается оркестратором, поэтому ему не нужно знать о событиях, публикуемых другими участниками повествования.
- ❑ *Улучшенное разделение ответственности и упрощенная бизнес-логика.* Вся координирующая логика повествования находится в оркестраторе. Благодаря этому доменные объекты становятся проще и им не нужно знать о повествованиях, в которых они участвуют. Например, при использовании оркестрации класс `Order` ничего не знает о повествованиях, поэтому имеет более простую модель конечного автомата. Во время выполнения повествования `Create Order` он переходит напрямую из состояния `APPROVAL_PENDING` в состояние `APPROVED`. Класс `Order` не обладает никакими промежуточными состояниями, которые соответствуют этапам повествования. Это значительно упрощает бизнес-логику.

При этом оркестрация имеет один недостаток — риск избыточной централизации бизнес-логики в оркестраторе. В результате получается архитектура, в которой умный оркестратор командует глупыми сервисами. К счастью, этой проблемы можно избежать, если проектировать оркестраторы так, чтобы они отвечали лишь за последовательное выполнение действий и не содержали никакой дополнительной бизнес-логики.

Я рекомендую использовать оркестрацию для любых повествований, за исключением самых простых. Реализация координирующей логики для повествований — это лишь одна из задач проектирования, которые вы должны решить. Еще одной проблемой, которая наверняка создаст вам больше всего трудностей в ходе работы с повествованиями, является нехватка изолированности. Давайте рассмотрим ее и подумаем, как ее решить.

4.3. Что делать с недостаточной изолированностью

Буква *I* в аббревиатуре ACID означает *isolation* (изолированность). Это свойство гарантирует, что результат параллельного выполнения нескольких ACID-транзакций будет таким же, как при некоем последовательном выполнении. База данных дает иллюзию того, что каждая ACID-транзакция имеет эксклюзивный доступ к информации. Изолированность намного упрощает написание бизнес-логики, которая выполняется конкурентно.

Трудность в ходе работы с повествованиями состоит в том, что им не хватает изолированности ACID-транзакций. Дело тут в следующем: обновления, которые выполняет каждая локальная транзакция, сразу же фиксируются и становятся доступными любым другим повествованиям. Такое поведение может создать две проблемы. Во-первых, другие повествования могут изменить данные, к которым обращается используемое в данный момент повествование. Во-вторых, другие повествования могут читать данные в процессе их обновления, что чревато несогласованностью. На самом деле можно считать, что повествования соответствуют принципу ACD.

- ❑ *Atomicity (атомарность)* — реализация повествования гарантирует выполнение или отмену всех транзакций.
- ❑ *Consistency (согласованность)* — за ссылочную целостность внутри сервиса отвечает локальная база данных, за ссылочную целостность между сервисами — сами сервисы.
- ❑ *Durability (устойчивость)* — обеспечивается локальной базой данных.

Нехватка изолированности может вызвать *аномалии* (этот термин встречается в литературе по базам данных). Так называется ситуация, когда параллельное выполнение транзакций дает иные результаты, чем последовательное.

На первый взгляд отсутствие изолированности кажется неприемлемым. Но на практике разработчики часто уменьшают изолированность для получения более высокой производительности. СУРБД позволяют выбрать уровень изолированности для каждой транзакции (dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html). Полная изолированность, известная также как сериализуемые транзакции, обычно не используется по умолчанию. В реальном мире транзакции часто отклоняются от книжных определений ACID.

В следующем разделе мы обсудим ряд стратегий проектирования повествований, которые помогают справиться с нехваткой изолированности. Их также называют *контрмерами*. Некоторые из них реализуют изолированность на уровне приложения. Другие снижают бизнес-риски, связанные с ее отсутствием. Используя контрмеры, вы сможете написать бизнес-логику на основе повествований, которая будет работать корректно.

Я начну с описания аномалий, вызванных нехваткой изолированности. После этого мы поговорим о контрмерах, которые либо устраняют эти аномалии, либо снижают их бизнес-риски.

4.3.1. Обзор аномалий

Нехватка изолированности может привести к трем аномалиям.

- ❑ *Потеря обновлений* — одно повествование перезаписывает изменения, внесенные другим, не читая их при этом.
- ❑ *«Грязное» чтение* — транзакция или повествование читают незавершенные обновления другого повествования.
- ❑ *Нечеткое/неповторяемое чтение* — два разных этапа повествования читают одни и те же данные, но получают разные результаты, потому что другое повествование внесло изменения.

Вы можете столкнуться со всеми тремя аномалиями, но первые две самые распространенные и вызывающие больше проблем. Рассмотрим их подробнее. Начнем с потерянных обновлений.

Потерянные обновления

Эта аномалия возникает, когда повествование перезаписывает обновление, сделанное другим повествованием. Например, представьте себе такой сценарий.

1. Первый этап повествования **Create Order** создает заказ.
2. Пока это повествование выполняется, повествование **Cancel Order** отменяет заказ.
3. На завершающем этапе повествование **Create Order** подтверждает заказ.

В этой ситуации **Create Order** игнорирует и перезаписывает обновление, выполненное повествованием **Cancel Order**. В итоге приложение FTGO отправит уже отмененный заказ. Позже в этом разделе я покажу, как предотвратить потерю обновлений.

«Грязное» чтение

«Грязным» называют чтение, которое происходит в процессе обновления данных другим повествованием. Представьте, к примеру, разновидность приложения FTGO с поддержкой кредитного лимита. В этом случае повествование, отменяющее заказ, состоит из следующих транзакций:

- ❑ сервис **Consumer** — увеличивает доступный кредит;
- ❑ сервис **Order** — меняет состояние заказа на **CANCELLED**;
- ❑ сервис **Delivery** — отменяет доставку.

Теперь представьте, что во время выполнения повествований **Cancel Order** и **Create Order** первое откатывается, так как отменять доставку уже поздно. Есть вероятность того, что у нас получится следующая последовательность транзакций, которые вызывают сервис **Consumer**:

- ❑ **Cancel Order** — увеличивает доступный кредит;
- ❑ **Create Order** — уменьшает доступный кредит;

- ❑ **Cancel Order** — компенсирующая транзакция, которая уменьшает доступный кредит.

В этом сценарии повествование **Create Order** выполняет «грязное» чтение доступного кредита, что позволяет клиенту разместить заказ, превышающий его кредитный лимит. Вполне вероятно, что такой риск для компании неприемлем.

Посмотрим, как не дать этой и другим аномалиям повлиять на приложение.

4.3.2. Контрмеры на случай нехватки изолированности

Повествования имеют транзакционную модель ACD, нехватка изолированности в которой может привести к аномалиям, которые выводят приложение из строя. Разработчик обязан писать свои повествования так, чтобы избежать этих аномалий или минимизировать их влияние на бизнес. Может показаться, что эта задача не из легких, но вы уже видели пример стратегии, которая предотвращает аномалии. Один из возможных подходов — использование в заказе состояний вида ***_PENDING**, таких как **APPROVAL_PENDING**. Повествование, обновляющее заказы, например **Create Order**, вначале устанавливает состояние в ***_PENDING**. Благодаря этому другие транзакции будут знать, что заказ обновляется повествованием, и смогут среагировать соответствующим образом.

Применение в заказах состояний вида ***_PENDING** — это пример того, что Ларс Фрэнк (Lars Frank) и Торбен Захл (Torben U. Zahle) называют в своей статье «Семантические ACID-свойства в среде с несколькими БД и использованием удаленного вызова процедур и распространения обновлений» (<https://dl.acm.org/citation.cfm?id=284472.284478>) *контрмерой семантической блокировки*. Эта статья объясняет, как справляться с недостаточной изоляцией в архитектуре с несколькими базами данных, которая не использует распределенных транзакций. Многие приемы, которые в ней описаны, могут пригодиться при проектировании повествований. Для борьбы с аномалиями, вызванными нехваткой изолированности, в ней предложен ряд контрмер, которые либо предотвращают одну или несколько аномалий, либо минимизируют их последствия для бизнеса. Решения, рассмотренные в этой статье, перечислены далее.

- ❑ *Семантическая блокировка* — блокировка на уровне приложения.
- ❑ *Коммутативные обновления* — проектирование операций обновления таким образом, чтобы их можно было выполнить в любом порядке.
- ❑ *Пессимистическое представление* — перестановка этапов повествования для минимизации бизнес-рисков.
- ❑ *Повторное чтение значения* — предотвращение «грязного» чтения путем повторного считывания данных. Это позволяет убедиться в их неизменности перед тем, как их перезаписывать.
- ❑ *Файл версий* — ведение записей об обновлениях, чтобы их можно было менять местами.
- ❑ *По значению* — использование бизнес-рисков каждого запроса для динамического выбора механизма конкурентности.

Все эти контрмеры будут рассмотрены позже в этом разделе, но сначала я хочу познакомить вас с технологией описания структуры повествований, которая пригодится при обсуждении контрмер.

Структура повествования

В статье о контрмерах, которая упоминается в предыдущем разделе, предложена полезная модель для структурирования повествований (рис. 4.8). В ней повествование состоит из трех типов транзакций.

- ❑ *Транзакции, доступные для компенсации*, — транзакции, которые потенциально можно откатить с помощью компенсирующих транзакций.
- ❑ *Поворотная транзакция* — решающий момент в повествовании. Если поворотная транзакция фиксируется, повествование отработает до конца. Поворотная транзакция может оказаться недоступной ни для компенсации, ни для повторения. Это также может быть последняя компенсируемая или первая повторяемая транзакция.
- ❑ *Транзакции, доступные для повторения*, — транзакции, идущие за поворотной. Всегда завершаются успешно.

Компенсируемые транзакции:
должны поддерживать откат

Этап	Сервис	Транзакция	Компенсирующая транзакция
1	Order	createOrder()	rejectOrder()
2	Consumer	verifyConsumerDetails()	—
3	Kitchen	createTicket()	rejectTicket()
4	Accounting	authorizeCreditCard()	—
5	Restaurant Order	approveRestaurantOrder()	—
6	Order	approveOrder()	—

Поворотная транзакция: решающая
транзакция повествования.
Если пройдет успешно, повествование
отрабатывает до конца

Повторяемые транзакции:
завершение гарантировано

Рис. 4.8. Повествование состоит из транзакций трех типов: компенсируемых (их можно откатить при наличии компенсирующих транзакций), поворотной (решающий момент повествования) и повторяемых (их не нужно откатывать, и они всегда завершаются)

В рамках саги *Create Order* компенсируемыми транзакциями являются этапы `createOrder()`, `verifyConsumerDetails()` и `createTicket()`. У операций `createOrder()` и `createTicket()` есть компенсирующие транзакции, которые отменяют их обновления. Операция `verifyConsumerDetails()` выполняет лишь чтение, поэтому ее не нужно компенсировать. `authorizeCreditCard()` — это поворотная транзакция

в данном повествовании. Если банковскую карту заказчика удастся авторизовать, завершение повествования гарантировано. За поворотной транзакцией следуют операции `approveTicket()` и `approveOrder()`, которые можно повторить.

Особенно важно здесь различие между компенсируемыми и повторяемыми транзакциями. Как вы увидите сами, каждый тип транзакций играет свою роль в контрмерах. В главе 13 утверждается, что при переходе на микросервисы монолит иногда должен принимать участие в повествованиях, и все будет намного проще, если ему нужно будет выполнять исключительно повторяемые транзакции.

Теперь рассмотрим каждую отдельную контрмеру, начиная с семантической блокировки.

Контрмера «семантическая блокировка»

При использовании семантической блокировки компенсируемая транзакция устанавливает флаг во всех записях, которые она создает или обновляет. Он говорит о том, что запись *не зафиксирована* и может измениться. Это может быть либо блокировка, которая закрывает доступ к записи другим транзакциям, либо предупреждение о том, что данную запись следует перепроверять. Флаг сбрасывается либо повторяемой (повествование успешно завершается), либо компенсирующей транзакцией (повествование откатывается обратно).

Поле `Order.state` — отличный пример семантической блокировки. Для ее реализации используются состояния вида `*_PENDING`, такие как `APPROVAL_PENDING` и `REVISION_PENDING`. Всем, кто обращается к заказу, они говорят о том, что в данный момент он обновляется другим повествованием. Например, на первом этапе (который является компенсируемой транзакцией) повествование `Create Order` создает заказ с состоянием `APPROVAL_PENDING`. На заключительном этапе (повторяемая транзакция) это поле меняется на `APPROVED`, а компенсирующая транзакция присваивает ему значение `REJECTED`.

Но управление блокировкой — это лишь половина проблемы. Вам также нужно решить, как каждое отдельное повествование будет обращаться с заблокированной записью. Рассмотрим в качестве примера системную команду `cancelOrder()`. С ее помощью клиент может отменить заказ, находящийся в состоянии `APPROVAL_PENDING`.

Эту задачу можно решить несколькими способами. Системная команда `cancelOrder()` может просто отказать и посоветовать клиенту повторить попытку позже. Основное преимущество этого подхода — простая реализация. А недостаток в том, что клиент усложняется за счет логики повторного вызова.

В качестве еще одного варианта команда `cancelOrder()` может сама заблокироваться до снятия блокировки. Преимущество семантических блокировок состоит в том, что они, в сущности, воссоздают уровень изолированности, обеспеченный ACID-транзакциями. Повествования, обновляющие одну и ту же запись, сериализуются, что значительно упрощает написание кода. Еще одной положительной стороной является то, что клиенту больше не нужно отвечать за повторные вызовы. Однако при этом приложение должно управлять блокировками. Оно должно также реализовать алгоритм обнаружения взаимного блокирования, который откатывает повествование, чтобы снять блокировку, и выполняет его заново.

Контрмера «коммутативные обновления»

Простой и понятной контрмерой является проектирование коммутативных операций обновления. Операции называют *коммутативными*, если их можно выполнить в любом порядке. В качестве примера можно привести команды `debit()` и `credit()` из сервиса **Accounting** (если не брать во внимание проверки перерасхода средств). Это полезная контрмера, так как она устраняет множество обновлений.

Представьте себе сценарий, в котором повествование нужно откатить после того, как компенсируемая транзакция уже сняла (или возместила) средства со счета. Компенсирующая транзакция может просто возместить (или снять) нужную сумму, чтобы отменить обновление. Возможность того, что это перезапишет обновления, сделанные другими повествованиями, отсутствует.

Контрмера «пессимистическое представление»

Справиться с недостаточной изолированностью, можно также с помощью *пессимистического представления*. Оно меняет местами этапы повествования, чтобы минимизировать бизнес-риски, связанные с «грязным» чтением. Вернемся к примеру аномалии «грязного» чтения, которую мы обсуждали ранее. В этом сценарии повествование **Create Order** выполняет «грязное» чтение доступных кредитных средств и создает заказ, превышающий кредитный лимит заказчика. Чтобы уменьшить вероятность этой ситуации, данная контрмера переставит этапы повествования следующим образом.

1. Сервис **Order**. Меняет состояние заказа на **CANCELLED**.
2. Сервис **Delivery**. Отменяет доставку.
3. Сервис **Customer**. Увеличивает доступный кредит.

В этой упорядоченной версии повествования доступные кредитные средства увеличиваются в рамках повторяемой транзакции, что исключает возможность «грязного» чтения.

Контрмера «повторное чтение значения»

Повторное чтение значения предотвращает потерю обновлений. Повествование, использующее эту контрмеру, повторно считывает запись перед ее обновлением, убеждается в том, что та не изменилась, и только потом обновляет. Если запись изменилась, повествование прекращает работу и, возможно, запускается заново. Это разновидность шаблона «Оптимистичная автономная блокировка» (<https://martinfowler.com/eaCatalog/optimisticOfflineLock.html>).

Повествование **Create Order** может применять эту контрмеру для сценария, в котором заказ отменяется в процессе подтверждения. Транзакция, подтверждающая заказ, проверяет, не изменился ли он с момента создания в текущем повествовании. Не обнаружив изменений, транзакция подтверждает заказ. Но если заказ был отменен, транзакция прерывает повествование, в результате чего выполняются его компенсирующие транзакции.

Контрмера «файл версий»

Файл версий назван так потому, что в него записываются операции, которые выполняются с записью. Благодаря этому порядок следования операций можно изменить. Это способ превращения некоммутативных обновлений в коммутативные. Чтобы показать, как работает эта контрмера, рассмотрим сценарий, в котором повествования `Create Order` и `Cancel Order` выполняются параллельно. Если здесь не применяется семантическая блокировка, существует вероятность того, что повествование `Cancel Order` отменит авторизацию банковской карты заказчика до того, как `Create Order` ее авторизует.

Чтобы справиться с этими перепутанными запросами, сервис `Accounting` может записывать операции по мере поступления и затем выполнять их в правильном порядке. В рассматриваемом случае он сначала запишет запрос `Cancel Authorization`. Затем, получив запрос `Authorize Card`, просто пропустит авторизацию банковской карты, так как у него уже есть информация о получении запроса `Cancel Authorization`.

Контрмера «по значению»

Последняя контрмера называется «по значению». Это стратегия выбора механизмов конкурентности на основе бизнес-рисков. Приложение, которое ее применяет, использует свойства всех запросов, чтобы сделать выбор между повествованиями и распределенными транзакциями. Таким образом, запросы с низким уровнем риска выполняются в виде повествований и, возможно, с помощью контрмер, описанных в предыдущем разделе. Но запросы с повышенным риском (например, связанные с большими суммами денег) задействуют распределенные транзакции. Благодаря этой стратегии приложение может динамически искать баланс между бизнес-рисками, доступностью и масштабируемостью.

При написании повествований в своем приложении вам, вероятно, придется использовать одну или несколько из этих контрмер. Давайте подробно рассмотрим архитектуру и реализацию повествования `Create Order` с применением семантической блокировки.

4.4. Архитектура сервиса `Order` и повествования `Create Order`

Итак, мы обсудили различные трудности, присущие архитектуре и реализации повествований. Теперь рассмотрим пример. Архитектура сервиса `Order` показана на рис. 4.9. Его бизнес-логика состоит из традиционных классов, которые описывают сам сервис (`OrderService`) и заказы (`Order`). Есть также классы-оркестраторы, например `CreateOrderSaga` для оркестрации повествования `Create Order`. К тому же, поскольку сервис `Order` участвует в собственных повествованиях, у него есть класс-адаптер `OrderCommandHandlers`, который обрабатывает командные сообщения, вызывая `OrderService`.

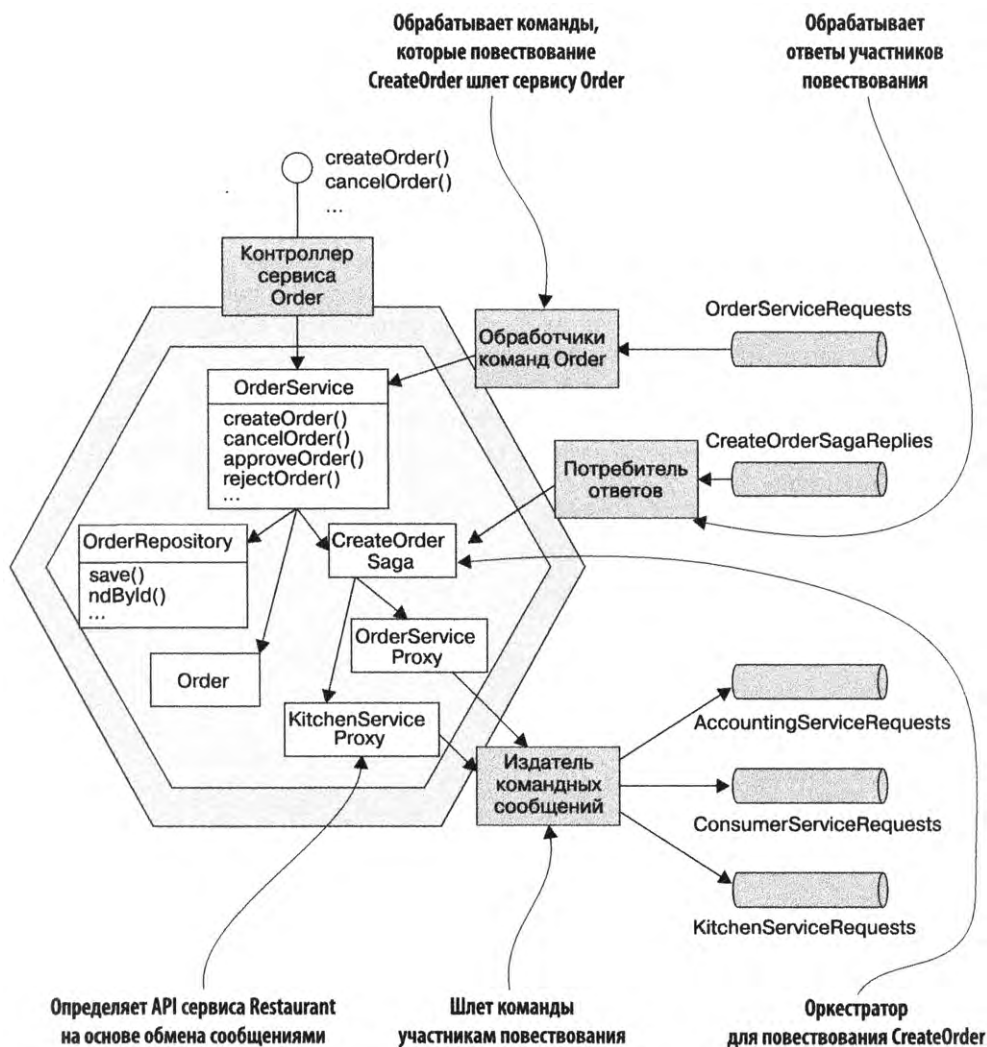


Рис. 4.9. Архитектура сервиса Order и его повествований

Некоторые участки сервиса Order должны быть вам знакомы. Как и в традиционных приложениях, ядро бизнес-логики реализуется классами OrderService, Order и OrderRepository. В этой главе мы лишь кратко по ним пройдемся, а подробное описание будет представлено в главе 5.

Менее знакомыми могут показаться классы, связанные с повествованиями. Этот сервис — одновременно и оркестратор, и участник повествований. У него есть несколько классов-оркестраторов, таких как CreateOrderSaga. Они отправляют командные сообщения участникам повествования, используя такие прокси-классы, как KitchenServiceProxy и OrderServiceProxy. Прокси-классы

описывают API обмена сообщениями участника. Сервис `Order` также содержит класс `OrderCommandHandlers`, который обрабатывает командные сообщения, отправленные его повествованиями.

Рассмотрим эту архитектуру подробнее. Начнем с класса `OrderService`.

4.4.1. Класс `OrderService`

Класс `OrderService` — это доменный сервис, который вызывается через его API. Он отвечает за создание и обновление заказов. На рис. 4.10, помимо `OrderService`, показано несколько сопутствующих классов. `OrderService` создает и обновляет заказы, сохраняет их с помощью `OrderRepository` и использует `SagaManager` для создания повествований, таких как `CreateOrderSaga`. `SagaManager` — это один из классов, входящих в состав фреймворка Eventuate Tram Saga, предназначенного для написания оркестраторов и участников повествований. Поговорим о нем чуть позже в этой главе.

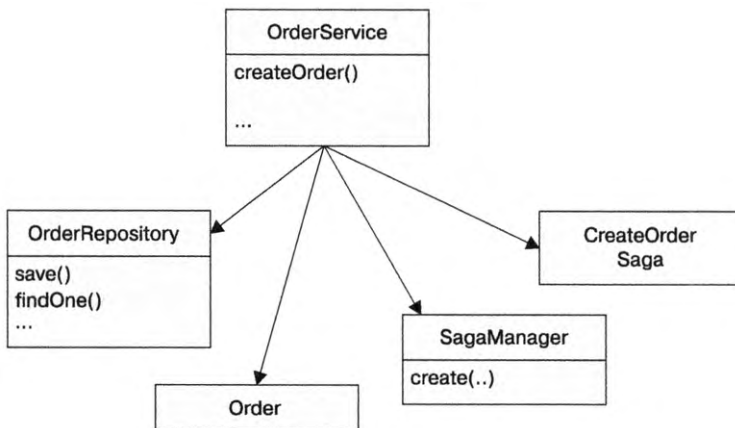


Рис. 4.10. `OrderService` создает и обновляет заказы, сохраняет их с помощью `OrderRepository` и создает повествования, включая `CreateOrderSaga`

Более подробно этот класс обсуждается в главе 5. А пока сосредоточимся на методе `createOrder()` из класса `OrderService`. Далее вы увидите его листинг 4.1. Сначала этот метод создает заказ, а затем проверяет его с помощью `CreateOrderSaga`.

Метод `createOrder()` создает заказ, вызывая фабричный метод `Order.createOrder()`. Затем сохраняет заказ с помощью класса `OrderRepository`, который представляет собой репозиторий на основе JPA. Он вызывает метод `SagaManager.create()`, чтобы создать `CreateOrderSaga`, и передает объект `CreateOrderSagaState` с идентификатором только что сохраненного заказа и информацией о нем (в виде `OrderDetails`). Создав экземпляр оркестратора повествований, класс `SagaManager` отправляет командное сообщение первому участнику и сохраняет оркестратор в базе данных.

Листинг 4.1. Класс OrderService и его метод createOrder()

```

@Transactional
public class OrderService {

    @Autowired
    private SagaManager<CreateOrderSagaState> createOrderSagaManager;

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private DomainEventPublisher eventPublisher;

    public Order createOrder(OrderDetails orderDetails) {
        ...
        ResultWithEvents<Order> orderAndEvents = Order.createOrder(...);
        Order order = orderAndEvents.result;
        orderRepository.save(order);
        eventPublisher.publish(Order.class,
                               Long.toString(order.getId()),
                               orderAndEvents.events);
        CreateOrderSagaState data =
            new CreateOrderSagaState(order.getId(), orderDetails);
        createOrderSagaManager.create(data, Order.class, order.getId());
        return order;
    }
    ...
}

```

Делаем методы сервиса транзакционными

Создаем заказ

Сохраняем заказ в базе данных

Публикуем доменные события

Создаем CreateOrderSaga

Рассмотрим повествование CreateOrderSaga и связанные с ним классы.

4.4.2. Реализация повествования Create Order

На рис. 4.11 показаны классы, реализующие повествование Create Order. Они имеют следующие обязанности.

- ❑ **CreateOrderSaga** — класс-синглтон, который описывает конечный автомат повествования. Он создает командные сообщения, вызывая **CreateOrderSagaState**, и рассылает их участникам с помощью каналов, указанных такими прокси-классами, как **KitchenServiceProxy**.
- ❑ **CreateOrderSagaState** — сохраненное состояние повествования, генерирующее командные сообщения.
- ❑ *Прокси-классы участников, такие как KitchenServiceProxy*, — каждый прокси-класс определяет API обмена сообщениями для участника повествования, который состоит из командного канала, типов командных сообщений и типов ответов.

Эти классы написаны с помощью фреймворка Eventuate Tram Saga.

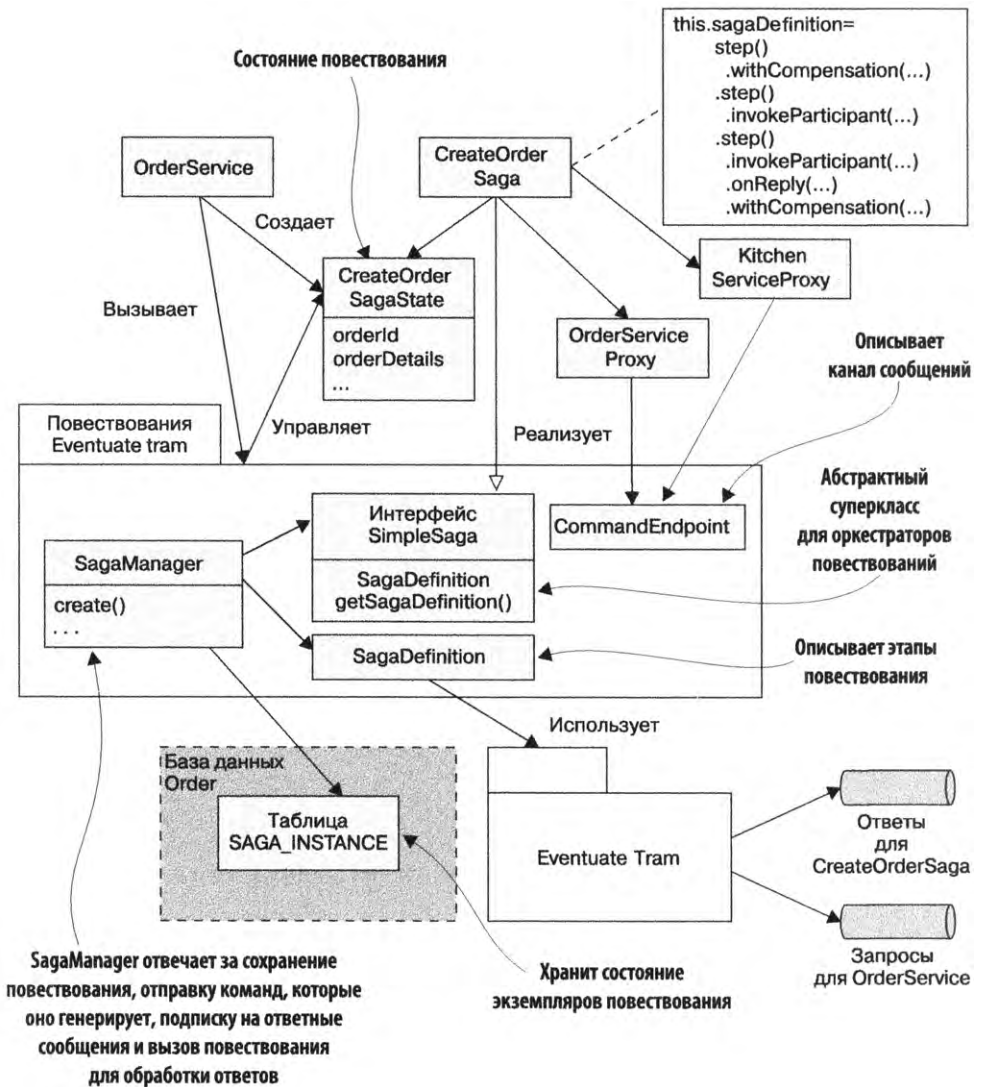


Рис. 4.11. Повествования сервиса OrderService, такие как Create Order, реализуются с помощью фреймворка Eventuate Tram Saga

Фреймворк Eventuate Tram Saga предоставляет предметно-ориентированный язык (domain-specific language, DSL) для описания конечного автомата повествования. Eventuate Tram также помогает с запуском этого конечного автомата, обменом сообщениями с участниками повествования и сохранением состояния повествования в базе данных.

Давайте подробнее обсудим реализацию повествования Create Order, начиная с класса CreateOrderSaga.

Оркестратор CreateOrderSaga

Класс `CreateOrderSaga` определяет конечный автомат, показанный ранее на рис. 4.7. Он реализует базовый интерфейс повествований, `SimpleSaga`. Его основной частью является описание повествования, представленное в листинге 4.2. Для определения этапов `Create Order` этот класс использует фреймворк Eventuate Tram Saga.

Листинг 4.2. Определение класса `CreateOrderSaga`

```
public class CreateOrderSaga implements SimpleSaga<CreateOrderSagaState> {

    private SagaDefinition<CreateOrderSagaState> sagaDefinition;

    public CreateOrderSaga(OrderServiceProxy orderService,
                           ConsumerServiceProxy consumerService,
                           KitchenServiceProxy kitchenService,
                           AccountingServiceProxy accountingService) {
        this.sagaDefinition =
            step()
                .withCompensation(orderService.reject,
                                CreateOrderSagaState::makeRejectOrderCommand)
            .step()
                .invokeParticipant(consumerService.validateOrder,
                                CreateOrderSagaState::makeValidateOrderByConsumerCommand)
            .step()
                .invokeParticipant(kitchenService.create,
                                CreateOrderSagaState::makeCreateTicketCommand)
                .onReply(CreateTicketReply.class,
                        CreateOrderSagaState::handleCreateTicketReply)
                .withCompensation(kitchenService.cancel,
                                CreateOrderSagaState::makeCancelCreateTicketCommand)
            .step()
                .invokeParticipant(accountingService.authorize,
                                CreateOrderSagaState::makeAuthorizeCommand)
            .step()
                .invokeParticipant(kitchenService.confirmCreate,
                                CreateOrderSagaState::makeConfirmCreateTicketCommand)
            .step()
                .invokeParticipant(orderService.approve,
                                CreateOrderSagaState::makeApproveOrderCommand)
            .build();
    }

    @Override
    public SagaDefinition<CreateOrderSagaState> getSagaDefinition() {
        return sagaDefinition;
    }
}
```

Конструктор `CreateOrderSaga` создает определение повествования и сохраняет его в поле `sagaDefinition`. Для возвращения этого определения используется метод `getSagaDefinition()`.

Чтобы понять, как работает `CreateOrderSaga`, рассмотрим определение третьего этапа повествования, который показан в листинге 4.3. На этом этапе для создания заявки вызывается сервис `Kitchen`. Отменяется заявка с помощью компенсирующей транзакции. Методы `step()`, `invokeParticipant()`, `onReply()` и `withCompensation()` — это часть языка DSL, предоставленного фреймворком `Eventuate Tram Saga`.

Листинг 4.3. Определение третьего этапа повествования

```
public class CreateOrderSaga ...
public CreateOrderSaga(..., KitchenServiceProxy kitchenService,
    ...) {
    ...
    .step()
    .invokeParticipant(kitchenService.create,
        CreateOrderSagaState::makeCreateTicketCommand)
    .onReply(CreateTicketReply.class,
        CreateOrderSagaState::handleCreateTicketReply)
    .withCompensation(kitchenService.cancel,
        CreateOrderSagaState::makeCancelCreateTicketCommand)
    ...
};
```

Вызываем `handleCreateTicketReply()`
при получении успешного ответа

Определяем
прямую транзакцию

Определяем
компенсирующую транзакцию

Вызов `invokeParticipant()` определяет прямую транзакцию. Он создает командное сообщение `CreateTicket`, вызывая метод `CreateOrderSagaState.makeCreateTicketCommand()`, и отправляет его в канал, заданный операцией `kitchenService.create`. В вызове `onReply()` указано, что метод `CreateOrderSagaState.handleCreateTicketReply()` должен вызываться при получении успешного ответа от сервиса `Kitchen`. Этот метод сохраняет возвращенный идентификатор `ticketId` в состоянии `CreateOrderSagaState`. Вызов `withCompensation()` определяет компенсирующую транзакцию. Он создает командное сообщение `RejectTicketCommand`, вызывая `CreateOrderSagaState.makeCancelCreateTicket()`, и отправляет его в канал, заданный операцией `kitchenService.create`.

Другие этапы повествования описываются примерно так же. `CreateOrderSagaState` создает каждое сообщение, которое повествованием отправляется в конечную точку, заданную классом `KitchenServiceProxy`. Рассмотрим каждый из этих классов, начиная с `CreateOrderSagaState`.

Класс `CreateOrderSagaState`

Класс `CreateOrderSagaState`, показанный в листинге 4.4, представляет состояние экземпляра повествования. Экземпляр этого класса создается сервисом `OrderService` и сохраняется в базе данных фреймворком `Eventuate Tram Saga`. Его основная обязанность заключается в создании сообщений, которые рассылаются участникам повествования.

Листинг 4.4. CreateOrderSagaState хранит состояние экземпляра повествования

```
public class CreateOrderSagaState {

    private Long orderId;

    private OrderDetails orderDetails;
    private long ticketId;

    public Long getOrderId() {
        return orderId;
    }

    private CreateOrderSagaState() {

    }

    public CreateOrderSagaState(Long orderId, OrderDetails orderDetails) {
        this.orderId = orderId;
        this.orderDetails = orderDetails;
    }

    CreateTicket makeCreateTicketCommand() {
        return new CreateTicket(getOrderDetails().getRestaurantId(),
                                getOrderId(), makeTicketDetails(getOrderDetails()));
    }

    void handleCreateTicketReply(CreateTicketReply reply) {
        logger.debug("getTicketId {}", reply.getTicketId());
        setTicketId(reply.getTicketId());
    }

    CancelCreateTicket makeCancelCreateTicketCommand() {
        return new CancelCreateTicket(getOrderId());
    }

    ...
}
```

Вызывается сервисом OrderService
для создания экземпляра
CreateOrderSagaState

Создает командное
сообщение CreateTicket

Сохраняет ID только что
созданной заявки

Создает командное
сообщение CancelCreateTicket

CreateOrderSaga вызывает CreateOrderSagaState для создания командных сообщений, которые затем отправляются в конечную точку, заданную классами SagaParticipantProxy. Обсудим один из этих классов — KitchenServiceProxy.

Класс KitchenServiceProxy

Класс KitchenServiceProxy, показанный в листинге 4.5, описывает конечные точки командных сообщений для сервиса Kitchen. Всего таких точек три:

- ❑ create — создает заявку;
- ❑ confirmCreate — подтверждает создание;
- ❑ cancel — отменяет заявку.

Каждая точка CommandEndpoint указывает тип команды, канал командного сообщения и типы ожидаемых ответов.

Листинг 4.5. KitchenServiceProxy описывает конечные точки командных сообщений для сервиса Kitchen

```
public class KitchenServiceProxy {

    public final CommandEndpoint<CreateTicket> create =
        CommandEndpointBuilder
            .forCommand(CreateTicket.class)
            .withChannel(
                KitchenServiceChannels.kitchenServiceChannel)
            .withReply(CreateTicketReply.class)
            .build();

    public final CommandEndpoint<ConfirmCreateTicket> confirmCreate =
        CommandEndpointBuilder
            .forCommand(ConfirmCreateTicket.class)
            .withChannel(
                KitchenServiceChannels.kitchenServiceChannel)
            .withReply(Success.class)
            .build();

    public final CommandEndpoint<CancelCreateTicket> cancel =
        CommandEndpointBuilder
            .forCommand(CancelCreateTicket.class)
            .withChannel(
                KitchenServiceChannels.kitchenServiceChannel)
            .withReply(Success.class)
            .build();
}
```

Строго говоря, прокси-классы, такие как `KitchenServiceProxy`, — необязательные. Повествование может просто отправлять командные сообщения непосредственно участникам. Однако прокси-классы обладают важными преимуществами. Во-первых, прокси-класс описывает статически типизированные конечные точки — это снижает вероятность того, что повествование отправит сервису некорректное сообщение. Во-вторых, прокси-класс представляет собой строго очерченный API для вызова сервиса, упрощающий понимание и тестирование кода. Например, в главе 10 показано, как написать тесты для `KitchenServiceProxy`, которые проверяют корректность обращения к сервису `Kitchen` со стороны сервиса `Order`. Без `KitchenServiceProxy` написание такого узкоспециализированного теста было бы невозможным.

Фреймворк Eventuate Tram Saga

Фреймворк Eventuate Tram Saga, представленный на рис. 4.12, предназначен для написания как оркестраторов, так и участников повествований. Он использует возможности транзакционного обмена сообщениями, встроенные в Eventuate Tram (см. главу 3).

Самая сложная часть данного фреймворка — пакет `saga orchestration`. Он предоставляет `SimpleSaga` — базовый интерфейс для повествований и `SagaManager` — класс

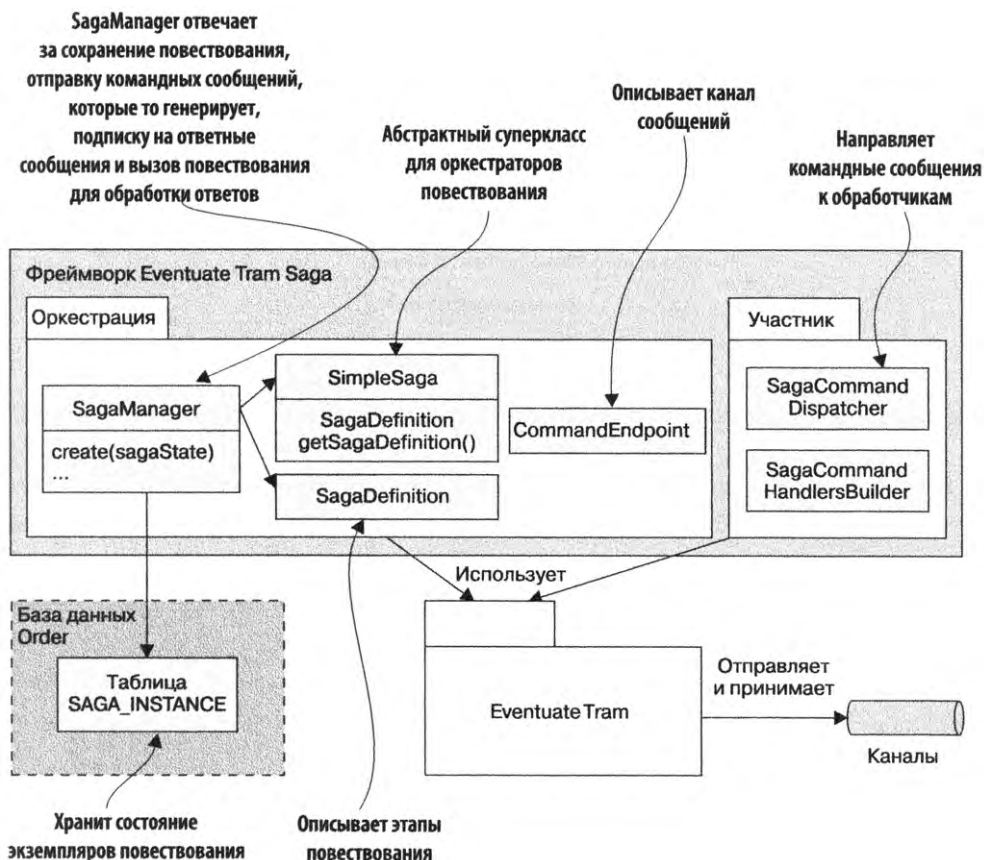


Рис. 4.12. Фреймворк Eventuate Tram Saga предназначен для написания как оркестраторов, так и участников повествований

для создания экземпляров повествования и управления ими. **SagaManager** отвечает также за сохранение повествования, отправку командных сообщений, которые оно генерирует, подписку на ответные сообщения и вызов повествования для обработки ответов. На рис. 4.13 показана последовательность событий, когда **OrderService** создает повествование. Она состоит из следующих шагов.

1. Сервис **OrderService** создает **CreateOrderSagaState**.
2. Он создает экземпляр повествования путем вызова **SagaManager**.
3. **SagaManager** выполняет первый этап из определения повествования.
4. Вызывается **CreateOrderSagaState** для генерации командного сообщения.
5. **SagaManager** шлет командное сообщение участнику повествования (сервису **Consumer**).
6. **SagaManager** сохраняет экземпляр повествования в базе данных.

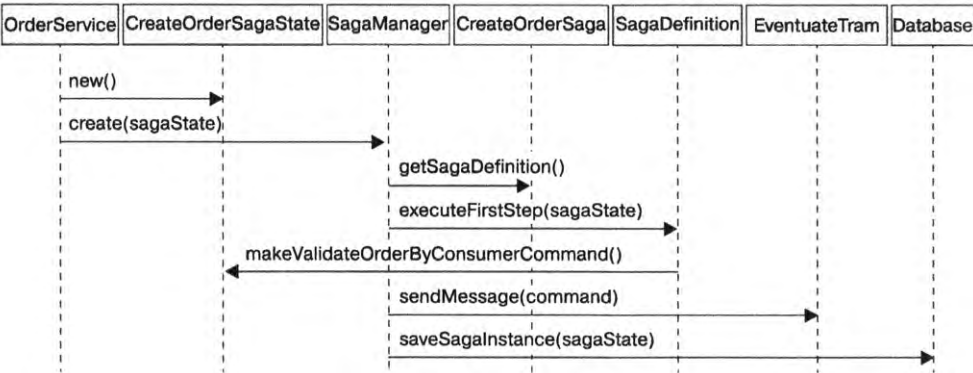


Рис. 4.13. Последовательность событий, когда OrderService создает повествование Create Order

На рис. 4.14 показана последовательность событий, когда SagaManager получает ответ от сервиса Consumer.

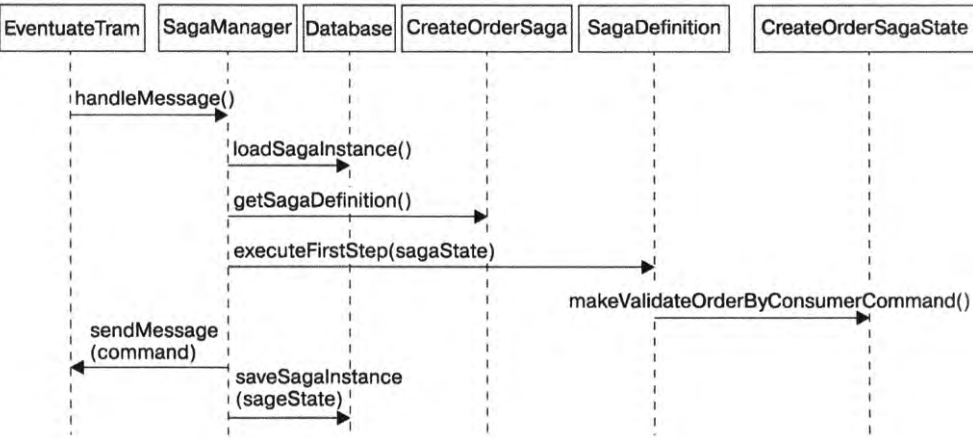


Рис. 4.14. Последовательность событий, когда SagaManager получает ответ от участника повествования

Последовательность событий выглядит так.

1. Eventuate Tram вызывает SagaManager с ответом, полученным от сервиса Consumer.
2. SagaManager извлекает экземпляр повествования из базы данных.
3. SagaManager выполняет следующий этап из определения повествования.
4. Вызывается CreateOrderSagaState, чтобы сгенерировать командное сообщение.
5. SagaManager шлет командное сообщение заданному участнику повествования (сервису Kitchen).
6. SagaManager сохраняет обновленный экземпляр повествования в базе данных.

Если участнику повествования не удастся выполнить команду, `SagaManager` вызовет компенсирующие транзакции в обратном порядке.

Еще одной частью фреймворка Eventuate Tram Saga является пакет `saga participant`. Он предоставляет классы `SagaCommandHandlersBuilder` и `SagaCommandDispatcher` для написания участников повествования. Эти классы направляют командные сообщения к методам-обработчикам, которые вызывают бизнес-логику участника и генерируют ответы. Посмотрим, как эти классы применяются в сервисе `Order`.

4.4.3. Класс `OrderCommandHandlers`

Сервис `Order` участвует в своих собственных повествованиях. Например, `CreateOrderSaga` обращается к нему, чтобы он подтвердил или отклонил заказ. Класс `OrderCommandHandlers` (рис. 4.15) определяет методы-обработчики для командных сообщений, отправляемых этими повествованиями.

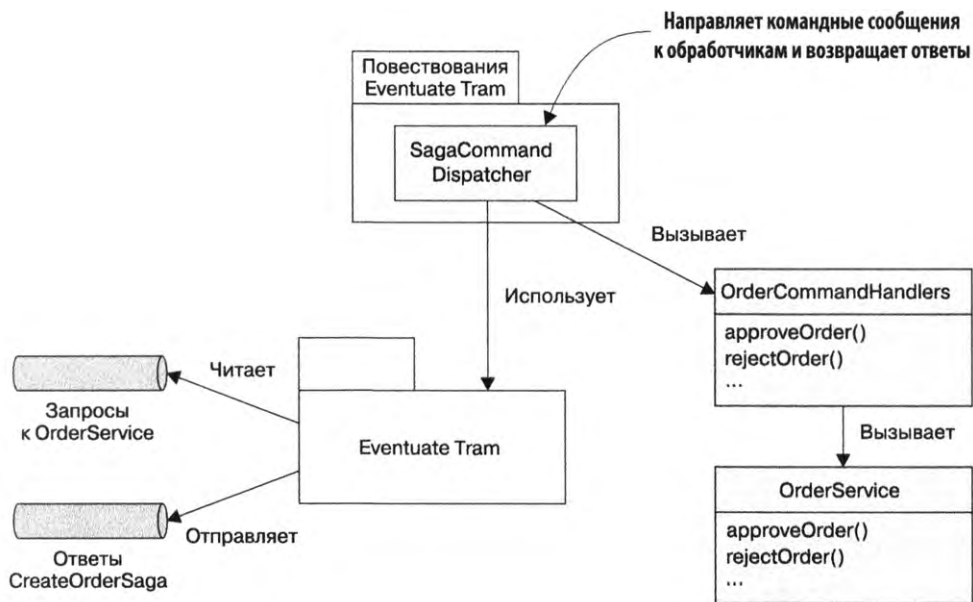


Рис. 4.15. `OrderCommandHandlers` реализует обработчики для команд, отправляемых различными повествованиями сервиса `Order`

Каждый обработчик вызывает `OrderService`, чтобы обновить заказ, и генерирует ответное сообщение. Класс `SagaCommandDispatcher` направляет командные сообщения подходящим методам и возвращает ответ.

В листинге 4.6 показан класс `OrderCommandHandlers`. Его метод `commandHandlers()` привязывает типы командных сообщений к подходящим обработчикам. Каждый обработчик принимает командное сообщение в качестве параметра, вызывает `OrderService` и возвращает ответ.

Листинг 4.6. Обработчики команд для сервиса Order

```

public class OrderCommandHandlers {

    @Autowired
    private OrderService orderService;

    public CommandHandlers commandHandlers() {
        return SagaCommandHandlersBuilder
            .fromChannel("orderService")
            .onMessage(ApproveOrderCommand.class, this::approveOrder)
            .onMessage(RejectOrderCommand.class, this::rejectOrder)
            ...
            .build();
    }

    public Message approveOrder(CommandMessage<ApproveOrderCommand> cm) {
        long orderId = cm.getCommand().getOrderId();
        orderService.approveOrder(orderId);
        return withSuccess();
    }

    public Message rejectOrder(CommandMessage<RejectOrderCommand> cm) {
        long orderId = cm.getCommand().getOrderId();
        orderService.rejectOrder(orderId);
        return withSuccess();
    }
}

```

Направляет каждое командное сообщение подходящему методу-обработчику

Меняет состояние заказа на AUTHORIZED

Возвращает стандартное сообщение об успешном выполнении

Меняет состояние заказа на REJECTED

Методы `approveOrder()` и `rejectOrder()` обновляют заданный заказ, вызывая `OrderService`. Другие сервисы, участвующие в повествовании, имеют похожие классы для обработки команд, которые обновляют их доменные объекты.

4.4.4. Класс `OrderServiceConfiguration`

Сервис `Order` использует фреймворк Spring. В листинге 4.7 показан фрагмент конфигурационного класса `OrderServiceConfiguration`, который создает экземпляры `@Bean` из состава Spring и соединяет их между собой.

Листинг 4.7. `OrderServiceConfiguration` — это конфигурационный класс из состава Spring, который определяет экземпляры `@Bean` для сервиса Order

```

@Configuration
public class OrderServiceConfiguration {

    @Bean
    public OrderService orderService(RestaurantRepository restaurantRepository,
        ...
        SagaManager<CreateOrderSagaState>
            createOrderSagaManager,
        ...) {
        return new OrderService(restaurantRepository,
            ...

```

```

        createOrderSagaManager
        ...);
    }

    @Bean
    public SagaManager<CreateOrderSagaState> createOrderSagaManager(CreateOrderSaga saga) {
        return new SagaManagerImpl<>(saga);
    }

    @Bean
    public CreateOrderSaga createOrderSaga(OrderServiceProxy orderService,
                                           ConsumerServiceProxy consumerService,
                                           ...) {
        return new CreateOrderSaga(orderService, consumerService, ...);
    }

    @Bean
    public OrderCommandHandlers orderCommandHandlers() {
        return new OrderCommandHandlers();
    }

    @Bean
    public SagaCommandDispatcher orderCommandHandlersDispatcher(OrderCommandHandlers orderCommandHandlers) {
        return new SagaCommandDispatcher("orderService", orderCommandHandlers.commandHandlers());
    }

    @Bean
    public KitchenServiceProxy kitchenServiceProxy() {
        return new KitchenServiceProxy();
    }

    @Bean
    public OrderServiceProxy orderServiceProxy() {
        return new OrderServiceProxy();
    }

    ...
}

```

Этот класс определяет несколько экземпляров `@Bean`, включая `orderService`, `createOrderSagaManager`, `createOrderSaga`, `orderCommandHandlers` и `orderCommandHandlersDispatcher`. Он также определяет экземпляры `@Bean` для различных прокси-классов, в том числе `kitchenServiceProxy` и `orderServiceProxy`.

`CreateOrderSaga` — это лишь одно из множества повествований сервиса `Order`. Многие другие его системные операции тоже используют повествования. Например, операция `cancelOrder()` задействует повествование `Cancel Order`, а `reviseOrder()` — `Revise Order`. В итоге, несмотря на наличие у многих сервисов внешних API

с синхронными протоколами, такими как REST или gRPC, большая часть интенсивного взаимодействия будет основана на асинхронных сообщениях.

Как видите, управление транзакциями и некоторые аспекты проектирования бизнес-логики в микросервисной архитектуре довольно уникальны. К счастью, оркестраторы повествований обычно представляют собой простые конечные автоматы, а для упрощения написания повествований можно использовать специальные фреймворки. Конечно, по сравнению с монолитной архитектурой управление транзакциями, несомненно, выглядит сложнее. Но это невысокая цена за огромные преимущества микросервисов.

Резюме

- ❑ Некоторым системным операциям нужно обновлять данные, разбросанные по разным сервисам. Распределенные транзакции, основанные на XA/2PC, — не самый подходящий выбор для современных приложений. Вместо них лучше использовать шаблон «Повествование». Повествование — это последовательность локальных транзакций, которые координируются с помощью сообщений. Каждая локальная транзакция обновляет данные лишь в одном сервисе. При этом все изменения фиксируются, поэтому, если повествование нужно откатить из-за нарушения бизнес-правила, оно должно выполнить компенсирующие транзакции, чтобы явно отменить внесенные изменения.
- ❑ Для координации этапов повествования можно применять либо хореографию, либо оркестрацию. В повествованиях, основанных на хореографии, локальная транзакция публикует события, которые заставляют других участников выполнить свои локальные транзакции. При использовании оркестрации централизованный оркестратор рассылает участникам сообщения с инструкциями, какие локальные транзакции нужно выполнить. Вы можете упростить разработку и тестирование, смоделировав оркестратор в виде конечного автомата. Хореография подходит для простых повествований, но в сложных случаях лучше применять оркестрацию.
- ❑ Проектирование бизнес-логики, основанной на повествованиях, может оказаться проблематичным, поскольку повествования, в отличие от ACID-транзакций, не изолированы друг от друга. В связи с этим часто приходится задействовать контрмеры — стратегии проектирования, которые предотвращают аномалии конкурентного выполнения, присущие транзакционной модели ACD. Иногда для упрощения бизнес-логики необходимо использовать блокировки, которые сами по себе чреваты взаимным блокированием.

Проектирование бизнес-логики в микросервисной архитектуре

В этой главе

- Применение шаблонов организации бизнес-логики, таких как сценарий транзакции и доменная модель.
- Проектирование бизнес-логики с помощью шаблона «Агрегат» (предметно-ориентированное проектирование).
- Применение шаблона «Доменное событие» в микросервисной архитектуре.

Сердцем промышленных приложений является бизнес-логика, которая реализует бизнес-правила. Разработка сложной бизнес-логики всегда сопряжена с определенными трудностями. Приложение FTGO реализует довольно замысловатую бизнес-логику, особенно для управления заказами и доставкой. Мэри поощряла свою команду применять принципы объектно-ориентированного проектирования, поскольку, исходя из ее опыта, это лучший способ реализации сложной бизнес-логики. На некоторых участках приложения использовался процедурный шаблон «Сценарий транзакции». Но большая часть кода была реализована в соответствии с объектно-ориентированной доменной моделью, которая накладывалась на базу данных с помощью JPA.

В микросервисной архитектуре разрабатывать сложную бизнес-логику оказывается еще труднее, потому что она распределена между разными микросервисами. Вам необходимо решить две ключевые проблемы. Типичная доменная модель выглядит как паутина из связанных между собой классов. В монолитных приложениях в этом нет ничего плохого, но в микросервисной архитектуре, где классы разбросаны по разным сервисам, нужно избавиться от ссылок на объекты, которые пересекают

границы сервисов. Еще одна проблема заключается в проектировании бизнес-логики, которая работает в рамках ограничений, накладываемых работой с транзакциями в микросервисной архитектуре. Вы можете применять ACID-транзакции внутри одного сервиса, но, как говорилось в главе 4, для обеспечения согласованности данных между сервисами следует использовать шаблон «Повествование».

К счастью, для преодоления этих трудностей можно воспользоваться шаблоном «Агрегат» из состава DDD. Он структурирует бизнес-логику приложения в виде набора агрегатов. *Агрегат* — это кластер объектов, с которыми можно обращаться как с единым целым. Есть две причины, почему агрегаты могут пригодиться при разработке бизнес-логики в микросервисной архитектуре.

- ❑ Агрегаты исключают любую возможность того, что ссылки на объекты могут выйти за рамки одного сервиса, потому что межагрегатная ссылка — это скорее значение первичного ключа, а не объектная ссылка.
- ❑ Транзакция может создать или обновить лишь один агрегат, поэтому агрегаты соответствуют ограничениям транзакционной модели микросервисов.

Благодаря этому ACID-транзакция никогда не выйдет за пределы одного сервиса.

Я начну эту главу с описания разных способов организации бизнес-логики — шаблонов «Сценарий транзакции» и «Доменная модель». Затем вы познакомитесь с концепцией агрегатов из DDD и узнаете, почему они являются хорошими строительными блоками для бизнес-логики сервисов. После этого я опишу шаблон «Доменная модель» и объясню, почему сервису следует публиковать свои события. В конце главы будут представлены несколько примеров бизнес-логики из сервисов *Kitchen* и *Order*.

Рассмотрим шаблоны организации бизнес-логики.

5.1. Шаблоны организации бизнес-логики

На рис. 5.1 показана архитектура типичного сервиса. Как говорилось в главе 2, бизнес-логика является ядром шестигранной архитектуры. Ее окружают входящие и исходящие адаптеры. *Входящий адаптер* обрабатывает запросы от клиентов и вызывает бизнес-логику. *Исходящий адаптер*, который сам вызывается бизнес-логикой, обращается к другим сервисам и приложениям.

Этот сервис состоит из бизнес-логики и следующих адаптеров:

- ❑ *адаптера REST API* — входящего адаптера, который реализует REST API для вызова бизнес-логики;
- ❑ *OrderCommandHandlers* — входящего адаптера, который потребляет из канала командные сообщения и вызывает бизнес-логику;
- ❑ *адаптера базы данных* — исходящего адаптера, который вызывается бизнес-логикой для доступа к базе данных;
- ❑ *адаптера публикации доменных событий* — исходящего адаптера, который публикует события для брокера сообщений.

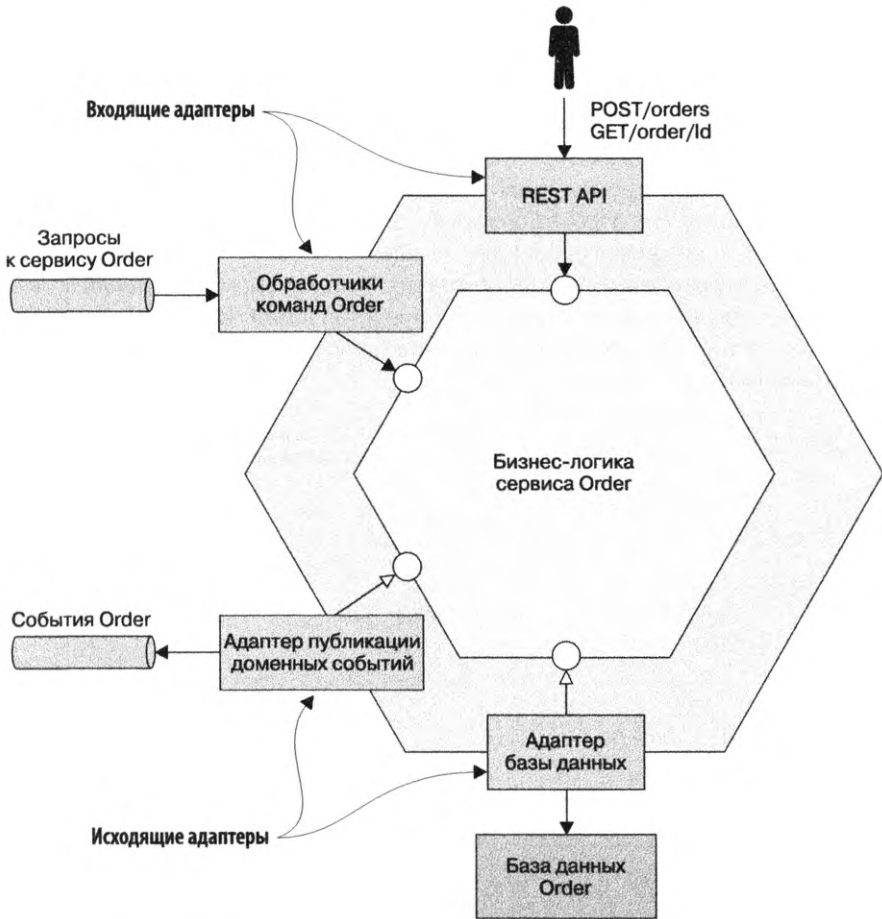


Рис. 5.1. Сервис Order имеет шестигранную архитектуру. Он состоит из бизнес-логики и одного или нескольких адаптеров для доступа к внешним приложениям или другим сервисам

Бизнес-логика обычно оказывается самой сложной частью сервиса. Вы должны осознанно организовать ее таким образом, который лучше всего подходит для вашего приложения. Я уверен, что вам уже знакомо разочарование от поддержки плохо структурированного кода, написанного кем-то другим. Большинство приложений уровня предприятия написаны на объектно-ориентированных языках, таких как Java, поэтому они состоят из классов и методов. Но использование объектно-ориентированного языка вовсе не означает, что бизнес-логика имеет объектно-ориентированную структуру. Ключевое решение, которое вам придется принять в ходе разработки, заключается в том, какой подход лучше применять — объектно-ориентированный или процедурный. Существует два основных шаблона для организации бизнес-логики: процедурный «Сценарий транзакции» и объектно-ориентированный, который называется «Доменная модель».

5.1.1. Проектирование бизнес-логики с помощью шаблона «Сценарий транзакции»

Я большой сторонник объектно-ориентированного подхода, но в некоторых ситуациях он излишен — например, при разработке простой бизнес-логики. В таких случаях лучше писать процедурный код, используя шаблон, который Мартин Фаулер в своей книге *Patterns of Enterprise Application Architecture* (Addison-Wesley Professional, 2002)¹ называет сценарием транзакции. Вместо объектно-ориентированного проектирования вы создаете метод под названием «*Сценарий транзакции*», который обрабатывает запросы из уровня представления. Важная характеристика этого подхода — то, что классы, реализующие поведение, отделены от классов, которые хранят состояние (рис. 5.2).

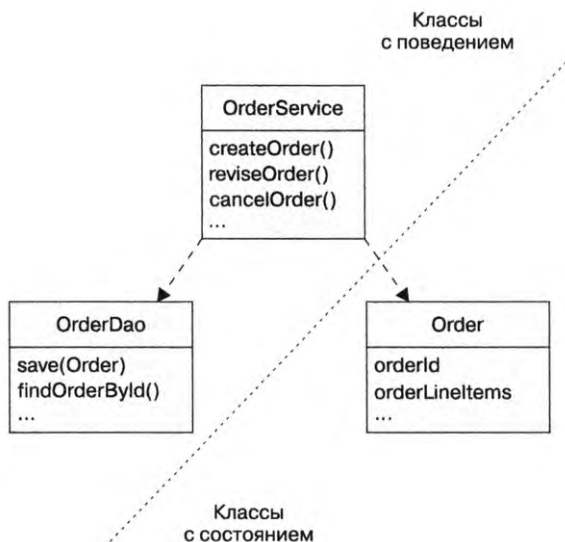


Рис. 5.2. Организация бизнес-логики в виде сценариев транзакции. В такой архитектуре один набор классов обычно реализует поведение, а другой хранит состояние. Сценарии транзакции организованы в виде классов, у которых нет состояния. Они применяют классы данных, которые, как правило, не обладают поведением

При использовании этого шаблона сценарии обычно размещаются в классе сервиса (в данном случае `OrderService`). Класс сервиса имеет по одному методу для каждого запроса или системной операции. Метод реализует бизнес-логику для определенного запроса. Он обращается к БД с помощью объектов доступа к данным (data access object, DAO), таких как `OrderDao`. Здесь примером такого объекта является класс `Order`, он предназначен исключительно для работы с данными, и у него почти (или совсем) нет никакого поведения.

¹ Фаулер М. Шаблоны корпоративных приложений. — М.: Вильямс, 2016.

Шаблон «Сценарий транзакции»

Организует бизнес-логику в виде набора процедурных сценариев транзакций, по одному для каждого типа запросов.

Этот стиль проектирования в основном является процедурным, но при этом использует несколько возможностей объектно-ориентированных языков программирования. Вы бы с помощью этого подхода писали программы на С и других языках без поддержки объектно-ориентированного программирования (ООП). Тем не менее в процедурном проектировании, если оно применяется в подходящей ситуации, нет ничего постыдного. Оно хорошо подходит для простой бизнес-логики, но сложную логику с его помощью лучше не реализовывать.

5.1.2. Проектирование бизнес-логики с помощью шаблона «Доменная модель»

Простота процедурного подхода может показаться довольно соблазнительной. Вы можете писать код без тщательного продумывания организации классов. Но проблема в том, что при довольно значительном усложнении бизнес-логики поддержка вашего кода превратится в сплошной кошмар. Как и монокитные приложения, сценарии транзакций склонны постоянно разрастаться. Поэтому, если ваше приложение не является предельно простым, следует воздерживаться от написания процедурного кода. Вместо этого стоит применять доменную модель и вести разработку в объектно-ориентированном стиле.

Шаблон «Доменная модель»

Организует бизнес-логику в виде объектной модели, состоящей из классов с состоянием и поведением.

В объектно-ориентированном проектировании бизнес-логика состоит из объектной модели — сети относительно небольших классов. Эти классы обычно напрямую соотносятся с концепциями из проблемной области. В такой архитектуре некоторые классы обладают либо состоянием, либо поведением, но многие имеют и то и другое, что является признаком хорошо спроектированного класса. Пример шаблона «Доменная модель» показан на рис. 5.3.

Как и в случае с шаблоном «Сценарий транзакции», у класса `OrderService` предусмотрено по одному методу для каждого запроса или системной операции. Но при использовании доменной модели методы сервисов обычно получаются более простыми. Это связано с тем, что существенная часть бизнес-логики делегируется доменным объектам. Метод сервиса может, например, извлечь доменный объект из базы данных и вызвать один из его собственных методов. В этом примере класс

Order обладает как состоянием, так и поведением. Более того, его состояние является приватным, и доступ к нему осуществляется лишь через его методы.

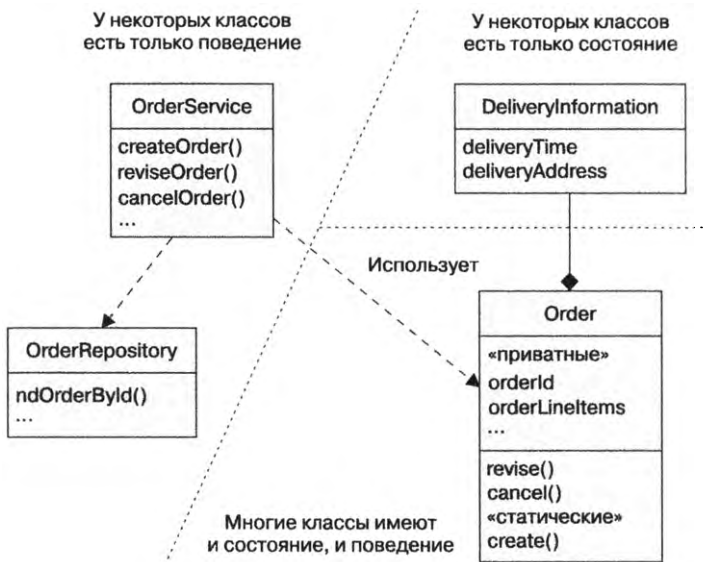


Рис. 5.3. Организация бизнес-логики в виде доменной модели. Большинство классов имеют и состояние, и поведение

Применение объектно-ориентированного проектирования имеет ряд преимуществ. Во-первых, это упрощает понимание и поддержку архитектуры. Вместо одного большого класса, который берет на себя все функции, сервис состоит из нескольких мелких классов, у каждого из которых есть свой небольшой набор обязанностей. Кроме того, такие классы, как Account, BankingTransaction и OverdraftPolicy, довольно точно отражают реальный мир, благодаря чему их роль в архитектуре проще понять. Во-вторых, нашу объектно-ориентированную архитектуру легче тестировать: каждый класс может и должен тестироваться отдельно. И наконец, объектно-ориентированный код проще расширять, поскольку в нем можно использовать хорошо известные шаблоны проектирования, такие как «Стратегия» и «Шаблонный метод», которые позволяют расширять компонент без изменения его кода.

Шаблон «Доменная модель» хорошо себя зарекомендовал, но у него есть целый ряд проблем, особенно в контексте микросервисной архитектуры. Чтобы разобраться с ними, нужно использовать более узкую версию ООП, известную как DDD.

5.1.3. О предметно-ориентированном проектировании

Предметно-ориентированное проектирование (DDD), описанное в книге Эрика Эванса *Domain-Driven Design*, — это более узкая разновидность ООП, предназначенная для разработки сложной бизнес-логики. Мы познакомились с DDD в главе 2 при обсуждении поддоменов и того, насколько они подходят для разбиения при-

ложений на сервисы. В DDD каждый сервис имеет собственную доменную модель, что позволяет избежать проблем с единой доменной моделью, которая охватывает все приложение. Поддомены и связанная с ними концепция изолированного контекста — это два стратегических шаблона DDD.

В DDD есть также тактические шаблоны, которые служат строительными блоками для доменных моделей. Каждый шаблон представляет собой роль, которую класс играет в доменной модели, и описывает характеристики этого класса. Разработчики широко применяют следующие строительные блоки.

- ❑ *Сущность* — объект, обладающий устойчивой идентичностью. Две сущности, чьи атрибуты имеют одинаковые значения, — это все равно разные объекты. В приложении Java EE классы, которые сохраняются с помощью аннотации `@Entity` из JPA, обычно представляют собой сущности DDD.
- ❑ *Объект значений* — объект, представляющий собой набор значений. Два объекта значений с одинаковыми атрибутами взаимозаменяемы. Примером таких объектов может служить класс `Money`, который состоит из валюты и суммы.
- ❑ *Фабрика* — объект или метод, реализующий логику создания объектов, которую ввиду ее сложности не следует размещать прямо в конструкторе. Фабрика также может скрывать конкретные классы, экземпляры которых создает. Она реализуется в виде статического метода или класса.
- ❑ *Репозиторий* — объект, предоставляющий доступ к постоянным сущностям и инкапсулирующий механизм доступа к базе данных.
- ❑ *Сервис* — объект, реализующий бизнес-логику, которой не место внутри сущности или объекта значений.

Многие разработчики используют эти строительные блоки. Некоторые из них поддерживаются такими фреймворками, как JPA и Spring. Но есть еще одна концепция, которую обычно игнорируют все (и я тоже!), за исключением истинных ценителей DDD. Речь идет об агрегатах. Несмотря на свою непопулярность, этот строительный блок чрезвычайно полезен при разработке микросервисов. Давайте рассмотрим некоторые неочевидные проблемы классического ООП, которые можно решить с помощью агрегатов.

5.2. Проектирование доменной модели с помощью шаблона «Агрегат» из DDD

В традиционном объектно-ориентированном проектировании доменная модель описывает набор классов и отношения между ними. Классы обычно сгруппированы в пакеты. Например, на рис. 5.4 показана часть доменной модели приложения FTGO. Это типичная доменная модель, представляющая собой паутину взаимосвязанных классов.

Этот пример содержит несколько классов, которые соотносятся с бизнес-объектами: `Consumer`, `Order`, `Restaurant` и `Courier`. Но что интересно, в традиционной доменной модели не хватает четких границ между разными бизнес-объектами.

Например, она не уточняет, какие классы входят в состав бизнес-объекта `Order`. Иногда такое расплывчатое разделение может вызвать проблемы, особенно в микросервисной архитектуре.

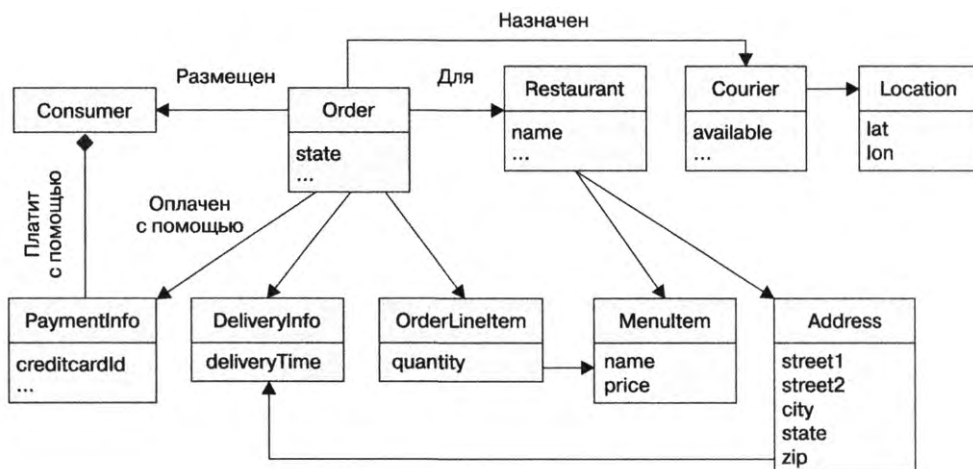


Рис. 5.4. Традиционная доменная модель представляет собой паутину взаимосвязанных классов. Она не определяет четкие границы между бизнес-объектами, такими как `Consumer` и `Order`

Я начну этот раздел с демонстрации проблем, вызванных нечеткими границами. Затем опишу концепцию агрегата и то, как она определяет четкие границы. После этого перечислю правила, которым должны подчиняться агрегаты, и покажу, как они делают эту концепцию хорошим выбором для микросервисной архитектуры. Вы также научитесь тщательно очерчивать границы для своих агрегатов и узнаете, почему это важно. В конце мы поговорим о том, как проектировать бизнес-логику с помощью агрегатов. Но сначала рассмотрим проблемы, вызванные расплывчатыми границами.

5.2.1. Проблемы с расплывчатыми границами

Представьте, к примеру, что вы хотите выполнить с бизнес-объектом `Order` операцию, такую как загрузка или удаление. Что именно это означает? Какова область применения этой операции? Загрузить или удалить объект `Order` — не проблема. Но в реальности заказ не ограничивается одним лишь этим объектом. Вам придется иметь дело с позициями заказа, информацией о платеже и т. д. На рис. 5.4 границы доменного объекта оставлены на усмотрение разработчика.

Помимо концептуальной неопределенности, отсутствие четких границ вызывает проблемы при обновлении бизнес-объекта. Типичный бизнес-объект имеет *инварианты* — бизнес-правила, которые всегда должны соблюдаться. Например, для заказа есть минимальная сумма. Приложение FTGO должно следить за тем, чтобы попытка

обновления заказа не нарушила инвариант. Но для соблюдения инвариантов вы должны тщательно спроектировать бизнес-логику.

Давайте посмотрим, как соблюсти минимальную сумму заказа в ситуации, когда он создается совместно несколькими клиентами. Два клиента, Сэм и Мэри, вместе составляют заказ и одновременно приходят к выводу, что он им не по карману. Сэм уменьшает количество пирожков, а Мэри отказывается от нескольких пшеничных лепешек. С точки зрения приложения оба клиента запрашивают заказ и его позиции из базы данных. Затем оба обновляют определенную позицию, чтобы снизить цену. С точки зрения каждого отдельного клиента сумма заказа не опустилась ниже допустимой отметки. Вот как выглядит последовательность транзакций.

<div>Consumer - Mary</div> <div>BEGIN TXN</div> <div>SELECT ORDER_TOTAL FROM ORDER WHERE ORDER ID = X</div> <div>SELECT * FROM ORDER_LINE_ITEM WHERE ORDER_ID = X</div> <div>...</div> <div>END TXN</div> <div>Verify minimum is met</div>	<div>Consumer - Sam</div> <div>BEGIN TXN</div> <div>SELECT ORDER_TOTAL FROM ORDER WHERE ORDER ID = X</div> <div>SELECT * FROM ORDER_LINE_ITEM WHERE ORDER_ID = X</div> <div>...</div> <div>END TXN</div>
<div>BEGIN TXN</div> <div>UPDATE ORDER_LINE_ITEM SET VERSION=..., QUANTITY=... WHERE VERSION = <loaded version> AND ID = ...</div> <div>END TXN</div>	<div>Verify minimum is met</div> <div>BEGIN TXN</div> <div>UPDATE ORDER_LINE_ITEM SET VERSION=..., QUANTITY=... WHERE VERSION = <loaded version> AND ID = ...</div> <div>END TXN</div>

Каждый клиент изменяет позицию заказа, используя последовательность из двух транзакций. Первая транзакция загружает заказ и его позиции. Затем пользовательский интерфейс проверяет, достигнута ли минимальная сумма заказа. После этого вторая транзакция обновляет количество единиц в заданной позиции, задействуя проверку с оптимистичной автономной блокировкой, чтобы убедиться в неизменности позиции заказа с момента его загрузки первой транзакцией.

В этом сценарии Сэм уменьшает общую сумму заказа на X долларов, а Мэри — на Y долларов. В итоге заказ оказывается недействительным, хотя после обновления каждым из клиентов приложение подтвердило, что сумма заказа не опустилась ниже минимальной отметки. Как видите, обновление частей бизнес-объекта напрямую может вылиться в нарушение бизнес-правил. Агрегаты из состава DDD призваны решить эту проблему.

5.2.2. Агрегаты имеют четкие границы

Агрегат — это кластер доменных объектов, с которыми можно обращаться как с единым целым. Он состоит из корневой сущности и иногда одной или нескольких сущностей и объектов значений. Многие бизнес-объекты моделируются в виде агрегатов. Например, в главе 2 мы создали доменную модель общего вида, проанализировав имена существительные, которые использовались в бизнес-требованиях и среди специалистов в данной области. Многие из этих имен существительных, такие как `Order`, `Consumer` и `Restaurant`, являются агрегатами.

Шаблон «Агрегат»

Организует доменную модель в виде набора агрегатов — графов объектов, с которыми можно работать как с единым целым.

Агрегат `Order` и его границы показаны на рис. 5.5. Он состоит из сущности `Order` и одного или нескольких объектов значений, таких как `OrderLineItem`, `Address` и `PaymentInformation`.

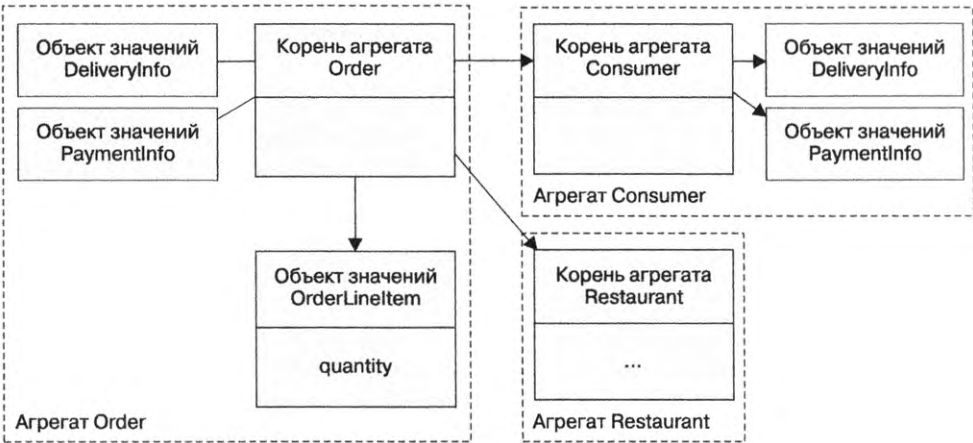


Рис. 5.5. Структурирование доменной модели в виде набора агрегатов создает четкие границы

Агрегаты разбивают доменную модель на блоки, в которых легче разобраться по отдельности. Они также проясняют область применения операций, таких как загрузка, обновление и удаление. Эти операции распространяются на весь агрегат, а не на какие-то его части. Агрегат часто загружается из базы данных целиком, что позволяет избежать любых проблем с ленивой загрузкой. Вместе с агрегатом из базы данных удаляются все его объекты.

Агрегаты — это границы согласованности

Обновление целого агрегата, а не отдельных его частей решает проблемы с согласованностью, подобных описанным в предыдущем примере. Операции обновления вызываются для корня агрегата, который обеспечивает соблюдение инвариантов. Кроме того, чтобы поддерживать конкурентность, корень агрегата блокируется с помощью, скажем, номера версии или блокировки уровня базы данных. Например, вместо непосредственного обновления количества единиц для определенных позиций клиент должен вызвать метод из корня агрегата `Order`, который следит за соблюдением таких инвариантов, как минимальная сумма заказа. Однако следует упомянуть, что этот подход не требует обновления всего агрегата в базе данных. Приложение может, к примеру, обновить поля, относящиеся к заказу `Order` и измененному объекту `OrderLineItem`.

Главное — определить агрегаты

В DDD ключевым аспектом проектирования доменной модели является определение агрегатов, их границ и корней. Детали внутренней структуры агрегатов вторичны. Однако преимущества этого подхода далеко не ограничены разделением доменной модели на модули. Причина этого в том, что агрегаты обязаны придерживаться определенных правил.

5.2.3. Правила для агрегатов

DDD требует, чтобы агрегаты подчинялись набору правил. Это делает агрегат автономной единицей, способной обеспечивать соблюдение инвариантов. Рассмотрим эти правила.

Правило 1. Ссылайтесь только на корень агрегата

Предыдущий пример проиллюстрировал риски прямого обновления объекта `OrderLineItems`. Первое правило призвано устранить эту проблему. Оно требует, чтобы корневая сущность была единственной частью агрегата, на которую могут ссылаться внешние классы. Для обновления агрегата клиенту необходимо вызвать метод из его корня.

Например, сервис использует репозиторий, чтобы загрузить агрегат из базы данных и получить ссылку на его корень. С помощью метода, вызываемого из корня, он обновляет агрегат. Это правило гарантирует, что агрегат способен обеспечивать соблюдение своих инвариантов.

Правило 2. Межагрегатные ссылки должны применять первичные ключи

Правило состоит в том, что агрегаты ссылаются друг на друга по уникальному значению, например по первичному ключу, а не по объектным ссылкам. На рис. 5.6 показано, как заказ ссылается на своего заказчика с помощью `consumerId`, а не ссылки на объект `Consumer`. Аналогичным образом заказ ссылается на ресторан с использованием `restaurantId`.

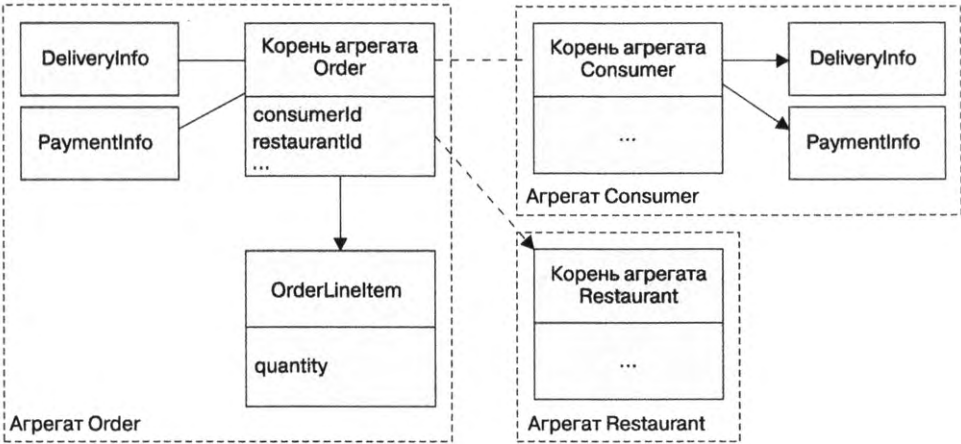


Рис. 5.6. Агрегаты ссылаются друг на друга с помощью первичных ключей, а не объектных ссылок. У агрегата Order есть идентификаторы агрегатов Consumer и Restaurant. Внутри одного агрегата объекты могут ссылаться друг на друга

Этот подход заметно отличается от традиционного моделирования объектов, с точки зрения которого использование внешних ключей в доменной модели — плохое архитектурное решение. Данный метод имеет ряд преимуществ. Отказ от объектных ссылок в пользу уникальных идентификаторов означает, что агрегаты слабо связаны между собой. Это позволяет четко определить границы между ними и избежать случайного обновления не того агрегата. К тому же нам не нужно беспокоиться об объектных ссылках, которые выходят за пределы одного сервиса.

Этот подход также упрощает сохранение состояния, поскольку агрегат является единицей хранения. Благодаря этому агрегаты становятся легче хранить в базах данных NoSQL, таких как MongoDB. К тому же это устраняет необходимость в прозрачной ленивой загрузке и проблемы, которые с ней связаны. Масштабирование базы данных путем сегментирования агрегатов — довольно простая задача.

Правило 3. Одна транзакция создает или обновляет один агрегат

Еще одно правило, которому должны подчиняться агрегаты, состоит в том, что транзакция может создать или обновить только один агрегат. Когда я впервые прочитал о нем много лет назад, оно показалось мне бессмысленным! В то время я занимался разработкой традиционных монолитных приложений с использованием СУРБД, поэтому в моем случае транзакции могли обновлять множественные агрегаты. Но в наши дни это ограничение идеально подходит для микросервисной архитектуры. Оно гарантирует, что транзакция не выйдет за пределы сервиса. А также хорошо согласуется с ограниченной транзакционной моделью большинства баз данных NoSQL.

Это правило усложняет реализацию операций, которым нужно создавать или обновлять несколько агрегатов. Но это явно одна из тех проблем, для решения которых предназначены повествования, описанные в главе 4. Каждый этап повествования создает или обновляет ровно один агрегат. На рис. 5.7 показано, как это работает.

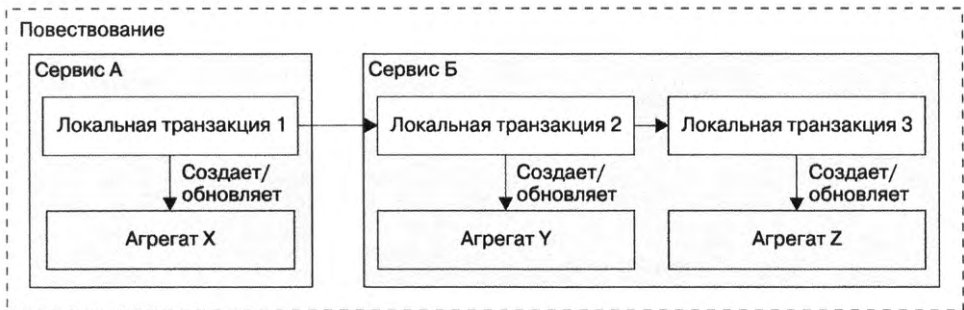


Рис. 5.7. Транзакция может создать или обновить только один агрегат, поэтому для обновления нескольких агрегатов приложение использует повествования. Каждый этап повествования создает или обновляет один агрегат

В этом примере повествование состоит из трех транзакций. Первая транзакция обновляет агрегат X в сервисе А. Две другие происходят в сервисе В: одна транзакция обновляет агрегат Y, а другая — агрегат Z.

Есть и альтернативный вариант. Чтобы обеспечить согласованность между несколькими агрегатами внутри одного сервиса, мы могли бы «сжульничать» и обновить все эти агрегаты в рамках одной транзакции. Например, одной транзакции могло бы быть достаточно для обновления агрегатов Y и Z в сервисе В. Но это возможно только в СУРБД с развитой транзакционной моделью. Если вы применяете базу данных NoSQL, которая поддерживает только простые транзакции, у вас нет другого варианта, кроме как использовать повествования.

Но так ли это? Оказывается, границы между агрегатами не высечены в камне. При разработке доменной модели вы сами можете определить, где они должны проходить. Но, помня о том, как колониальные державы чертили государственные границы в прошлом веке, вы должны подходить к этому осторожно.

5.2.4. Размеры агрегатов

При разработке доменной модели важно решить, насколько большим вы хотите сделать тот или иной агрегат. С одной стороны, в идеале агрегаты должны быть мелкими. Это увеличит количество одновременных запросов, которые может обработать ваше приложение, и улучшит масштабируемость, поскольку обновления каждого агрегата сериализуются. Это также положительно скажется на опыте взаимодействия, так как снижается вероятность того, что два пользователя попытаются внести конфликтующие изменения в один и тот же агрегат. Но с другой стороны, агрегат — это область применения транзакции, поэтому, чтобы обеспечить атомарность определенного обновления, иногда стоит сделать его более крупным.

Ранее я упоминал о том, что в доменной модели приложения FTGO `Order` и `Consumer` являются отдельными агрегатами. В качестве альтернативы мы могли бы сделать `Order` частью `Consumer` (рис. 5.8).

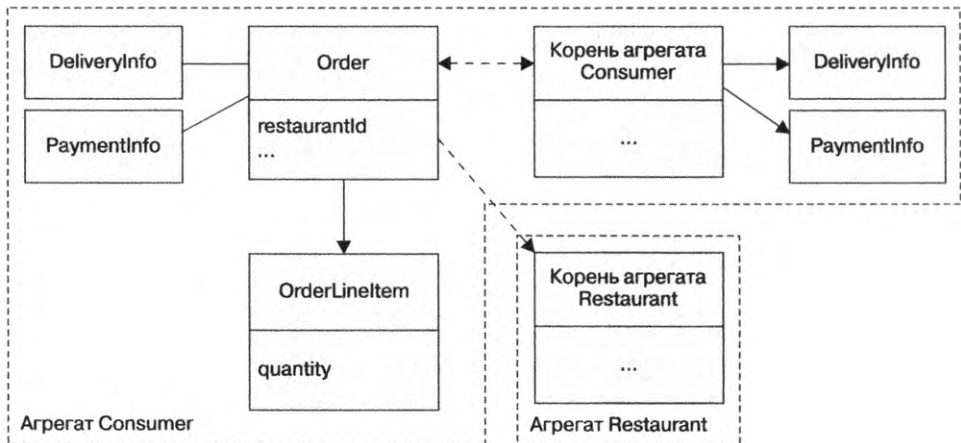


Рис. 5.8. Альтернативное решение описывает агрегат `Customer`, который включает в себя классы `Customer` и `Order`. Такой подход позволяет приложению атомарно обновлять заказчика и один или несколько его заказов

Преимущество этого более крупного агрегата `Consumer` заключается в том, что приложение может атомарно обновлять заказчика и один или несколько его заказов. Недостатком этого подхода является ухудшение масштабируемости. Транзакции, обновляющие разные заказы для одного клиента, будут сериализованы. К тому же, если два пользователя попытаются отредактировать разные заказы одного клиента, получится конфликт.

Еще одним отрицательным аспектом крупных агрегатов в контексте микросервисной архитектуры является то, что они препятствуют декомпозиции. Например, бизнес-логика для заказов и клиентов должна находиться в одном сервисе, что делает этот сервис более объемным. Учитывая эти проблемы, агрегаты лучше делать как можно более мелкими.

5.2.5. Проектирование бизнес-логики с помощью агрегатов

В типичном (микро)сервисе основная часть бизнес-логики состоит из агрегатов. Остальной код принадлежит доменным сервисам и повествованиям. Повествования оркестрируют цепочки локальных транзакций, чтобы обеспечить согласованность данных. Сервисы служат точками входа в бизнес-логику и вызываются входящими адаптерами. Сервис использует репозиторий для извлечения агрегатов или их сохранения в базу данных. Каждый репозиторий реализуется исходящим адаптером, который обращается к БД. Структура бизнес-логики сервиса `Order` на основе агрегатов показана на рис. 5.9.

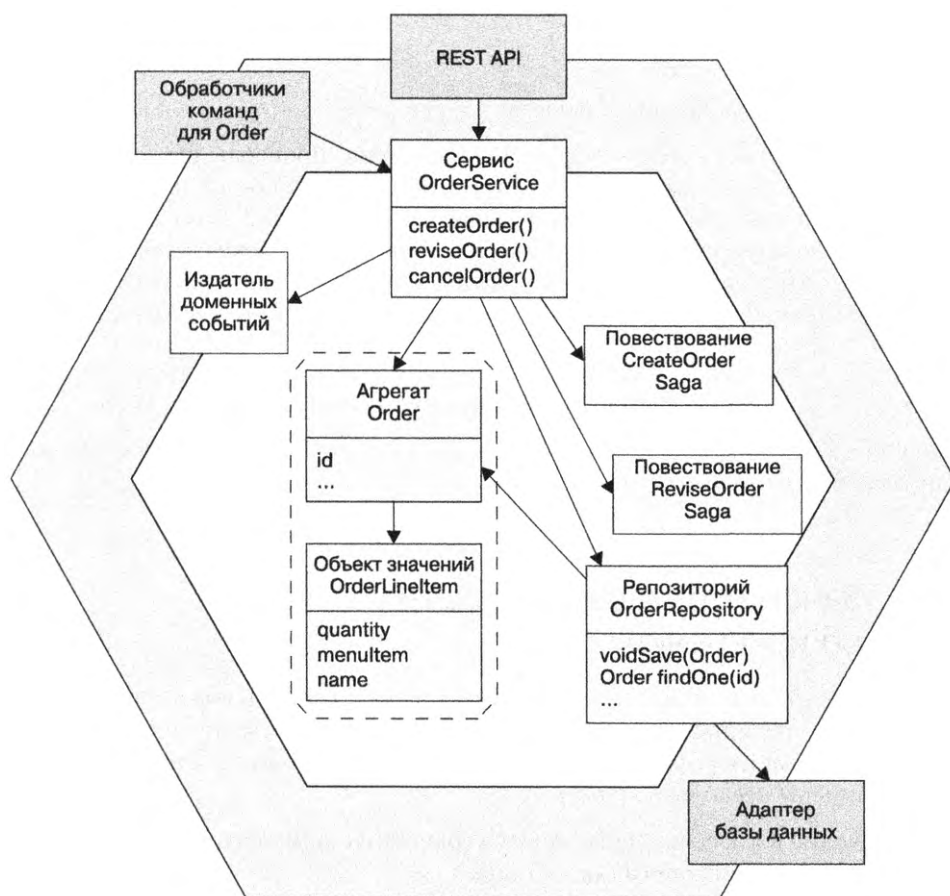


Рис. 5.9. Структура бизнес-логики сервиса `Order` на основе агрегатов

Бизнес-логика состоит из агрегата `Order`, класса сервиса `OrderService`, репозитория `OrderRepository` и одного или нескольких повествований. `OrderService`

обращается к `OrderRepository`, чтобы сохранять и загружать заказы. Если сервис получает простые запросы, которые не выходят за его рамки, он обновляет агрегат `Order`. Если запрос охватывает несколько сервисов, `OrderService` создает повествование, как было описано в главе 4.

Прежде чем переходить к коду, рассмотрим концепцию, тесно связанную с агрегатами, — доменные события.

5.3. Публикация доменных событий

В толковом словаре Merriam-Webster (<https://www.merriam-webster.com/dictionary/event>) перечислено несколько определений слова «событие», включая следующие:

- ❑ что-то, что случается;
- ❑ примечательный случай;
- ❑ социальный повод или деятельность.

В контексте DDD доменное событие — это нечто произошедшее с агрегатом. В доменной модели оно имеет вид класса. Событие обычно представляет изменение состояния. Возьмем, к примеру, агрегат `Order` в приложении FTGO. В число событий, которые меняют его состояние, входят `Order Created`, `Order Cancelled`, `Order Shipped` и т. д. При наличии заинтересованных потребителей агрегат `Order` может публиковать одно из этих событий в момент изменения своего состояния.

Шаблон «Доменное событие»

Агрегат публикует доменное событие во время своего создания или в ходе какого-то другого существенного изменения.

5.3.1. Зачем публиковать события об изменениях

Полезность доменных событий связана с тем, что другие стороны взаимодействия (пользователи, внешние приложения или другие компоненты внутри того же приложения) часто заинтересованы в информации об изменениях в состоянии агрегата. Далее приведены несколько примеров.

- ❑ Обеспечение согласованности между сервисами с помощью повествований на основе хореографии, описанных в главе 4.
- ❑ Уведомление сервиса, который хранит реплику, об изменении в источнике данных. Этот подход, рассмотренный в главе 7, известен как разделение ответственности командных запросов (CQRS).
- ❑ Уведомление другого приложения через зарегистрированный веб-хук или брокер сообщений, чтобы инициировать следующий этап в бизнес-процессе.

- ❑ Уведомление другого компонента того же приложения, чтобы, к примеру, послать браузеру пользователя сообщение через WebSocket или обновить текстовую базу данных, такую как Elasticsearch.
- ❑ Рассылка пользователям уведомлений (текстовых сообщений или электронных писем) об отправке их заказа, готовности медицинского рецепта или задержке самолета.
- ❑ Мониторинг доменных событий для проверки корректности поведения системы.
- ❑ Анализ событий для моделирования поведения пользователя.

Во всех этих сценариях причиной уведомления служит изменение состояния агрегата в базе данных приложения.

5.3.2. Что такое доменное событие

Доменное событие — это класс с именем на основе страдательного причастия прошедшего времени. Он содержит свойства, которые выразительно передают это событие. Каждое свойство представляет собой либо простое значение, либо объект. Например, класс события `OrderCreated` содержит свойство `orderId`.

У доменного события обычно есть метаданные, такие как его идентификатор и временная метка. Оно может нести в себе идентификатор пользователя, который сделал изменение, поскольку это полезно для аудита. Метаданные могут быть частью объекта события — возможно, определенные в родительском классе. Или же они могут находиться внутри обертки вокруг объекта события. Идентификатор агрегата, который сгенерировал событие, тоже может быть не его непосредственным свойством, а частью обертки.

`OrderCreated` является примером доменного события. У него нет никаких полей, поскольку идентификатор заказа входит в состав обертки. В листинге 5.1 показаны класс `DomainEventEnvelope` и класс события `OrderCreated`.

Листинг 5.1. Событие `OrderCreated` и класс `DomainEventEnvelope`

```
interface DomainEvent {}

interface OrderDomainEvent extends DomainEvent {}

class OrderCreated implements OrderDomainEvent {}

class DomainEventEnvelope<T extends DomainEvent> {
    private String aggregateType;
    private Object aggregateId;
    private T event;
    ...
}
```

← Метаданные-события

`DomainEvent` — это интерфейс-маркер, который определяет класс как доменное событие. `OrderDomainEvent` — это интерфейс-маркер для таких событий, как `OrderCreated`, которые публикуются агрегатом `Order`. `DomainEventEnvelope` — это класс, который содержит объект события и его метаданные. Это обобщенный класс, параметризованный типом доменного события.

5.3.3. Обогащение события

Допустим, вы пишете потребитель событий, который обрабатывает события `Order`. Класс события `OrderCreated`, показанный ранее, отражает суть произошедшего. Но при обработке события `OrderCreated` вашему потребителю могут понадобиться подробности о заказе. Он может извлечь эту информацию из класса `OrderService`. Но недостатком запрашивания агрегата у сервиса являются накладные расходы на выполнение этого запроса.

В качестве альтернативы можно использовать *обогащение событий* — это когда события содержат информацию, которая нужна потребителю. В результате потребители событий становятся более простыми, поскольку им больше не нужно запрашивать данные у сервиса, опубликовавшего событие. Агрегат `Order` может обогатить событие `OrderCreated`, включив в него информацию о заказе. Обогащенный вариант `OrderCreated` показан в листинге 5.2.

Листинг 5.2. Обогащенное событие `OrderCreated`

```
class OrderCreated implements OrderEvent {
    private List<OrderLineItem> lineItems;
    private DeliveryInformation deliveryInformation;
    private PaymentInformation paymentInformation;
    private long restaurantId;
    private String restaurantName;
    ...
}
```

Данные, которые обычно
необходимы потребителям

Эта версия события `OrderCreated` содержит подробности о заказе, поэтому потребителям, таким как `OrderHistoryService` из главы 7, больше не нужно запрашивать данные при его обработке.

Обогащение событий упрощает потребителей, но его обратной стороной является риск снижения стабильности классов событий. Эти классы потенциально могут меняться каждый раз, когда обновляются требования их потребителей. Это способно отрицательно сказаться на поддерживаемости, поскольку изменения такого рода могут затронуть несколько частей приложения. К тому же удовлетворение требований каждого потребителя может оказаться недостижимой целью. К счастью, во многих ситуациях довольно легко определить, какие свойства следует включить в событие.

Теперь, рассмотрев основы доменных событий, мы можем поговорить о том, как их обнаруживать.

5.3.4. Определение доменных событий

Есть несколько разных стратегий для определения доменных событий. Часто сценарии, в которых необходимы уведомления, описываются в требованиях к приложению. Требования могут быть сформулированы так: «Если произойдет X, сделайте Y». Например, вот одно из требований к приложению FTGO: «Если размещен заказ, отправьте клиенту электронное письмо». Это указывает на существование доменного события.

Еще один подход, который набирает популярность, называется событийным штурмом. *Событийный шторм* — это формат собрания для обсуждения сложного домена, сосредоточенный вокруг событий. Специалисты в предметной области собираются в комнате с большой доской для рисования или рулоном бумаги, куда будут крепиться небольшие записки. Результатом этого процесса становится событийная доменная модель, состоящая из агрегатов и событий.

Событийный шторм имеет три этапа.

1. *Интенсивное определение событий.* Специалистов в проблемной области просят определить доменные события. Это будет выглядеть как цепочка из оранжевых записок, выложенная на поверхности для моделирования.
2. *Определение причин событий.* Специалистов в проблемной области просят определить причину каждого события, которая может быть одной из следующих:
 - действие пользователя (представлено в виде команды на синей записке);
 - внешняя система (фиолетовая записка);
 - другое доменное событие;
 - истечение времени.
3. *Определение агрегатов.* Специалистов в проблемной области просят определить агрегат, который потребляет каждую команду и генерирует соответствующее событие. Агрегаты представлены в виде желтых записок.

Результат собрания с событийным штурмом показан на рис. 5.10. Всего за несколько часов участники определили множество доменных событий, команд и агрегатов. Это был отличный первый шаг на пути к созданию доменной модели.



Рис. 5.10. Результат собрания с событийным штурмом, которое длилось пару часов. Записки, которые здесь видны, — это события, выложенные вдоль временного графика, команды, представляющие действия пользователей, и агрегаты, которые генерируют события в ответ на команды

Событийный шторм помогает быстро разработать доменную модель.

Разобравшись с доменными событиями, мы можем приступить к механизму их генерации и публикации.

5.3.5. Генерация и публикация доменных событий

Взаимодействие с помощью доменных событий — это разновидность асинхронного обмена сообщениями, который мы обсудили в главе 3. Но прежде, чем бизнес-логика сможет опубликовать событие для брокера сообщений, она должна его создать. Посмотрим, как это делается.

Генерация доменных событий

На концептуальном уровне доменные события публикуются агрегатами. Агрегат знает, когда меняется его состояние и, следовательно, какое событие нужно опубликовать. Агрегат может обращаться непосредственно к API для обмена сообщениями. Недостаток этого подхода связан с тем, что агрегат не поддерживает внедрение зависимостей, из-за чего API для обмена сообщениями придется передавать в метод в виде аргумента. Это приведет к переплетению инфраструктуры и бизнес-логики, что крайне нежелательно.

Более подходящим подходом будет разделение ответственности между агрегатом и сервисом (или эквивалентным классом), который к нему обращается. Сервисы могут использовать внедрение зависимостей для получения ссылки на API для обмена сообщениями, что позволяет им легко публиковать события. Агрегат генерирует события при изменении своего состояния и возвращает их сервису (последняя часть может быть реализована несколькими способами). Один из вариантов — включить в значение, возвращаемое методом агрегата, список событий. Например, в листинге 5.3 показано, как метод `accept()` агрегата `Ticket` возвращает событие `TicketAcceptedEvent` вызывающей стороне.

Листинг 5.3. Метод `accept()` агрегата `Ticket`

```
public class Ticket {

    public List<DomainEvent> accept(ZonedDateTime readyBy) {
        ...
        this.acceptTime = ZonedDateTime.now(); ← Обновляем Ticket
        this.readyBy = readyBy;
        return singletonList(new TicketAcceptedEvent(readyBy)); ← Возвращаем событие
    }
}
```


Сервис вызывает метод из корня агрегата и затем публикует события. Например, в листинге 5.4 показано, как `KitchenService` публикует события после вызова `Ticket.accept()`.

Листинг 5.4. KitchenService вызывает Ticket.accept()

```
public class KitchenService {

    @Autowired
    private TicketRepository ticketRepository;

    @Autowired
    private DomainEventPublisher domainEventPublisher;
    public void accept(long ticketId, ZonedDateTime readyBy) {
        Ticket ticket =
            ticketRepository.findById(ticketId)
                .orElseThrow(() ->
                    new TicketNotFoundException(ticketId));
        List<DomainEvent> events = ticket.accept(readyBy);
        domainEventPublisher.publish(Ticket.class, orderId, events);
    }
```



Вначале метод `accept()` обращается к `TicketRepository`, чтобы загрузить `Ticket` из базы данных. Затем он обновляет `Ticket`, вызывая другой метод `accept()`. После этого `KitchenService` публикует события, возвращенные агрегатом `Ticket`, вызывая метод `DomainEventPublisher.publish()`, который мы вскоре рассмотрим.

Это довольно простой подход. Методы, которые прежде имели бы пустой тип возвращаемого значения, теперь возвращают `List<Event>`. Единственный потенциальный недостаток связан с тем, что возвращаемые типы непустых методов стали сложнее. Пример такого метода вы увидите чуть позже.

Еще один вариант — накапливать события в поле корня агрегата. После этого сервис извлекает эти события и публикует их. В листинге 5.5 показана разновидность класса `Ticket`, которая работает по этому принципу.

Листинг 5.5. Ticket наследует абстрактный класс, который записывает доменные события

```
public class Ticket extends AbstractAggregateRoot {

    public void accept(ZonedDateTime readyBy) {
        ...
        this.acceptTime = ZonedDateTime.now();
        this.readyBy = readyBy;
        registerDomainEvent(new TicketAcceptedEvent(readyBy));
    }

}
```

`Ticket` наследует класс `AbstractAggregateRoot`, в котором определен метод `registerDomainEvent()`, записывающий события. Для извлечения этих событий сервису нужно сделать вызов `AbstractAggregateRoot.getDomainEvents()`.

Лично я отдаю предпочтение первому варианту, в котором метод возвращает события сервису. Но накапливание событий в корне агрегата — тоже жизнеспособный подход. Так называемый поезд релиза (release train) Spring Data Ingalls (spring.io/blog/2017/01/30/what-s-new-in-spring-data-release-ingalls) реализует механизм,

который автоматически публикует события в контексте приложения Spring. Основной недостаток этой методики в том, что для уменьшения дублирования кода корни агрегатов должны наследовать такой родительский класс, как `AbstractAggregateRoot`, а это может противоречить требованиям к наследованию каких-нибудь других классов. Есть еще одна проблема: вызывать `registerDomainEvent()` из методов корня агрегата довольно удобно, однако делать это из других классов агрегата оказывается не так просто. В большинстве случаев для этого пришлось бы передавать события в корень.

Надежная публикация доменных событий

В главе 3 обсуждалась надежная отправка сообщений в рамках локальных транзакций базы данных. Доменные события ничем в этом смысле не отличаются. Чтобы события публиковались как часть транзакции, которая обновляет агрегат в базе данных, сервис должен использовать транзакционный обмен сообщениями. Такой механизм реализован в фреймворке Eventuate Tram, описанном в главе 3. Он вставляет события в таблицу `OUTBOX` в рамках ACID-транзакции, которая обновляет базу данных. После фиксации транзакции события, вставленные в таблицу `OUTBOX`, публикуются для брокера сообщений.

Фреймворк Eventuate Tram предоставляет интерфейс `DomainEventPublisher`, показанный в листинге 5.6. Он определяет несколько перегруженных методов `publish()`, которые принимают в качестве параметров тип и ID агрегата, а также список доменных событий.

Листинг 5.6. Интерфейс `DomainEventPublisher` из фреймворка Eventuate Tram

```
public interface DomainEventPublisher {
    void publish(String aggregateType, Object aggregateId,
        List<DomainEvent> domainEvents);
```

Для транзакционной публикации этих событий здесь используется интерфейс `MessageProducer` из состава Eventuate Tram.

Сервис мог бы вызывать `DomainEventPublisher` напрямую. Но так у нас не было бы способа гарантировать, что все публикуемые события действительны. Например, сервис `KitchenService` должен публиковать только события, которые реализуют `TicketDomainEvent` — интерфейс-маркер для событий агрегата `Ticket`. Вместо этого сервису лучше реализовать подкласс `AbstractAggregateDomainEventPublisher`, показанный в листинге 5.7, — абстрактный класс, который предоставляет типобезопасный интерфейс для публикации доменных событий. Этот класс также является обобщенным и имеет два типизированных параметра: `A` (тип агрегата) и `E` (тип интерфейса-маркера для доменных событий). Сервис публикует события путем вызова метода `publish()`, который принимает два параметра: агрегат типа `A` и список событий типа `E`.

Листинг 5.7. Родительский класс типобезопасных издателей событий

```
public abstract class AbstractAggregateDomainEventPublisher<A, E extends
DomainEvent> {
```

```

private Function<A, Object> idSupplier;
private DomainEventPublisher eventPublisher;
private Class<A> aggregateType;

protected AbstractAggregateDomainEventPublisher(
    DomainEventPublisher eventPublisher,
    Class<A> aggregateType,
    Function<A, Object> idSupplier) {
    this.eventPublisher = eventPublisher;
    this.aggregateType = aggregateType;
    this.idSupplier = idSupplier;
}

public void publish(A aggregate, List<E> events) {
    eventPublisher.publish(aggregateType, idSupplier.apply(aggregate),
        (List<DomainEvent>) events);
}
}

```

Метод `publish()` извлекает ID агрегата и вызывает `DomainEventPublisher.publish()`. В листинге 5.8 показан класс `TicketDomainEventPublisher`, который публикует доменные события для агрегата `Ticket`.

Листинг 5.8. Типобезопасный интерфейс для публикации доменных событий агрегата `Ticket`

```

public class TicketDomainEventPublisher extends
    AbstractAggregateDomainEventPublisher<Ticket, TicketDomainEvent> {

    public TicketDomainEventPublisher(DomainEventPublisher eventPublisher) {
        super(eventPublisher, Ticket.class, Ticket::getId);
    }
}

```

Этот класс публикует только события, наследованные от `TicketDomainEvent`.

Итак, мы разобрались с тем, как публиковать доменные события. Теперь посмотрим, как их потреблять.

5.3.6. Потребление доменных событий

В конечном счете доменные события доходят до брокера (например, Apache Kafka) в виде сообщений. Потребитель может напрямую воспользоваться клиентским API брокера. Но удобнее использовать API, такой как `DomainEventDispatcher` из состава фреймворка Eventuate Tram (см. главу 3). `DomainEventDispatcher` направляет доменные события к подходящему методу-обработчику. Пример класса для обработки событий показан в листинге 5.9. Класс `KitchenServiceEventConsumer` подписывается на события, публикуемые сервисом `Restaurant` при каждом обновлении меню ресторана. Он отвечает за поддержание в актуальном состоянии реплики сервиса `Kitchen`.

Листинг 5.9. Передача событий методам-обработчикам

```
public class KitchenServiceEventConsumer {
    @Autowired
    private RestaurantService restaurantService;

    public DomainEventHandlers domainEventHandlers() {
        return DomainEventHandlersBuilder
            .forAggregateType("net.chrisrichardson.ftgo.restaurantservice.Restaurant")
            .onEvent(RestaurantMenuRevised.class, this::reviseMenu)
            .build();
    }

    public void reviseMenu(DomainEventEnvelope<RestaurantMenuRevised> de) {
        long id = Long.parseLong(de.getAggregateId());
        RestaurantMenu revisedMenu = de.getEvent().getRevisedMenu();
        restaurantService.reviseMenu(id, revisedMenu);
    }
}
```

Сопоставляет события
с обработчиками

Обработчик для событий
RestaurantMenuRevised

Метод `reviseMenu()` обрабатывает события `RestaurantMenuRevised`. Он вызывает метод `restaurantService.reviseMenu()`, который обновляет меню ресторана и возвращает список доменных событий, опубликованных обработчиком.

Обсудив агрегаты и доменные события, мы можем перейти к некоторым примерам бизнес-логики, реализованной на основе агрегатов.

5.4. Бизнес-логика сервиса Kitchen

Первым примером будет сервис `Kitchen`, который позволяет ресторану управлять своими заказами. Два основных агрегата этого сервиса — `Restaurant` и `Ticket`. Первый умеет проверять заказы и знает меню ресторана и его время работы, а второй представляет собой заказ, который ресторан должен подготовить для доставки курьером. Эти агрегаты и другие части бизнес-логики сервиса, включая его адаптеры, показаны на рис. 5.11.

Помимо агрегатов, важными элементами бизнес-логики сервиса `Kitchen` являются классы `KitchenService`, `TicketRepository` и `RestaurantRepository`. `KitchenService` — это точка входа в бизнес-логику. Она определяет методы для создания и обновления агрегатов `Restaurant` и `Ticket`. Классы `TicketRepository` и `RestaurantRepository` определяют методы для сохранения экземпляров `Ticket` и `Restaurant` соответственно.

У сервиса `Kitchen` есть три входящих адаптера:

- ❑ **REST API** — REST API, к которому через пользовательский интерфейс обращаются работники ресторана. Он вызывает `KitchenService` для создания и обновления заявок;

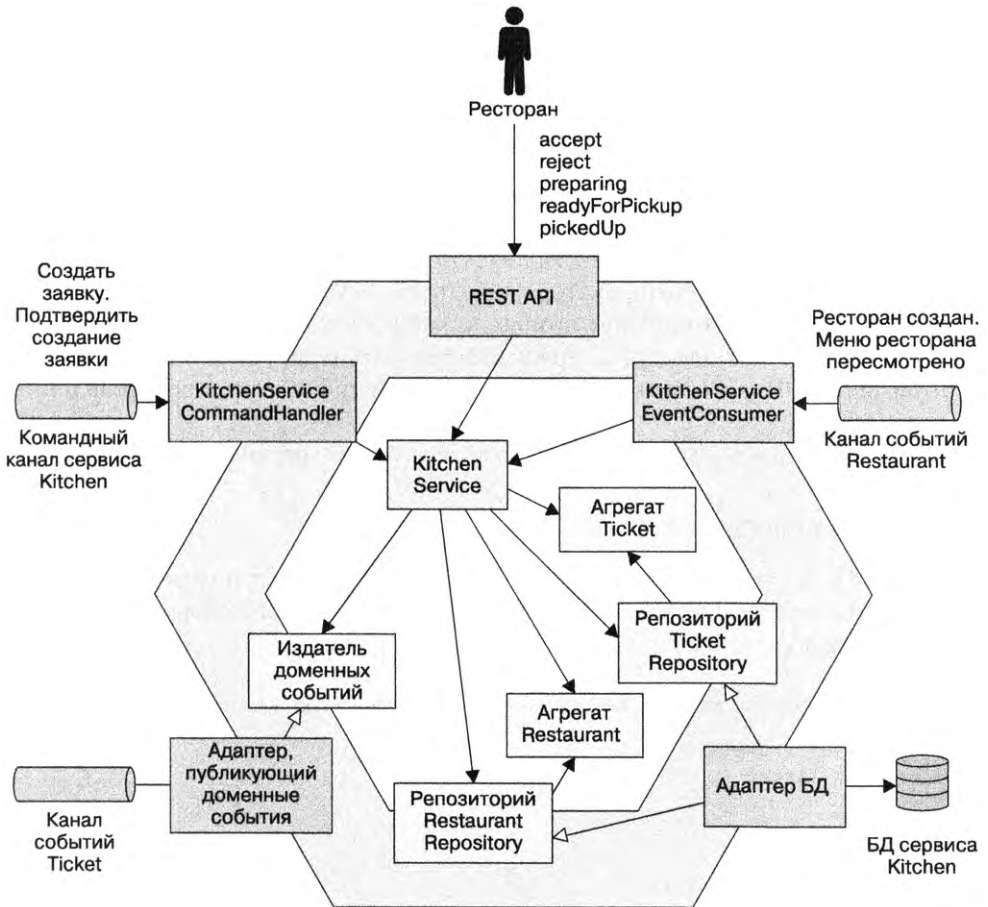


Рис. 5.11. Структура сервиса Kitchen

- ❑ `KitchenServiceCommandHandler` — API, основанный на асинхронных запросах/ответах, который вызывается повестованиями. Для создания и обновления заявок он обращается к `KitchenService`;
- ❑ `KitchenServiceEventConsumer` — подписывается на события, публикуемые сервисом `Restaurant`. Он вызывает `KitchenService` для создания и обновления экземпляров `Restaurant`.

У этого сервиса есть также два исходящих адаптера:

- ❑ *адаптер БД* — реализует интерфейсы `TicketRepository` и `RestaurantRepository`, обращается к базе данных;
- ❑ *DomainEventPublishingAdapter* — реализует интерфейс `DomainEventPublisher` и публикует доменные события `Ticket`.

Давайте поближе познакомимся со структурой `KitchenService`, начиная с агрегата `Ticket`.

5.4.1. Агрегат `Ticket`

`Ticket` — это один из агрегатов сервиса `Kitchen`. Как упоминалось в главе 2 при обсуждении изолированного контекста, этот агрегат является представлением заказа с точки зрения кухни ресторана. Он не содержит никакой информации о заказчике — ни его идентификатора, ни адреса доставки, ни платежных данных. Он сосредоточен на том, чтобы помочь кухне ресторана подготовить заказ для доставки курьером. Более того, сервис `Kitchen` не генерирует уникальный ID для этого агрегата. Вместо этого он использует идентификатор, предоставленный сервисом `Order`.

Сначала взглянем на структуру этого класса, а затем рассмотрим его методы.

Структура класса `Ticket`

В листинге 5.10 показан фрагмент кода из этого класса. `Ticket` похож на традиционный доменный класс. Его основная особенность — то, что ссылки на другие агрегаты выполнены в виде первичных ключей.

Листинг 5.10. Фрагмент класса `Ticket`, который является сущностью JPA

```
@Entity(table="tickets")
public class Ticket {

    @Id
    private Long id;
    private TicketState state;
    private Long restaurantId;

    @ElementCollection
    @CollectionTable(name="ticket_line_items")
    private List<TicketLineItem> lineItems;

    private ZonedDateTime readyBy;
    private ZonedDateTime acceptTime;
    private ZonedDateTime preparingTime;
    private ZonedDateTime pickedUpTime;
    private ZonedDateTime readyForPickupTime;
    ...
}
```

Этот класс сохраняется с помощью JPA и накладывается на таблицу `TICKETS`. Поле `restaurantId` имеет тип `Long` и не является объектной ссылкой на `Restaurant`. Поле `readyBy` хранит примерное время готовности заказа к отправке. У класса `Ticket` есть несколько полей, которые отслеживают историю заказа, включая `acceptTime`, `preparingTime` и `pickupTime`. Обсудим эти методы.

Поведение агрегата Ticket

Агрегат `Ticket` определяет несколько методов. Как вы видели ранее, у него есть статический метод `create()`, который является фабрикой для создания заявок. Есть также методы, которые вызываются, когда ресторан обновляет состояние заказа:

- ❑ `accept()` — ресторан принял заказ;
- ❑ `preparing()` — ресторан начал подготовку заказа. Это означает, что заказ больше нельзя изменить или отменить;
- ❑ `readyForPickup()` — заказ готов к отправке.

Некоторые из методов `Ticket` показаны в листинге 5.11.

Листинг 5.11. Некоторые из методов класса `Ticket`

```
public class Ticket {

    public static ResultWithAggregateEvents<Ticket, TicketDomainEvent>
        create(Long id, TicketDetails details) {
        return new ResultWithAggregateEvents<>(new Ticket(id, details), new
            TicketCreatedEvent(id, details));
    }

    public List<TicketPreparationStartedEvent> preparing() {
        switch (state) {
            case ACCEPTED:
                this.state = TicketState.PREPARING;
                this.preparingTime = ZonedDateTime.now();
                return singletonList(new TicketPreparationStartedEvent());
            default:
                throw new UnsupportedOperationException(state);
        }
    }

    public List<TicketDomainEvent> cancel() {
        switch (state) {
            case CREATED:
            case ACCEPTED:
                this.state = TicketState.CANCELLED;
                return singletonList(new TicketCancelled());
            case READY_FOR_PICKUP:
                throw new TicketCannotBeCancelledException();
            default:
                throw new UnsupportedOperationException(state);
        }
    }
}
```

Метод `create()` создает заявку. Метод `preparing()` вызывается, когда ресторан начинает подготовку заказа. Он меняет состояние заказа на `PREPARING`, записывает время и публикует событие. Метод `cancel()` вызывается, когда пользователь

пытается отменить заказ. Если отмена возможна, он меняет состояние заказа и возвращает событие, в противном случае генерирует исключение. Эти методы вызываются в ответ на запросы к REST API, а также события и командные сообщения. Посмотрим, какие классы вызывают методы агрегата.

Доменный сервис KitchenService

Класс `KitchenService` вызывается входящими адаптерами сервиса. Он определяет различные методы для изменения состояния заказа, включая `accept()`, `reject()`, `preparing()` и др. Каждый метод загружает заданный агрегат, делает подходящий вызов к корню агрегата и публикует все доменные события, которые возникли в процессе. В листинге 5.12 показан метод `accept()`.

Листинг 5.12. Метод сервиса `accept()` обновляет `Ticket`

```
public class KitchenService {

    @Autowired
    private TicketRepository ticketRepository;

    @Autowired
    private TicketDomainEventPublisher domainEventPublisher;

    public void accept(long ticketId, ZonedDateTime readyBy) {
        Ticket ticket =
            ticketRepository.findById(ticketId)
                .orElseThrow(() ->
                    new TicketNotFoundException(ticketId));

        List<TicketDomainEvent> events = ticket.accept(readyBy);
        domainEventPublisher.publish(ticket, events);
    }
}
```

Публикуем
доменные события

Метод `accept()` вызывается, когда ресторан принимает новый заказ. У него есть два параметра:

- ❑ `orderId` — идентификатор принимаемого заказа;
- ❑ `readyBy` — предполагаемое время готовности заказа к доставке.

Этот метод извлекает агрегат `Ticket`, вызывает из него другой метод `accept()` и публикует любые сгенерированные события.

Теперь рассмотрим класс, который обрабатывает асинхронные команды.

Класс `KitchenServiceCommandHandler`

Класс `KitchenServiceCommandHandler` — это адаптер, который отвечает за обработку командных сообщений, отправляемых различными повествованиями сервиса `Order`. Для каждой команды этот класс определяет свой метод-обработчик, который об-

ращается к `KitchenService` для создания или обновления `Ticket`. Фрагмент этого класса показан в листинге 5.13.

Листинг 5.13. Обработка командных сообщений, отправляемых повестованиями

```
public class KitchenServiceCommandHandler {

    @Autowired
    private KitchenService kitchenService;

    public CommandHandlers commandHandlers() {
        return CommandHandlersBuilder
            .fromChannel("orderService")
            .onMessage(CreateTicket.class, this::createTicket)
            .onMessage(ConfirmCreateTicket.class,
                this::confirmCreateTicket)
            .onMessage(CancelCreateTicket.class,
                this::cancelCreateTicket)
            .build();
    }

    private Message createTicket(CommandMessage<CreateTicket>
                                cm) {

        CreateTicket command = cm.getCommand();
        long restaurantId = command.getRestaurantId();
        Long ticketId = command.getOrderId();
        TicketDetails ticketDetails =
            command.getTicketDetails();

        try {
            Ticket ticket =
                kitchenService.createTicket(restaurantId,
                                           ticketId, ticketDetails);

            CreateTicketReply reply =
                new CreateTicketReply(ticket.getId());
            return withSuccess(reply);
        } catch (RestaurantDetailsVerificationException e) {
            return withFailure();
        }
    }

    private Message confirmCreateTicket
        (CommandMessage<ConfirmCreateTicket> cm) {
        Long ticketId = cm.getCommand().getTicketId();
        kitchenService.confirmCreateTicket(ticketId);
        return withSuccess();
    }

    ...
}
```

Сопоставляет командные сообщения с обработчиками

Обращается к `KitchenService` для создания заявки

Возвращает успешный ответ

Возвращает ответ с отказом

Подтверждает заказ

Все методы обработки команд обращаются к `KitchenService` и возвращают либо успешный ответ, либо информацию об отказе.

Итак, вы познакомились с бизнес-логикой относительно простого сервиса. Теперь рассмотрим более сложный пример — сервис `Order`.

5.5. Бизнес-логика сервиса Order

Как упоминалось в начальных главах, сервис `Order` предоставляет API для создания, обновления и отмены заказов. Этот интерфейс в основном вызывается заказчиком. На рис. 5.12 показана общая структура сервиса. Центральным агрегатом `OrderService` является `Order`. Есть также агрегат `Restaurant`, который представляет собой частичную копию данных, принадлежащих сервису `Restaurant`. Он позволяет сервису `Order` проверять и оценивать позиции заказа.

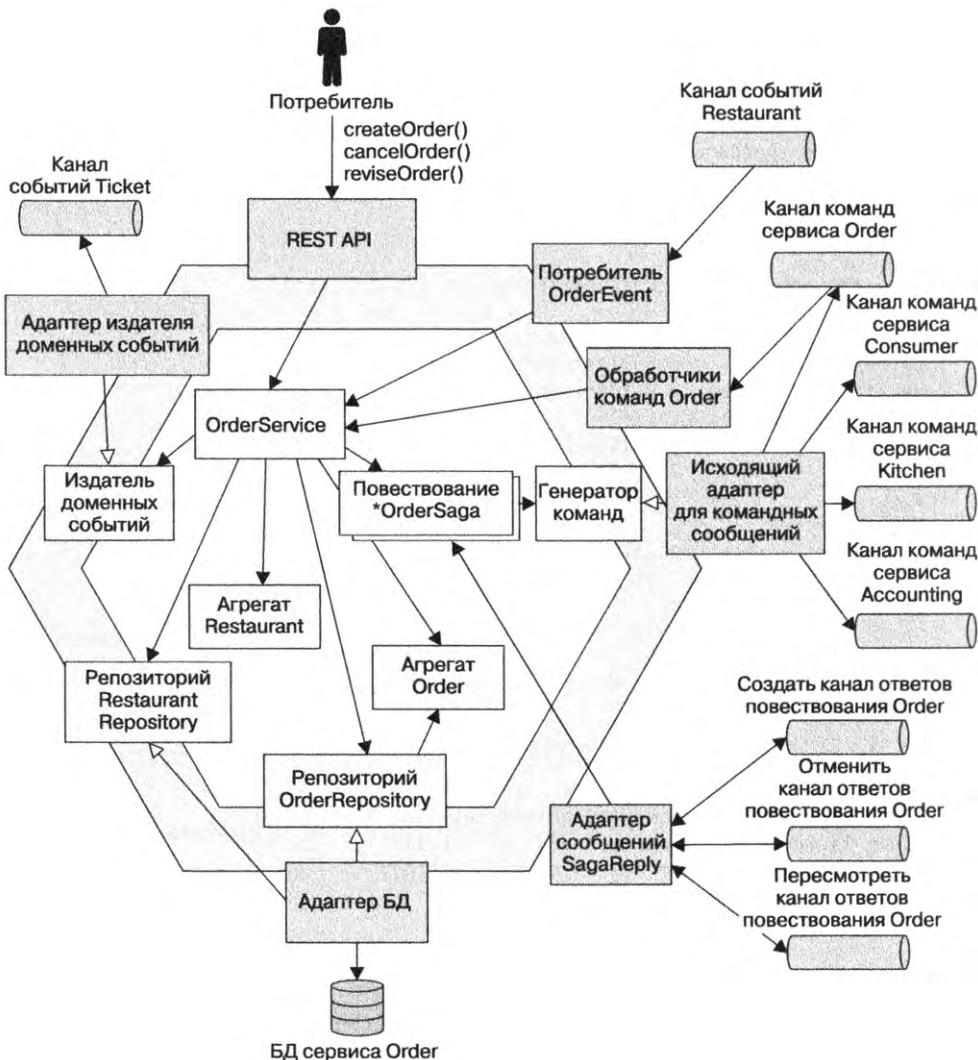


Рис. 5.12. Структура сервиса `Order`. Он имеет REST API для управления заказами и обменивается сообщениями и событиями с другими сервисами, используя несколько каналов сообщений

Помимо агрегатов `Order` и `Restaurant`, бизнес-логика имеет классы `OrderService`, `OrderRepository` и `RestaurantRepository`, а также повествования наподобие `CreateOrderSaga`, которое было описано в главе 4. `OrderService` является основной точкой входа в бизнес-логику и определяет методы для создания и обновления экземпляров `Order` и `Restaurant`. `OrderRepository` определяет методы для сохранения заказов, а у `RestaurantRepository` есть методы для сохранения агрегатов `Restaurant`. Сервис `Order` обладает несколькими входящими адаптерами.

- ❑ **REST API** — REST API, к которому через пользовательский интерфейс обращаются заказчики. Для создания и обновления заказов он вызывает `OrderService`.
- ❑ **OrderEventConsumer** — подписывается на события, публикуемые сервисом `Restaurant`. Он обращается к `OrderService` для создания и обновления своих копий агрегатов `Restaurant`.
- ❑ **OrderCommandHandlers** — API, основанный на асинхронных запросах/ответах, который вызывается повествованиями. Для обновления заказов он обращается к `OrderService`.
- ❑ **SagaReplyAdapter** — обращается к повествованию, предварительно подписавшись на канал его ответов.

У этого сервиса также есть исходящие адаптеры:

- ❑ **адантер БД** — реализует интерфейс `OrderRepository` и обращается к базе данных `OrderService`;
- ❑ **DomainEventPublishingAdapter** — реализует интерфейс `DomainEventPublisher` и публикует доменные события `Order`;
- ❑ **OutboundCommandMessageAdapter** — реализует интерфейс `CommandPublisher` и рассылает командные сообщения участникам повествования.

Сначала поближе рассмотрим агрегат `Order`, а затем исследуем `OrderService`.

5.5.1. Агрегат Order

Агрегат `Order` представляет собой заказ, размещенный клиентом. Вначале мы обсудим его структуру, а после этого перейдем к его методам.

Структура агрегата Order

Структура агрегата `Order` показана на рис. 5.13. Его корнем является класс `Order`. В состав агрегата входят также объекты значений, такие как `OrderLineItem`, `DeliveryInfo` и `PaymentInfo`.

У класса `Order` есть набор элементов `OrderLineItems`. К остальным двум своим агрегатам, `Consumer` и `Restaurant`, он обращается по значению первичного ключа. Внутри `Order` находятся еще два класса: `DeliveryInfo`, который хранит адрес и желательное время доставки, и `PaymentInfo`, содержащий информацию о платеже. Код показан в листинге 5.14.

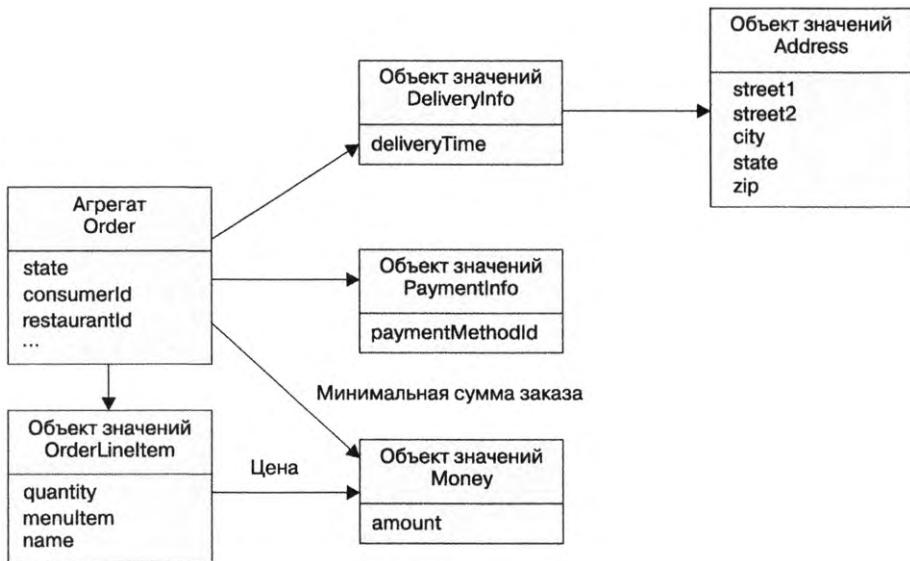


Рис. 5.13. Структура агрегата Order, который состоит из корня и различных объектов значений

Листинг 5.14. Класс Order и его поля

```
@Entity
@Table(name="orders")
@Access(AccessType.FIELD)
public class Order {

    @Id
    @GeneratedValue
    private Long id;

    @Version
    private Long version;

    private OrderState state;
    private Long consumerId;
    private Long restaurantId;

    @Embedded
    private OrderLineItems orderLineItems;

    @Embedded
    private DeliveryInformation deliveryInformation;

    @Embedded
    private PaymentInformation paymentInformation;

    @Embedded
    private Money orderMinimum;
```

Этот класс сохраняется с помощью JPA и накладывается на таблицу `ORDERS`. Поле `id` служит первичным ключом. Поле `version` используется для оптимистичного блокирования. Состояние заказа представлено перечислением `OrderState`. Поля `DeliveryInformation` и `PaymentInformation` связываются с помощью аннотации `@Embedded` и хранятся в виде столбцов таблицы `ORDERS`. Поле `orderLineItems` — это встроенный объект, который содержит позиции заказа. Агрегат `Order` состоит более чем из одного поля. Он также реализует бизнес-логику, которая может быть описана конечным автоматом. Посмотрим, как это выглядит.

Конечный автомат агрегата Order

Для создания и обновления заказов сервис `Order` должен взаимодействовать с другими сервисами, используя повествования. Метод, который проверяет возможность выполнения операции и меняет состояние заказа на `PENDING`, вызывается либо самим сервисом `Order`, либо первым этапом повествования. Как объяснялось в главе 4, состояние `PENDING` — это пример контрмеры под названием «семантическая блокировка», которая помогает изолировать повествования друг от друга. После обращения к сервисам-участникам повествование рано или поздно обновляет заказ, чтобы отразить результат выполнения. Например, как было описано в главе 4, у повествования есть несколько участников, включая сервисы `Consumer`, `Accounting` и `Kitchen`. Изначально `OrderService` создает заказ с состоянием `APPROVAL_PENDING` и позже меняет это состояние на `APPROVED` или `REJECTED`. Такое поведение агрегата `Order` можно смоделировать в виде конечного автомата (рис. 5.14).

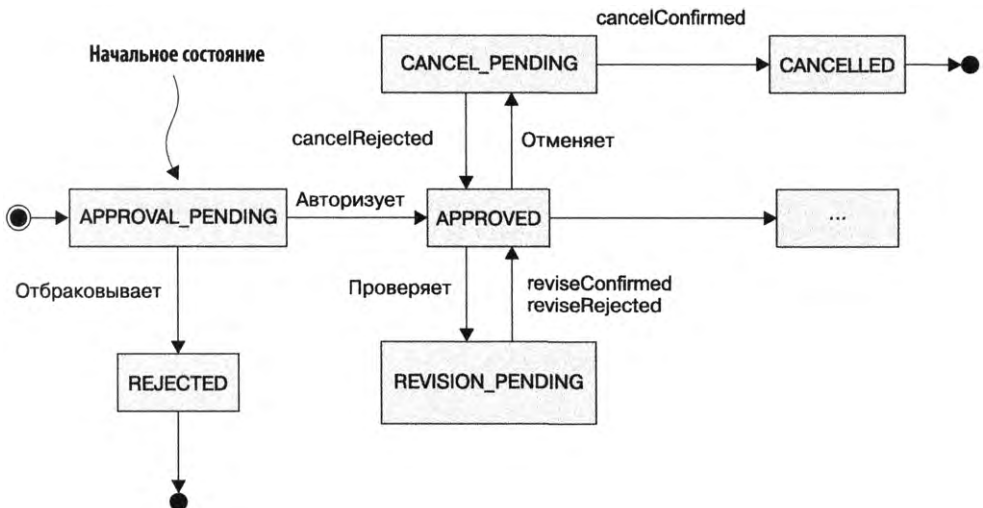


Рис. 5.14. Часть модели конечного автомата для агрегата `Order`

Точно так же другие операции сервиса `Order`, такие как `revise()` и `cancel()`, сначала меняют состояние заказа на `PENDING`, а затем используют повествование, чтобы проверить возможность выполнения операции. Затем, если проверка прошла

успешно, состояние заказа меняется на такое, которое отражает успешное выполнение операции. Если проверка оказалась неудачной, возвращается предыдущее состояние заказа. Например, операция `cancel()` вначале устанавливает заказ в состояние `CANCEL_PENDING`. Если заказ можно отменить, повествование `Cancel Order` меняет это состояние на `CANCELLED`. В противном случае, если операция `cancel()` отклоняется (скажем, уже слишком поздно для отмены заказа), заказ возвращается обратно к состоянию `APPROVED`.

Теперь посмотрим, как агрегат `Order` реализует этот конечный автомат.

Методы агрегата `Order`

У класса `Order` есть несколько групп методов, каждая из которых относится к определенному повествованию. Один из методов группы вызывается в самом начале повествования, остальные — в конце. Начнем с обсуждения бизнес-логики, которая создает заказ, после этого рассмотрим операцию его обновления. В листинге 5.15 показаны методы агрегата `Order`, вызываемые в процессе создания заказа.

Листинг 5.15. Методы, которые вызываются во время создания заказа

```
public class Order { ...

    public static ResultWithDomainEvents<Order, OrderDomainEvent>
        createOrder(long consumerId, Restaurant restaurant,
                    List<OrderLineItem> orderLineItems) {
        Order order = new Order(consumerId, restaurant.getId(), orderLineItems);
        List<OrderDomainEvent> events = singletonList(new OrderCreatedEvent(
            new OrderDetails(consumerId, restaurant.getId(), orderLineItems,
                order.getOrderTotal()),
            restaurant.getName()));
        return new ResultWithDomainEvents<>(order, events);
    }

    public Order(OrderDetails orderDetails) {
        this.orderLineItems = new OrderLineItems(orderDetails.getLineItems());
        this.orderMinimum = orderDetails.getOrderMinimum();
        this.state = APPROVAL_PENDING;
    }
    ...

    public List<DomainEvent> noteApproved() {
        switch (state) {
            case APPROVAL_PENDING:
                this.state = APPROVED;
                return singletonList(new OrderAuthorized());
            ...
            default:
                throw new UnsupportedOperationException(state);
        }
    }

    public List<DomainEvent> noteRejected() {
        switch (state) {
```

```

    case APPROVAL_PENDING:
        this.state = REJECTED;
        return singletonList(new OrderRejected());
        ...
    default:
        throw new UnsupportedOperationException(state);
}
}

```

`createOrder()` — это статический фабричный метод, который создает заказ и публикует событие `OrderCreatedEvent`. Последнее обогащается подробностями о заказе, включая его позиции, общую сумму, а также ID и название ресторана. В главе 7 мы поговорим о том, как сервис `Order History` поддерживает легкодоступную копию `Order`, используя такие события, как `OrderCreatedEvent`.

Начальное состояние заказа равно `APPROVAL_PENDING`. Когда повествование `Create Order` завершается, оно вызывает либо `noteApproved()`, либо `noteRejected()`. Метод `noteApproved()` вызывается в случае успешной авторизации банковской карты клиента. Метод `noteRejected()` вызывается, когда один из сервисов отклоняет заказ или авторизация оказывается неудачной. Как видите, поле `state` агрегата `Order` определяет поведение большинства его методов. Кроме того, агрегат `Order`, как и `Ticket`, генерирует события.

Класс `Order`, помимо `createOrder()`, определяет несколько методов обновления. Например, чтобы пересмотреть заказ, повествование `Revise Order` сначала вызывает метод `revise()`, а затем, подтвердив возможность выполнения этой операции, использует метод `confirmRevised()`. Эти методы показаны в листинге 5.16.

Листинг 5.16. Методы `Order` для пересмотра заказа

```

class Order ...

    public List<OrderDomainEvent> revise(OrderRevision orderRevision) {
        switch (state) {

            case APPROVED:
                LineItemQuantityChange change =
                    orderLineItems.lineItemQuantityChange(orderRevision);
                if (change.newOrderTotal.isGreaterThanOrEqualTo(orderMinimum)) {
                    throw new OrderMinimumNotMetException();
                }
                this.state = REVISION_PENDING;
                return singletonList(new OrderRevisionProposed(orderRevision,
                    change.currentOrderTotal, change.newOrderTotal));
            default:
                throw new UnsupportedOperationException(state);
        }
    }

    public List<OrderDomainEvent> confirmRevision(OrderRevision orderRevision) {
        switch (state) {
            case REVISION_PENDING:
                LineItemQuantityChange licd =

```

```

        orderLineItems.lineItemQuantityChange(orderRevision);

    orderRevision
        .getDeliveryInformation()
        .ifPresent(newDi -> this.deliveryInformation = newDi);

    if (!orderRevision.getRevisedLineItemQuantities().isEmpty()) {
        orderLineItems.updateLineItems(orderRevision);
    }

    this.state = APPROVED;
    return singletonList(new OrderRevised(orderRevision,
        licd.currentOrderTotal, licd.newOrderTotal));
    default:
        throw new UnsupportedOperationException(state);
    }
}
}
}

```

Метод `revise()` вызывается, чтобы инициировать пересмотр заказа. Помимо прочего, он проверяет, не нарушит ли это правило о минимальной сумме, и меняет состояние заказа на `REVISION_PENDING`. Успешно обновив сервисы `Kitchen` и `Accounting`, повествование `Revise Order` вызывает метод `confirmRevision()`, чтобы завершить пересмотр.

Эти методы вызываются классом `OrderService`. Рассмотрим его.

5.5.2. Класс `OrderService`

Класс `OrderService` определяет методы для создания и обновления заказов. Это главная точка входа в бизнес-логику, к которой обращаются входящие адаптеры, такие как `REST API`. Большинство методов этого класса формируют повествования для оркестрации создания и обновления агрегатов `Order`. В итоге этот сервис получается более сложным, чем класс `KitchenService`, который мы обсудили ранее. В листинге 5.17 показан фрагмент `OrderService`. В него внедряются разные зависимости, включая `OrderRepository`, `OrderDomainEventPublisher` и несколько диспетчеров повествований. Он определяет несколько методов, таких как `createOrder()` и `reviseOrder()`.

Листинг 5.17. Методы класса `OrderService` для создания заказов и управления ими

```

@Transactional
public class OrderService {

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private SagaManager<CreateOrderSagaState, CreateOrderSagaState>
        createOrderSagaManager;

    @Autowired
    private SagaManager<ReviseOrderSagaState, ReviseOrderSagaData>

```

```

    reviseOrderSagaManagement;

@Autowired
private OrderDomainEventPublisher orderAggregateEventPublisher;

public Order createOrder(OrderDetails orderDetails) {
    Restaurant restaurant = restaurantRepository.findById(restaurantId)
        .orElseThrow(() -
    > new RestaurantNotFoundException(restaurantId));

    List<OrderLineItem> orderLineItems =
        makeOrderLineItems(lineItems, restaurant);

    ResultWithDomainEvents<Order, OrderDomainEvent> orderAndEvents =
        Order.createOrder(consumerId, restaurant, orderLineItems);

    Order order = orderAndEvents.result;
    orderRepository.save(order);
    orderAggregateEventPublisher.publish(order, orderAndEvents.events);

    OrderDetails orderDetails =
        new OrderDetails(consumerId, restaurantId, orderLineItems,
            order.getOrderTotal());
    CreateOrderSagaState data = new CreateOrderSagaState(order.getId(),
        orderDetails);

    createOrderSagaManager.create(data, Order.class, order.getId());

    return order;
}

public Order reviseOrder(Long orderId, Long expectedVersion,
    OrderRevision orderRevision) {
    public Order reviseOrder(long orderId, OrderRevision orderRevision) {
        Order order = orderRepository.findById(orderId)
            .orElseThrow(() -> new OrderNotFoundException(orderId));
        ReviseOrderSagaData sagaData =
            new ReviseOrderSagaData(order.getConsumerId(), orderId,
                null, orderRevision);
        reviseOrderSagaManager.create(sagaData);
        return order;
    }
}

```

Создает агрегат Order

Сохраняет Order в базу данных

Публикует доменные события

Создает повествование Create Order

Извлекает Order

Создает повествование Revise Order

Метод `createOrder()` сначала создает и сохраняет агрегат `Order`, затем публикует сгенерированные им доменные события. А в конце создает `CreateOrderSaga`. Метод `reviseOrder()` извлекает `Order` и создает `ReviseOrderSaga`.

Бизнес-логика микросервисного и монолитного приложений не так уж сильно различается. Она состоит из классов, таких как сервисы, сущности на основе JPA

и репозитории. Но некоторые различия все же есть. Доменная модель микросервисов организована в виде набора агрегатов из DDD, которые накладывают различные архитектурные ограничения. В отличие от традиционной объектной модели, классы в разных агрегатах ссылаются друг на друга с помощью значений первичных ключей, а не объектных ссылок. Кроме того, транзакции могут создавать и обновлять только один агрегат. При изменении своего состояния агрегаты могут публиковать доменные события, что довольно полезно.

Еще одно важное различие состоит в том, что для обеспечения согласованности данных между несколькими сервисами часто используются повествования. Например, сервис `kitchen` всего лишь участвует в повествованиях, но никогда их не инициирует. Для сравнения: сервис `Order` полностью полагается на повествования при создании и обновлении заказов. Это вызвано тем, что заказы должны быть транзакционно согласованными с данными, принадлежащими другим сервисам. В итоге большинство методов класса `OrderService` не работают с заказами напрямую, а создают повествования.

Из этой главы вы узнали, как реализовать бизнес-логику с помощью традиционного подхода к сохранению данных. Сюда включена интеграция обмена сообщениями и публикации событий с управлением транзакциями базы данных. Код для публикации событий переплетается с бизнес-логикой. В следующей главе мы поговорим о шаблоне «Порождение событий», в котором генерация событий интегрирована в бизнес-логику, а не «прикручена сбоку».

Резюме

- ❑ Процедурный шаблон «Сценарий транзакции» часто является хорошим решением для реализации простой бизнес-логики. Но, когда бизнес-логика усложняется, стоит подумать об использовании объектно-ориентированной доменной модели.
- ❑ Хороший способ организации бизнес-логики сервиса — ее разделение на агрегаты по принципу DDD. Агрегаты делают доменную модель более модульной, исключают возможность применения объектных ссылок между сервисами и гарантируют, что каждая ACID-транзакция выполняется в рамках одного сервиса.
- ❑ При создании или обновлении агрегат должен публиковать доменные события. Эти события имеют множество сфер применения. В главе 4 вы могли видеть, как они реализуют повествования с использованием хореографии. А в главе 7 мы поговорим о том, как с их помощью можно обновлять реплицированные данные. Подписчики на доменные события могут уведомлять пользователей и другие приложения, а также публиковать сообщения в клиентском браузере через WebSocket.

Разработка бизнес-логики с порождением событий

В этой главе

- Использование шаблона «Порождение событий» в разработке бизнес-логики.
- Реализация хранилища событий.
- Интеграция повествований и бизнес-логики, основанной на порождении событий.
- Реализация оркестраторов повествований с помощью порождения событий.

Мэри понравилась идея, состоящая в структурировании бизнес-логики в виде набора агрегатов из DDD, которые публикуют доменные события (см. главу 5). Она могла представить себе, насколько полезным будет использование этих событий в микро-сервисной архитектуре. Мэри планировала применять события для реализации повествований на основе хореографии, которые обеспечивают согласованность данных между сервисами (см. главу 4). А также намеревалась задействовать представления CQRS — реплики с поддержкой эффективных запросов (см. главу 7).

Однако она была немного обеспокоена тем, что логика публикации событий может вызвать ошибки. С одной стороны, такая логика довольно проста: каждый метод агрегата, который инициализирует или меняет его состояние, возвращает список событий. Но с другой — она «прикручена сбоку» к бизнес-логике. Бизнес-логика продолжит работать, даже если разработчик забыл опубликовать событие. Мэри волновалась, что такой способ публикации событий может стать источником ошибок.

Много лет назад Мэри узнала о *порождении событий* (event sourcing) — событийном подходе к написанию бизнес-логики и сохранению доменных объектов.

В то время ее заинтриговали многочисленные преимущества этой методики, включая сохранение полной истории изменений агрегата. Но это было не более чем любопытство. Учитывая, насколько важную роль играют доменные события в микросервисной архитектуре, Мэри начала задумываться о возможности использования порождения событий в приложении FTGO. В конце концов, этот шаблон проектирования устраняет источник программных ошибок, гарантируя, что события всегда публикуются при создании и обновлении агрегата.

Я начну эту главу с описания принципа работы порождения событий и покажу, как применять его для написания бизнес-логики. Вы узнаете, как этот шаблон сохраняет все события каждого агрегата внутри так называемого *хранилища событий*. Мы взвесим преимущества и недостатки этого подхода и посмотрим, как реализовать вышеупомянутое хранилище. Вам будет представлен простой фреймворк для написания бизнес-логики на основе порождения событий. После я расскажу о том, почему этот шаблон — хорошая основа для реализации повествований. Начнем с рассмотрения разработки бизнес-логики с помощью порождения событий.

6.1. Разработка бизнес-логики с использованием порождения событий

Порождение событий — это еще один способ структурирования бизнес-логики и сохранения агрегатов. Агрегаты сохраняются в виде последовательности событий, каждое из которых представляет изменение состояния агрегата. Приложение воссоздает текущее состояние, воспроизводя записанные события.

Шаблон «Порождение событий»

Сохраняет агрегат в виде последовательности доменных событий, которые представляют изменения состояния. См. microservices.io/patterns/data/event-sourcing.html.

Порождение событий имеет несколько важных преимуществ. Например, оно сохраняет историю агрегатов, которая может пригодиться для аудита или соблюдения нормативно-правовых норм. Оно также выполняет надежную публикацию доменных событий, что особенно полезно в микросервисной архитектуре. Но у этого шаблона есть и недостатки. Он сложен в изучении, поскольку это совершенно другой способ написания бизнес-логики. К тому же обращение к хранилищу событий часто затруднительно и требует использования шаблона CQRS, который описывается в главе 7.

Я начну раздел с описания ограничений, свойственных традиционной модели сохранения данных. Затем мы подробно обсудим порождение событий и поговорим о том, как оно преодолевает эти ограничения. После этого я покажу, как реализовать агрегат `Order` с помощью порождения событий. В конце пройдемся по положительным и отрицательным аспектам этой методики.

Итак, рассмотрим ограничения традиционного подхода к сохранению данных.

6.1.1. Проблемы традиционного сохранения данных

Традиционный подход к сохранению информации подразумевает привязывание классов к таблицам баз данных, полей этих классов — к столбцам, а их экземпляров — к строкам этих таблиц. На рис. 6.1 показано, как агрегат `Order`, описанный в главе 5, накладывается на таблицу `ORDER`. Его поле `OrderLineItems` накладывается на таблицу `ORDER_LINE_ITEM`.

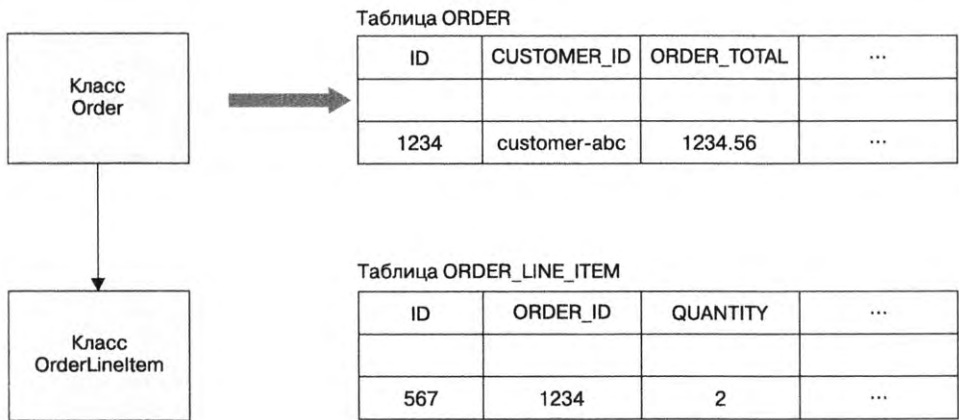


Рис. 6.1. Традиционный подход к сохранению информации подразумевает привязывание классов к таблицам, а объектов — к строкам в этих таблицах

Приложение хранит экземпляр заказа в виде строк в таблицах `ORDER` и `ORDER_LINE_ITEM`. Это можно сделать с помощью ORM-фреймворка, такого как JPA, или с использованием низкоуровневого пакета наподобие MyBatis.

Очевидно, что этот подход хорошо себя проявляет, так как большинство промышленных приложений хранят данные именно таким образом. Но у него есть несколько недостатков и ограничений.

- ❑ Объектно-реляционный разрыв.
- ❑ Отсутствие истории агрегатов.
- ❑ Реализация журналирования для аудита требует много усилий и чревата ошибками.
- ❑ Публикация событий является лишь дополнением к бизнес-логике.

Рассмотрим каждую из этих проблем, начиная с объектно-реляционного разрыва.

Объектно-реляционный разрыв

Есть одна старинная проблема под названием «*объектно-реляционный разрыв*». Она заключается в фундаментальном, концептуальном несоответствии между табличной реляционной схемой и графовой структурой развитой доменной модели с ее сложными отношениями. Некоторые аспекты этой проблемы отражены в непримиримых дискуссиях о целесообразности применения фреймворков для объектно-реляционного отображения (object/relational mapping, ORM). Например,

Тед Ньюард (Ted Neward) однажды сказал, что «объектно-реляционное отображение — это Вьетнамская война в мире информатики» (blogs.tedneward.com/post/the-vietnam-of-computer-science/). По правде сказать, я успешно использовал Hibernate для разработки приложений, в которых схема базы данных была построена на основе объектной модели. Но эта проблема более глубокая, чем избавление от какого-то конкретного ORM-фреймворка.

Нехватка истории агрегатов

Еще одно ограничение традиционного подхода к хранению данных состоит в том, что в нем предусмотрено хранение лишь текущего состояния агрегата. После обновления агрегата его предыдущее состояние теряется. Если приложению необходимо хранить историю агрегата (например, если это требуется правовыми нормами), разработчики сами должны реализовать этот механизм. Это занимает много времени и приводит к дублированию кода, который должен быть синхронизирован с бизнес-логикой.

Реализация журнала аудита — хлопотный процесс, чреватый ошибками

Еще одна проблема — журнал аудита. Многие приложения должны вести журнал, в который записывается информация о том, какие пользователи меняли агрегат. В некоторых случаях аудит требуется в целях безопасности или соблюдения нормативно-правовых норм. Но иногда история действий пользователя является важной функцией. Например, системы отслеживания проблем и управления задачами, такие как Asana и JIRA, выводят историю изменений задач и проблем. Трудность реализации аудита связана не только с необходимостью тратить на нее время и усилия, но и с тем, что бизнес-логика и код ведения журнала могут расходиться, что приводит к ошибкам.

Публикация событий не является частью бизнес-логики

Еще одно ограничение традиционного подхода к хранению данных состоит в том, что он обычно не поддерживает публикацию доменных событий. Как обсуждалось в главе 5, доменными называют события, которые публикуются при изменении состояния агрегата. В микросервисной архитектуре этот механизм помогает синхронизировать данные и отправлять уведомления. Некоторые ORM-фреймворки, такие как Hibernate, могут реагировать на изменение объектов данных с помощью функций обратного вызова, предоставленных приложением. Но поддержка автоматической публикации сообщений в рамках транзакции, которая обновляет данные, отсутствует. Таким образом, как мы уже видели на примере истории и аудита, разработчикам приходится дописывать логику генерации событий, которая потенциально может утратить синхронизацию с бизнес-логикой. К счастью, у этих проблем есть решение — порождение событий.

6.1.2. Обзор порождения событий

Порождение событий — это событийный подход к реализации бизнес-логики и сохранению агрегатов. Агрегат хранится в базе данных в виде цепочки событий. Каждое событие представляет изменение его состояния. Бизнес-логика агрегата структурирована вокруг требования о генерации и потреблении этих событий. Посмотрим, как это работает.

Сохранение агрегатов с помощью порождения событий

В подразделе 6.1.1 я показал, как традиционные механизмы хранения накладывают агрегаты на таблицы, их поля — на столбцы, а их экземпляры — на строки. Порождение событий основано на концепции доменных событий и работает совсем иначе. Оно сохраняет каждый агрегат в базе данных, так называемом хранилище событий, в виде последовательности событий.

Возьмем в качестве примера агрегат `Order`. Вместо сохранения каждого экземпляра `Order` в виде строки таблицы `ORDER` порождение событий помещает каждый агрегат `Order` в таблицу `EVENTS`, используя одну или несколько строк (рис. 6.2). Каждая строка — это доменное событие, такое как `Order Created`, `Order Approved`, `Order Shipped` и т. д.

event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
104	Order Shipped	Order	101	{...}
105	Order Delivered	Order	101	{...}
...

Таблица EVENTS

Рис. 6.2. Порождение событий сохраняет каждый агрегат в виде последовательности событий. Приложения на основе СУРБД могут, к примеру, хранить события в таблице `EVENTS`

Создавая или обновляя агрегат, приложение вставляет в таблицу `EVENTS` событие, которое тот сгенерировал. Приложение загружает агрегат из хранилища, извлекая

и воспроизводя события. Если говорить более подробно, загрузка агрегата состоит из следующих трех шагов.

1. Загрузка событий агрегата.
2. Создание экземпляра агрегата с помощью конструктора по умолчанию.
3. Перебор событий с вызовом `apply()`.

Например, в фреймворке Eventuate Client, который мы рассмотрим в подразделе 6.2.2, для реконструкции агрегата используется примерно такой код:

```
Class aggregateClass = ...;
Aggregate aggregate = aggregateClass.newInstance();
for (Event event : events) {
    aggregate = aggregate.applyEvent(event);
}
// использование агрегата...
```

Он создает экземпляр класса и проходит по событиям, вызывая метод агрегата `applyEvent()`. Если вы знакомы с функциональным программированием, то, наверное, догадались, что это операция *свертывания*.

Реконструкция состояния агрегата, хранимого в оперативной памяти, путем загрузки и воспроизведения событий может показаться странным и непривычным подходом. Но в некотором смысле это не сильно отличается от того, как ORM-фреймворки вроде JPA или Hibernate загружают свои сущности. В процессе загрузки объекта ORM-фреймворк выполняет одну или несколько инструкций `SELECT`, чтобы извлечь текущее сохраненное состояние, и создает экземпляр этого объекта, вызывая его конструктор по умолчанию. Для инициализации объектов применяется отражение. Отличительной чертой порождения событий является то, что состояние, хранимое в оперативной памяти, реконструируется с помощью событий.

Посмотрим, какие ограничения накладывает порождение событий на доменные события.

События представляют изменения состояния

Согласно определению, данному в главе 5, доменные события — это механизм уведомления подписчиков об изменениях, вносимых в агрегаты. События могут содержать минимальную информацию, такую как ID агрегата, или быть расширены с помощью данных, которые могут пригодиться типичному потребителю. Например, при создании заказа сервис `Order` может опубликовать событие `OrderCreated`. Содержимое `OrderCreated` может ограничиваться полем `orderId`. Или содержать весь заказ, чтобы его потребителю не нужно было извлекать эти данные из сервиса `Order`. Факт публикации события и его содержимое определяются тем, что именно нужно потребителю. Но в случае с порождением событий эта ответственность ложится на сам агрегат.

При использовании рассмотренного подхода события генерируются всегда. Каждое изменение состояния агрегата, включая его создание, представлено доменным событием. Каждый раз, когда агрегат меняет свое состояние, он обязан сгенерировать событие. Например, агрегат `Order` должен сгенерировать `OrderCreated` во время своего создания, а также события вида `Order*` при каждом своем обновлении.

Это требование куда более жесткое, чем то, что мы видели ранее, когда агрегат генерировал только те события, которые были интересны потребителям.

Но это еще не все. Событие должно содержать данные, необходимые агрегату для перехода к новому состоянию. Состояние агрегата включает в себя значения полей его объекта. Изменение состояния может заключаться в простом изменении поля объекта, такого как `Order.state`. Оно также может подразумевать добавление или удаление других объектов, таких как позиции заказа.

Представьте, что S — текущее состояние агрегата, а его новое состояние обозначено S' (рис. 6.3). Событие E , представляющее изменение состояния, должно содержать такие данные, чтобы после вызова `order.apply(E)` заказ перешел из состояния S в состояние S' . В следующем разделе вы увидите, что `apply()` — это метод, который изменяет состояние, представленное событием.

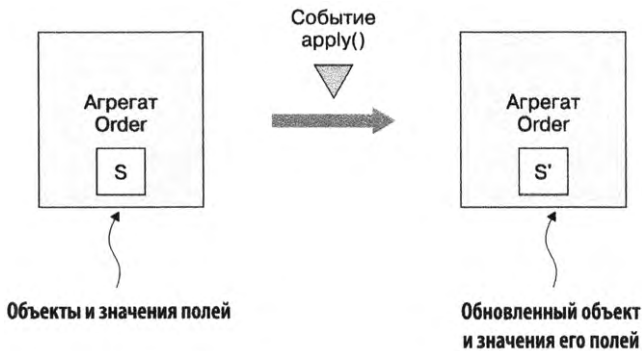


Рис. 6.3. Применение события E , когда заказ находится в состоянии S , должно изменить состояние на S' . Событие должно содержать данные, необходимые для выполнения перехода между состояниями

Некоторые события, такие как `Order Shipped`, содержат немного данных (или не содержат никаких) и просто описывают переход состояния. Метод `apply()` обрабатывает событие `Order Shipped`, меняя поле заказа `status` на `SHIPPED`. Но есть события, которые несут в себе много информации. Например, событие `OrderCreated` должно содержать все данные, необходимые методу `apply()` для инициализации заказа, включая его позиции, сведения о платеже, доставке и т. д. Поскольку событие `OrderCreated` используется для хранения агрегата, оно больше не может ограничиваться одним полем `orderId`.

Методы агрегата полностью полагаются на события

Чтобы обработать запрос и обновить агрегат, бизнес-логика вызывает командный метод из его корня. В традиционных приложениях командные методы обычно проверяют свои аргументы и затем обновляют одно или несколько полей агрегата. В приложениях, основанных на порождении событий, командные методы должны генерировать события. Результатом вызова командного метода агрегата является последовательность событий, описывающая изменения, которые нужно внести

в состояние (рис. 6.4). Эти события хранятся в базе данных и применяются к агрегату для его обновления.

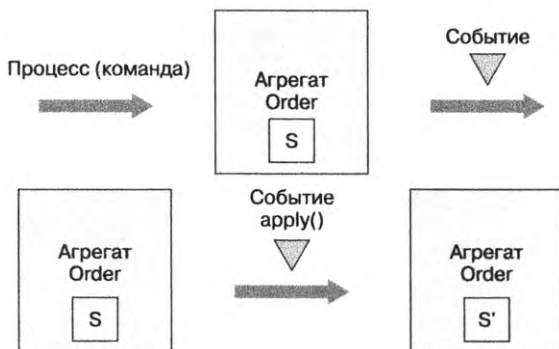


Рис. 6.4. Обработка команды генерирует события без изменения состояния агрегата. Агрегат обновляется путем применения событий

Обязательные генерация и применение событий требуют хоть и прямолинейной, но реструктуризации бизнес-логики. Порождение событий превращает командный метод в два и более метода. Первый из них принимает командный объект, который представляет запрос, и определяет, какое изменение состояния нужно выполнить. Он проверяет свои аргументы и, не меняя состояния агрегата, возвращает список событий, описывающих переход состояния. Если команду не удастся выполнить, этот метод обычно генерирует исключение.

Остальные методы принимают в качестве параметра событие определенного типа и обновляют агрегат. Для каждого события предусмотрен свой метод. Важно отметить, что ни один из этих методов не может отказать, поскольку событие представляет собой изменение состояния, которое *уже* произошло. В каждом случае агрегат обновляется на основе события.

В событийном фреймворке Eventuate Client, который мы подробнее рассмотрим в подразделе 6.2.2, эти методы называются `process()` и `apply()`. Первый принимает в качестве параметра командный объект, который содержит запрос на обновление, и возвращает список событий. Второй принимает событие и возвращает пустое значение. Агрегат определяет несколько перегруженных версий этих методов: по одной разновидности `process()` для каждого класса команд и по одному методу `apply()` для каждого типа событий, который генерируется агрегатом. Пример показан на рис. 6.5.

В этом примере метод `reviseOrder()` был заменен методами `process()` и `apply()`. `process()` принимает в качестве параметра команду `ReviseOrder`, ее класс определяется введением объекта-параметра (refactoring.com/catalog/introduceParameterObject.html) в метод `reviseOrder()`. Итогом работы метода `process()` является либо возвращение события `OrderRevisionProposed`, либо генерация исключения, если уже слишком поздно пересматривать заказ или предлагаемый пересмотр конфликтует с минимальной суммой заказа. Метод `apply()` для события `OrderRevisionProposed` меняет состояние заказа на `REVISION_PENDING`.

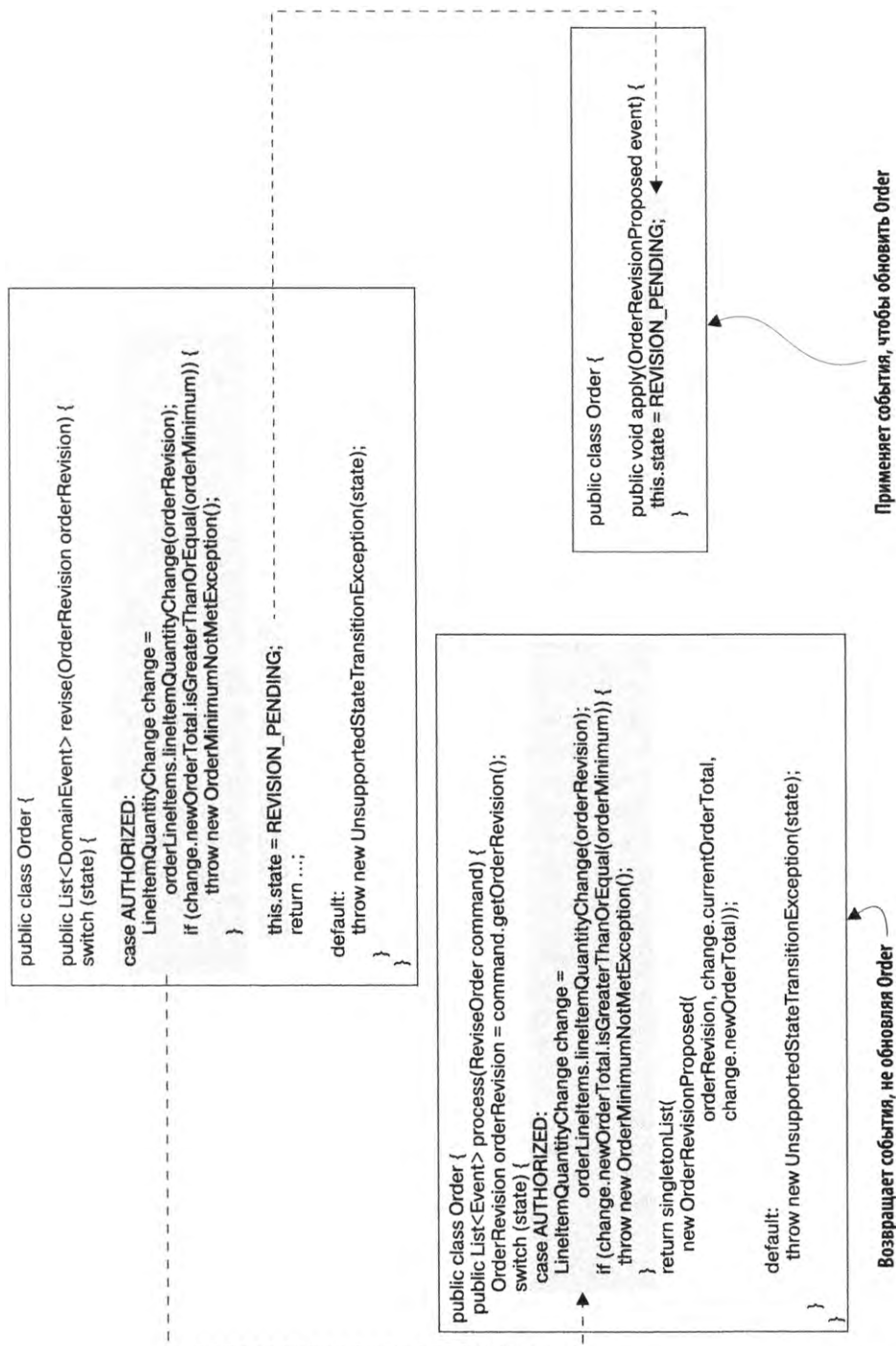


Рис. 6.5. Порождение событий разделяет метод для обновления агрегата на два других метода: `process()` принимает команду и возвращает события, `apply()` принимает событие и обновляет агрегат

Агрегат создается с помощью следующих шагов.

1. Создание экземпляра корня агрегата с помощью его конструктора по умолчанию.
2. Вызов `process()` для генерации новых событий.
3. Обновление агрегата путем перебора новых событий и вызова его метода `apply()`.
4. Сохранение новых событий в хранилище событий.

Обновление агрегата состоит из таких этапов.

1. Загрузка событий агрегата из хранилища событий.
2. Создание экземпляра корня агрегата с помощью его конструктора по умолчанию.
3. Перебор загруженных событий и вызов `apply()` из корня агрегата.
4. Вызов его метода `process()` для генерации новых событий.
5. Обновление агрегата путем перебора новых событий и вызова `apply()`.
6. Сохранение новых событий в хранилище событий.

Чтобы увидеть, как это работает, рассмотрим разновидность агрегата `Order`, основанную на порождении событий.

Агрегат `Order` на основе порождения событий

В листинге 6.1 показаны поля и методы агрегата `Order`, отвечающие за его создание. Версия этого агрегата, основанная на порождении событий, чем-то похожа на разновидность `Order` из главы 5 с использованием JPA. Их поля почти идентичны, и события, которые они генерируют, имеют много общего. Разница состоит в том, что в новой версии бизнес-логика сосредоточена на обработке команд, которые генерируют события, и применении этих событий для обновления своего состояния. Все методы, которые создают или обновляют агрегат на основе JPA, такие как `createOrder()` и `reviseOrder()`, заменены в новой версии на `process()` и `apply()`.

Листинг 6.1. Поля агрегата `Order` и его методы, которые инициализируют экземпляр

```
public class Order {

    private OrderState state;
    private Long consumerId;
    private Long restaurantId;
    private OrderLineItems orderLineItems;
    private DeliveryInformation deliveryInformation;
    private PaymentInformation paymentInformation;
    private Money orderMinimum;

    public Order() {
        // Проверяет команду
        // и возвращает OrderCreatedEvent
    }

    public List<Event> process(CreateOrderCommand command) {
        ... validate command ...
    }
}
```

```

return events(new OrderCreatedEvent(command.getOrderDetails()));
}

public void apply(OrderCreatedEvent event) {
    OrderDetails orderDetails = event.getOrderDetails();
    this.orderLineItems = new OrderLineItems(orderDetails.getLineItems());
    this.orderMinimum = orderDetails.getOrderMinimum();
    this.state = APPROVAL_PENDING;
}

```

← Применяет OrderCreatedEvent, инициализируя поля Order

Поля этого класса похожи на те, которые мы видели в агрегате Order, основанном на JPA. Единственное отличие в том, что идентификатор агрегата хранится за его пределами. В то же время методы выглядят совсем иначе. Вместо фабричного метода createOrder() мы видим две операции, process() и apply(). Метод process() принимает команду CreateOrder и генерирует событие OrderCreated. Метод apply() принимает событие OrderCreated и инициализирует поля класса Order.

Теперь рассмотрим чуть более сложную бизнес-логику для пересмотра заказа. Ранее этот код состоял из трех методов: reviseOrder(), confirmRevision() и rejectRevision(). В версии, основанной на порождении событий, они заменены тремя методами process() и несколькими методами apply(). То, как reviseOrder() и confirmRevision() выглядят после рефакторинга, показано в листинге 6.2.

Листинг 6.2. Методы process() и apply() для пересмотра агрегата Order

```

public class Order {

public List<Event> process(ReviseOrder command) {
    OrderRevision orderRevision = command.getOrderRevision();
    switch (state) {
        case APPROVED:
            LineItemQuantityChange change =
                orderLineItems.lineItemQuantityChange(orderRevision);
            if (change.newOrderTotal.isGreaterThanOrEqual(orderMinimum)) {
                throw new OrderMinimumNotMetException();
            }
            return singletonList(new OrderRevisionProposed(orderRevision,
                change.currentOrderTotal, change.newOrderTotal));
        default:
            throw new UnsupportedOperationException(state);
    }
}

public void apply(OrderRevisionProposed event) {
    this.state = REVISION_PENDING;
}

public List<Event> process(ConfirmReviseOrder command) {
    OrderRevision orderRevision = command.getOrderRevision();
    switch (state) {
        case REVISION_PENDING:
            LineItemQuantityChange licd =

```

← Проверим, можно ли пересмотреть заказ и достигает ли пересмотренная версия минимальной суммы

← Меняем состояние заказа на REVISION_PENDING

← Проверим, можно ли подтвердить изменения, и возвращаем событие OrderRevised


```

        orderLineItems.lineItemQuantityChange(orderRevision);
    return singletonList(new OrderRevised(orderRevision,
        licd.currentOrderTotal, licd.newOrderTotal));
    default:
        throw new UnsupportedOperationException(state);
    }
}

public void apply(OrderRevised event) {
    OrderRevision orderRevision = event.getOrderRevision();
    if (!orderRevision.getRevisedLineItemQuantities().isEmpty()) {
        orderLineItems.updateLineItems(orderRevision);
    }
    this.state = APPROVED;
}

```

← Пересматриваем заказ

Как видите, каждый метод был заменен операцией `process()` и одной или несколькими операциями `apply()`. Вместо `reviseOrder()` мы получили `process(ReviseOrder)` и `apply(OrderRevisionProposed)`. Аналогично метод `confirmRevision()` был заменен на `process(ConfirmReviseOrder)` и `apply(OrderRevised)`.

6.1.3. Обработка конкурентных обновлений с помощью оптимистичного блокирования

Ситуация, когда два запроса или больше одновременно обновляют один агрегат, не такая уж и редкость. Приложения, которые используют традиционную модель хранения данных, часто применяют оптимистичное блокирование, чтобы транзакции не перезаписывали изменения друг друга. Чтобы определить, изменился ли агрегат с тех пор, как он был прочитан, *оптимистичное блокирование* задействует столбец с версией. Приложение накладывает корень агрегата на таблицу со столбцом `VERSION`, который инкрементируется при каждом обновлении записи. Приложение обновляет агрегат с помощью выражения `UPDATE`, например:

```

UPDATE AGGREGATE_ROOT_TABLE
SET VERSION = VERSION + 1 ...
WHERE VERSION = <исходная версия>

```

Выражение `UPDATE` будет успешным, только если версия не изменилась с момента, когда приложение в последний раз считывало агрегат. Если агрегат считывается двумя транзакциями, успешно завершится только та, которая первой выполнит обновление. Вторая будет отменена, поскольку номер версии изменился, благодаря этому она не перезапишет изменения, внесенные первой транзакцией.

Хранилище событий тоже может использовать оптимистичное блокирование для обработки конкурентных обновлений. Каждый экземпляр агрегата содержит версию, которая считывается вместе с событиями. Когда приложение вставляет событие, хранилище проверяет, не изменилась ли его версия. В качестве номера версии можно взять количество событий. Но, как будет показано в разделе 6.2, хранилище событий может явно назначать номера версий.

6.1.4. Порождение и публикация событий

Строго говоря, шаблон «Порождение событий» сохраняет агрегат в виде событий, из которых затем восстанавливает его текущее состояние. Этот подход можно использовать также в качестве надежного механизма для публикации событий. Сохранение события в хранилище по своей сути атомарная операция. Нам нужно реализовать механизм для доставки всех сохраненных событий заинтересованным потребителям.

В главе 3 были описаны механизмы для публикации сообщений, которые вставляются в базу данных в рамках транзакции: опрашивание и отслеживание транзакционного журнала. Приложение, основанное на порождении событий, может публиковать события одним из этих способов. Основное отличие в том, что события хранятся в таблице **EVENTS** постоянно, а не временно, как в случае с таблицей **OUTBOX**, из которой они затем удаляются. Давайте рассмотрим эти подходы, начав с опрашивания.

Публикация событий с помощью опрашивания

Если для хранения событий используется таблица **EVENTS** (рис. 6.6), издатель может ее опрашивать на предмет новых записей, выполняя выражение **SELECT** и публикуя результат для брокера сообщений. Самое сложное — определить, какие из событий новые. Представьте, к примеру, что значения полей **eventId** увеличиваются монотонно. На первый взгляд было бы логично позволить издателю записывать последнее значение **eventId**, которое он обработал. После этого он смог бы извлечь новые события с помощью примерно такого запроса: **SELECT * FROM EVENTS where event_id > ? ORDER BY event_id ASC**.

Проблема этого подхода в том, что порядок фиксации транзакций может не совпасть с порядком, в котором они генерируют события. В итоге одно из событий может быть случайно пропущено издателем. Этот сценарий показан на рис. 6.6.

В этом сценарии транзакция А вставляет событие, у которого столбец **EVENT_ID** равен 1010. Затем транзакция В вставляет событие со столбцом **EVENT_ID**, равным 1020, и фиксируется. Теперь, если издатель выполнит запрос к таблице **EVENTS**, он найдет событие 1020. Позже, когда зафиксировается транзакция А, станет доступным событие 1010, но издатель его проигнорирует.

Одно из решений этой проблемы состоит в создании в таблице **EVENTS** дополнительного столбца, который отслеживает публикацию событий. В результате издатель использовал бы следующий процесс.

1. Найти неопубликованные события с помощью выражения **SELECT**: **SELECT * FROM EVENTS where PUBLISHED = 0 ORDER BY event_id ASC**.
2. Опубликовать события для брокера сообщений.
3. Пометить события такими, которые были опубликованы: **UPDATE EVENTS SET PUBLISHED = 1 WHERE EVENT_ID in**.

Благодаря этому подходу издатель не будет пропускать события.



Рис. 6.6. Сценарий, в котором событие пропускается из-за того, что его транзакция А фиксируется после транзакции В. Опрашивающий запрос видит eventId=1020 и позже пропускает eventId=1010

Надежная публикация событий с помощью отслеживания транзакционного журнала

Более развитые хранилища событий используют *отслеживание транзакционного журнала*, описанное в главе 3. Эта методика гарантирует публикацию событий и является более производительной и масштабируемой. Например, она применяется в открытом хранилище событий Eventuate Local — оно считывает события, вставленные в таблицу EVENTS из транзакционного журнала базы данных, и публикует их для брокера сообщений. Более подробно принцип работы Eventuate Local обсуждается в разделе 6.2.

6.1.5. Улучшение производительности с помощью снимков

Агрегат Order выполняет лишь несколько переходов между состояниями, поэтому генерирует небольшое количество событий. Обращение к хранилищу событий с последующей реконструкцией агрегата Order имеет высокую эффективность. Но у более долговечных агрегатов может быть намного больше событий. Примером такого агрегата служит Account. Со временем загрузка и сворачивание его событий могут существенно замедлиться.

Эту проблему часто решают периодическим сохранением снимков состояния агрегата. Пример использования снимка показан на рис. 6.7. Приложение восстанавливает состояние агрегата, загружая его последний снимок и только те события, которые произошли с момента его создания.

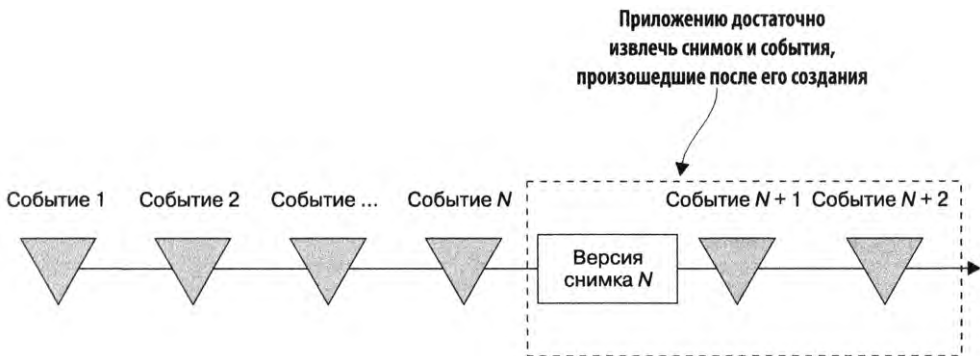


Рис. 6.7. Применение снимков улучшает производительность, так как вам больше не нужно загружать все события. Приложению достаточно извлечь снимок и события, произошедшие после его создания

В этом примере версия снимка имеет номер N . Чтобы восстановить состояние агрегата, приложению нужно загрузить только сам снимок и два события, которые за ним последовали. Предыдущие N событий не загружаются из хранилища.

При восстановлении агрегата из снимка приложение сначала использует снимок для создания экземпляра агрегата, а затем последовательно применяет события. Например, фреймворк Eventuate Client, описанный в разделе 6.2, восстанавливает агрегат с помощью примерно такого кода:

```
Class aggregateClass = ...;
Snapshot snapshot = ...;
Aggregate aggregate = recreateFromSnapshot(aggregateClass, snapshot);
for (Event event : events) {
    aggregate = aggregate.applyEvent(event);
}
// использование агрегата...
```

Экземпляр агрегата восстанавливается из снимка, а не с помощью конструктора по умолчанию. Если у агрегата простая, легко сериализуемая структура, в качестве снимка может использоваться представление в формате JSON. Снимки более сложных агрегатов создаются с помощью шаблона «Хранитель» ([ru.wikipedia.org/wiki/Хранитель_\(шаблон_проектирования\)](http://ru.wikipedia.org/wiki/Хранитель_(шаблон_проектирования))).

У агрегата Customer, показанного в примере онлайн-хранилища, очень простая структура: он содержит информацию о заказчике, его кредитном лимите и пр. Снимок представляет собой его состояние, переведенное в формат JSON. На рис. 6.8 показано, как воссоздать агрегат Customer из снимка, основанного на его состоянии на момент события 103. Сервису Customer нужно загрузить снимок и события, произошедшие после 103-го.

Сервис Customer воссоздает агрегат Customer путем десериализации снимка в формате JSON с последующей загрузкой и применением событий с 104-го по 106-е.

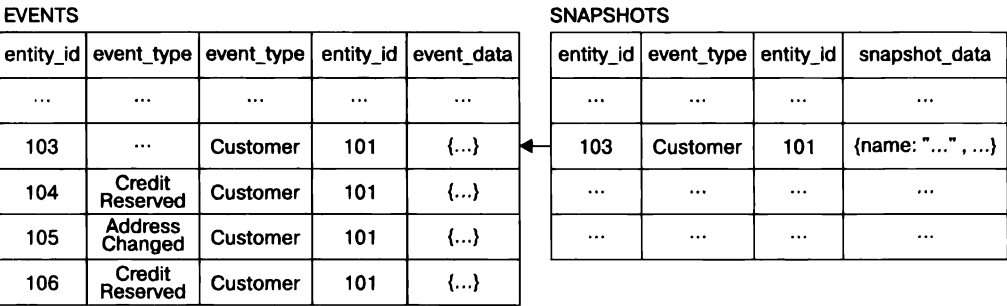


Рис. 6.8. Сервис Customer воссоздает агрегат Customer путем десериализации снимка в формате JSON с последующей загрузкой и применением событий с 104-го по 106-е

6.1.6. Идемпотентная обработка сообщений

Сервисы часто потребляют сообщения из других приложений или сервисов. Например, они могут потреблять доменные события, публикуемые агрегатом, или командные сообщения, отправляемые оркестратором повествования. Как описывалось в главе 3, при разработке потребителя сообщений важно убедиться в его идемпотентности, поскольку брокер может доставить одно и то же сообщение несколько раз.

Потребитель является идемпотентным, если его можно безопасно вызывать по несколько раз с одним и тем же сообщением. Например, фреймворк Eventuate Tram реализует идемпотентность, обнаруживая и отклоняя повторяющиеся сообщения. Он записывает *идентификаторы* обработанных сообщений в таблицу PROCESSED_MESSAGES в рамках локальной ACID-транзакции, с помощью которой бизнес-логика создает и обновляет агрегаты. Если ID сообщения уже находится в таблице PROCESSED_MESSAGES, это означает, что мы имеем дело с дубликатом, который можно отклонить. Бизнес-логика, основанная на порождении событий, должна реализовать аналогичный механизм. Как это сделать, зависит от того, реляционную СУРБД или NoSQL использует хранилище событий.

Идемпотентная обработка сообщений с хранилищем событий на основе СУРБД

Если приложение задействует хранилище событий на основе СУРБД, оно может применить идентичный подход к обнаружению и отклонению дубликатов. ID сообщения вставляется в таблицу PROCESSED_MESSAGES в рамках транзакции, которая добавляет события в таблицу EVENTS.

Идемпотентная обработка сообщений с хранилищем событий на основе NoSQL

Хранилище событий на основе NoSQL с ограниченной транзакционной моделью должно использовать иной механизм для реализации идемпотентной обработки сообщений. Потребитель должен каким-то образом автоматически сохранять собы-

тия и записывать ID сообщения. К счастью, для этого существует простое решение. На время обработки сообщения потребитель хранить его идентификатор внутри генерируемых событий. Для обнаружения дубликатов он следит за тем, чтобы ни одно из событий агрегата не содержало ID сообщения.

Одна из трудностей данного подхода связана с тем, что обработка сообщения может не сгенерировать никаких событий. Это означает следующее: у нас может не быть записи о том, что сообщение было обработано. Повторная доставка и обработка того же сообщения может привести к некорректному поведению. Рассмотрим, к примеру, такой сценарий.

1. Сообщение А было обработано, но не обновило агрегат.
2. Сообщение В было обработано, а его потребитель обновил агрегат.
3. Сообщение А доставляется повторно, но ввиду отсутствия записи о его обработке потребитель обновляет агрегат.
4. Сообщение В обрабатывается еще раз...

В этом сценарии повторная доставка событий приводит к разным и, возможно, некорректным результатам.

Чтобы этого избежать, можно сделать так, чтобы событие публиковалось всегда. Если агрегат ничего не генерирует, приложение сохраняет псевдособытие лишь для того, чтобы записать ID сообщения. Такие псевдособытия потребитель должен игнорировать.

6.1.7. Развитие доменных событий

Шаблон «Порождение событий», по крайней мере на концептуальном уровне, сохраняет события навсегда. Это палка о двух концах. С одной стороны, приложение получает журнал аудита с записями об изменениях, точность которых гарантируется. Это также позволяет приложению воссоздать любое предыдущее состояние агрегата. Но с другой — может возникнуть проблема, поскольку структура многих событий со временем меняется.

Существует вероятность того, что приложению придется иметь дело с несколькими версиями событий. Например, у сервиса, загружающего агрегат `Order`, может возникнуть необходимость в сохранении разных версий событий. Точно так же потребитель потенциально может видеть несколько версий.

Давайте сначала посмотрим, каким образом события могут меняться, а затем я опишу распространенный подход к обработке этих изменений.

Развитие структуры события

На концептуальном уровне приложение, основанное на порождении событий, имеет трехуровневую структуру.

- Состоит из одного или нескольких агрегатов.
- Определяет события, генерируемые каждым агрегатом.
- Определяет структуру событий.

В табл. 6.1 собраны разные типы изменений, которые могут возникнуть на каждом уровне.

Таблица 6.1. Разные способы развития событий приложения

Уровень	Изменение	Обратно совместимое
Структура	Определение нового типа агрегата	Да
Удаление агрегата	Удаление существующего агрегата	Нет
Переименование агрегата	Изменение названия типа агрегата	Нет
Агрегат	Добавление нового типа событий	Да
Удаление события	Удаление типа событий	Нет
Переименование события	Изменение названия типа событий	Нет
Событие	Добавление нового поля	Да
Удаление поля	Удаление поля	Нет
Переименование поля	Переименование поля	Нет
Изменение типа поля	Изменение типа поля	Нет

Эти изменения возникают естественным образом по мере развития доменной модели сервиса, например, когда меняются требования к сервису или его разработчики начинают лучше понимать проблемную область и улучшают доменную модель. На структурном уровне разработчики добавляют, удаляют и переименовывают классы агрегатов. На уровне отдельного агрегата могут измениться типы генерируемых событий. Разработчики могут изменить структуру типа события, добавив, удалив или изменив название либо тип поля.

К счастью, многие из этих изменений обратно совместимы. Например, добавление поля к событию вряд ли повлияет на потребителя — он просто проигнорирует неизвестное поле. Другие изменения не обладают обратной совместимостью. Например, изменение имени события или его поля требует обновления потребителя событий этого типа.

Управление структурными изменениями путем приведения к базовому типу

В мире SQL с изменениями структуры базы данных обычно справляются за счет миграции. Каждое структурное изменение представлено *миграцией* — SQL-скриптом, который меняет схему и переносит на нее имеющиеся данные. Миграции хранятся в системе управления версиями и применяются к базе данных с помощью таких инструментов, как Flyway.

Приложение, основанное на порождении событий, может использовать аналогичный подход для работы с обратно несовместимыми изменениями. Но вместо приведения событий к новой структуре соответствующий фреймворк преобразует их в момент загрузки из хранилища. Обновлением отдельных событий до новой версии

занимается компонент, который часто называют *upcaster*. В итоге код приложения всегда имеет дело с актуальной структурой событий.

Итак, мы обсудили принцип работы порождения событий. Теперь рассмотрим его преимущества и недостатки.

6.1.8. Преимущества порождения событий

Порождение событий имеет как положительные, так и отрицательные стороны. К положительным можно отнести:

- ❑ надежную публикацию доменных событий;
- ❑ сохранение истории изменений агрегата;
- ❑ отсутствие большинства проблем, связанных с объектно-реляционным разрывом;
- ❑ машину времени для разработчиков.

Рассмотрим каждое из этих преимуществ более подробно.

Надежная публикация доменных событий

Основное преимущество порождения событий — надежная публикация уведомлений о каждом изменении состояния агрегата. Это хорошая основа для событийной микросервисной архитектуры. К тому же каждое событие может хранить идентификатор пользователя, который внес изменение, что позволяет вести гарантированно корректный журнал аудита. Поток событий можно использовать для целого ряда других задач, включая уведомление пользователей, интеграцию приложений, аналитику и мониторинг.

Сохранение истории изменений агрегата

Еще одно преимущество порождения событий состоит в сохранении всей истории изменений каждого агрегата. Вы можете с легкостью реализовать временные запросы, которые извлекают агрегат в одном из его предыдущих состояний. Чтобы определить состояние агрегата в заданный момент времени, нужно свернуть события, которые произошли до этого момента. Например, вы можете легко рассчитать доступные кредитные средства клиента в какой-то момент в прошлом.

Отсутствие большинства проблем, связанных с объектно-реляционным разрывом

Порождение событий основывается скорее на их постоянном хранении, чем на агрегации. События обычно имеют простую структуру, которую легко сериализовать. Как уже упоминалось, сервис может сделать снимок сложного агрегата, сериализовав его текущее состояние. Это вводит новый уровень опосредованности между агрегатом и его сериализованным представлением.

Машина времени для разработчиков

Шаблон «Порождение событий» хранит историю всего, что происходило на протяжении жизненного цикла приложения. Представьте, что разработчикам FTGO нужно реализовать новое требование к заказчикам, которые добавляют товар в корзину покупок и затем удаляют его оттуда. Традиционное приложение не сохранило бы этой информации, поэтому таким пользователям можно было бы показывать рекламу только после реализации соответствующей возможности. Для сравнения: приложение на основе порождения событий может сразу же начать показывать рекламу пользователям, которые производили такие действия в прошлом. Разработчики получают своего рода машину времени, с помощью которой могут перемещаться в прошлое и реализовывать непредвиденные требования.

6.1.9. Недостатки порождения событий

Порождение событий не панацея. Вот его недостатки.

- ☐ Оно имеет другую модель программирования с высоким порогом вхождения.
- ☐ Оно так же сложно, как приложение, основанное на обмене сообщениями.
- ☐ Меняющиеся события могут создать проблемы.
- ☐ Усложняется удаление данных.
- ☐ Обращение к хранилищу событий связано с определенными трудностями.

Рассмотрим каждый из этих пунктов.

Другая модель программирования с высоким порогом вхождения

Эта модель программирования из-за своей необычности имеет высокий порог вхождения. Чтобы интегрировать порождение событий в существующее приложение, вы должны переписать бизнес-логику. К счастью, это довольно прямолинейное преобразование, которое можно выполнить во время миграции приложения на микросервисы.

Сложность приложения, основанного на обмене сообщениями

Еще одним недостатком порождения событий является то, что брокер сообщений обычно гарантирует доставку *не менее одного раза*. Если обработчики событий неидемпотентные, они должны обнаруживать и отклонять дубликаты. В этом способны помочь фреймворки, которые назначают каждому событию монотонно растущий идентификатор. Обработчик событий может обнаружить дубликат, отслеживая самое большое значение ID, которое ему встречалось. Это может происходить автоматически, когда обработчики событий обновляют агрегаты.

Меняющиеся события могут создать проблемы

При использовании порождения событий структура событий (и снимков!) будет меняться со временем. Поскольку события хранятся вечно, агрегатам, возможно, придется сворачивать их с учетом нескольких версий структуры. Существует реальный риск того, что агрегаты станут слишком раздутыми из-за кода, предназначенного для разных версий. Как упоминалось в подразделе 6.1.7, хорошим решением этой проблемы будет обновление событий до последней версии во время их загрузки из хранилища. Это позволяет хранить код обновления событий отдельно, что упрощает агрегаты, поскольку им нужно применять лишь самую последнюю версию событий.

Усложняется удаление данных

Одна из целей порождения событий заключается в сохранении истории агрегатов, поэтому данные намеренно хранятся вечно. При использовании этого шаблона традиционно применяется мягкое удаление. Приложение удаляет агрегат, устанавливая ему флаг *удален*, а тот, как правило, генерирует событие *Deleted*, уведомляющее всех заинтересованных потребителей. Любой код, который обращается к агрегату, может проверить нужный флаг и действовать соответственно.

Мягкое удаление подходит для многих видов данных. Но одна из трудностей заключается в соблюдении общего регламента защиты данных (General Data Protection Regulation, GDPR). Это европейская правовая норма по защите информации и конфиденциальности, которая дает человеку право стереть свое присутствие в Интернете (gdpr-info.eu/art-17-gdpr/). Приложение должно быть способно «забыть» о личной информации пользователя, такой как адрес его электронной почты. Проблема состоит в том, что в приложениях, основанных на порождении событий, электронный адрес может храниться в событии *AccountCreated* или использоваться в качестве первичного ключа для агрегата. Приложение должно каким-то образом забыть о пользователе, не удаляя события.

Один из механизмов решения этой проблемы — шифрование. Каждый пользователь имеет ключ шифрования, который хранится в отдельной таблице базы данных. Прежде чем попасть в хранилище, все события, содержащие личную информацию о пользователе, шифруются с помощью этого ключа. Когда пользователь запрашивает удаление всех своих данных, приложение удаляет из базы данных запись с ключом. Таким образом, личные данные пользователя, в сущности, удалены, так как события больше нельзя расшифровать.

Шифрование событий решает большинство проблем, связанных с удалением личных данных. Но если какой-то аспект личной информации пользователя задействован в идентификаторе агрегата, удаления одного лишь ключа может оказаться недостаточно. Например, в разделе 6.2 описывается хранилище событий с таблицей *entities*, первичным ключом которой является ID агрегата. Одно из решений — методика *псевдоанонимизации* — замена электронного адреса на токен *UUID* и использование его в качестве ID агрегата. Приложение хранит в базе данных связь между токеном *UUID* и электронным адресом. Когда пользователь запрашивает удаление своих данных, приложение удаляет из таблицы запись об электронной почте, в результате чего связь между ней и токеном *UUID* безвозвратно теряется.

Трудности при обращении к хранилищу событий

Представьте, что вам нужно найти клиентов, которые исчерпали свой кредитный лимит. Поскольку у вас нет столбца с кредитными средствами, вы не можете написать `SELECT * FROM CUSTOMER WHERE CREDIT_LIMIT = 0`. Вместо этого нужно использовать сложный и потенциально неэффективный запрос с вложенными выражениями `SELECT` — кредитный лимит вычисляется сворачиванием событий, установивших изначальные кредитные средства, с последующей их подгонкой. Что еще хуже, хранилища событий на основе NoSQL обычно поддерживают только запросы по первичному ключу. Следовательно, вам необходимо реализовать запросы с помощью методики CQRS, описанной в главе 7.

6.2. Реализация хранилища событий

Приложение, использующее порождение событий, хранит события в отдельном хранилище. *Хранилище событий* — это гибрид базы данных и брокера сообщений. Оно ведет себя как БД, потому что у него есть API для вставки и извлечения событий агрегата по первичному ключу. Но оно похоже и на брокер сообщений, потому что у него есть API, который позволяет подписываться на события.

Существует несколько разных способов реализации хранилища событий. Вы можете написать с нуля все, в том числе собственный фреймворк для порождения событий. Хранить записи можно, к примеру, в СУРБД. Простой, хотя и низкопроизводительный способ публикации событий состоит в том, что подписчики опрашивают таблицу `EVENTS` на предмет новых событий. Но, как упоминалось в подразделе 6.1.4, одна из трудностей этого подхода связана с необходимостью гарантировать, что подписчик будет обрабатывать все события в правильном порядке.

Еще один вариант заключается в использовании специализированного хранилища событий, которое обычно предоставляет богатый набор возможностей, а также улучшенные производительность и масштабируемость. Вы можете выбрать один из следующих проектов.

- ❑ *Event Store* — хранилище событий с открытым исходным кодом на основе .NET от Грегга Янга (Greg Young), пионера в области порождения событий (<https://eventstore.org/>).
- ❑ *Lagom* — микросервисный фреймворк от компании Lightbend, ранее известной как Typesafe (www.lightbend.com/lagom-framework).
- ❑ *Axon* — фреймворк с открытым исходным кодом на языке Java для разработки событийных приложений, которые используют порождение событий и CQRS (www.axonframework.org).
- ❑ *Eventuate* — фреймворк, разработанный моим стартапом Eventuate (eventuate.io). Он имеет две версии: Eventuate SaaS — облачный сервис и Eventuate Local — открытый проект на основе Apache Kafka/СУРБД.

Все эти фреймворки имеют свои особенности, но основная концепция у них общая. Поскольку лучше всего я знаком с проектом Eventuate, именно он и будет рассматри-

ваться в этой книге. Он имеет прямолинейную, простую для понимания архитектуру, которая иллюстрирует концепции порождения событий. Вы можете использовать его в своих приложениях, заново реализовать его принципы или выбрать один из аналогичных фреймворков, применив знания, которые здесь приобретете.

Я начну с описания работы хранилища событий Eventuate Local. Затем мы рассмотрим фреймворк Eventuate Client для Java, с помощью которого легко написать бизнес-логику, основанную на порождении событий, и хранилище Eventuate Local.

6.2.1. Принцип работы хранилища событий Eventuate Local

Eventuate Local — это хранилище событий с открытым исходным кодом. Его архитектура показана на рис. 6.9. События хранятся в базе данных, такой как MySQL. Приложение вставляет и извлекает события агрегатов по первичному ключу. Сами события отправляются брокером сообщений, таким как Apache Kafka. События из базы данных попадают к брокеру сообщений с помощью механизма отслеживания журнала транзакций.

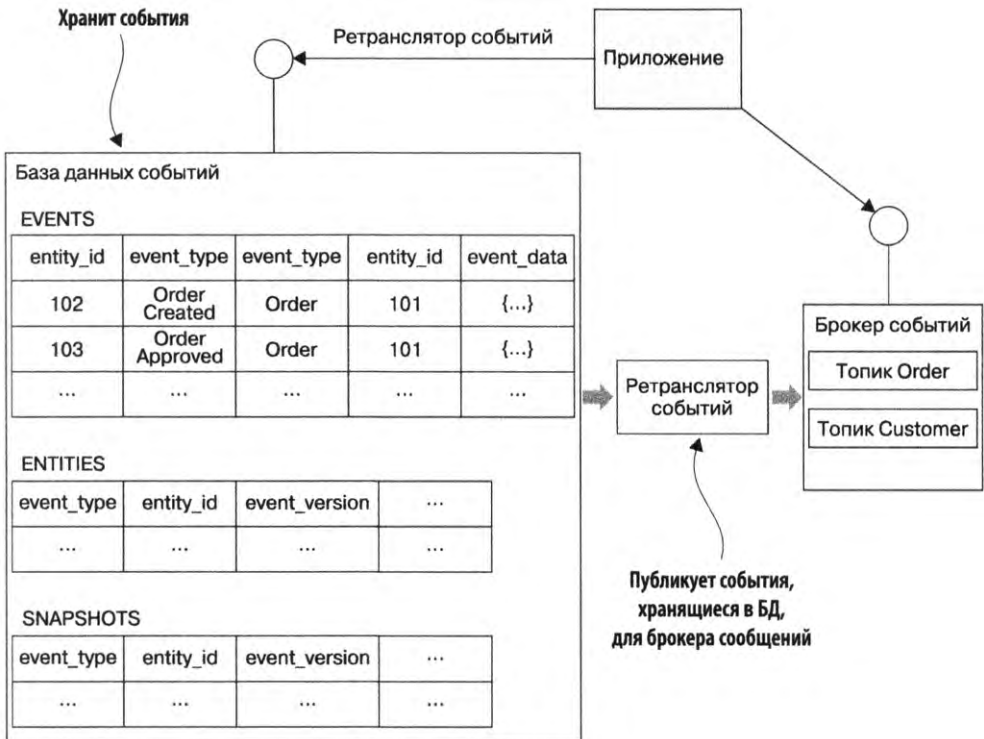


Рис. 6.9. Архитектура Eventuate Local. Она состоит из базы данных (такой как MySQL) для хранения событий, брокера (такого как Apache Kafka), доставляющего события к подписчикам, и ретранслятора, который публикует события, хранящиеся в БД, для брокера

Рассмотрим разные компоненты Eventuate Local, начиная со структуры базы данных.

Структура базы данных событий Eventuate Local

База данных событий состоит из трех таблиц.

- ❑ `events` — хранит события.
- ❑ `entities` — по одной строке для каждой сущности.
- ❑ `snapshots` — хранит снимки.

Главная таблица — `events`. Ее структура очень похожа на ту, что показана на рис. 6.2. Вот ее определение:

```
create table events (
  event_id varchar(1000) PRIMARY KEY,
  event_type varchar(1000),
  event_data varchar(1000) NOT NULL,
  entity_type VARCHAR(1000) NOT NULL,
  entity_id VARCHAR(1000) NOT NULL,
  triggering_event VARCHAR(1000)
);
```

Столбец `triggering_event` используется для обнаружения повторяющихся событий/сообщений. Он хранит ID сообщения/события, обработка которого сгенерировала это событие.

В таблице `entities` хранится текущая версия каждой сущности. Она применяется для реализации оптимистичного блокирования. Ее определение:

```
create table entities (
  entity_type VARCHAR(1000),
  entity_id VARCHAR(1000),
  entity_version VARCHAR(1000) NOT NULL,
  PRIMARY KEY(entity_type, entity_id)
);
```

Во время сохранения сущности в эту таблицу вставляется новая запись. При каждом изменении сущности обновляется столбец `entity_version`.

Таблица `snapshots` хранит снимки каждой сущности. Ее определение выглядит так:

```
create table snapshots (
  entity_type VARCHAR(1000),
  entity_id VARCHAR(1000),
  entity_version VARCHAR(1000),
  snapshot_type VARCHAR(1000) NOT NULL,
  snapshot_json VARCHAR(1000) NOT NULL,
  triggering_events VARCHAR(1000),
  PRIMARY KEY(entity_type, entity_id, entity_version)
)
```

Столбцы `entity_type` и `entity_id` определяют сущность снимка. В столбце `snapshot_json` находится сериализованное представление снимка, а в `snapshot_type` — его тип. В `entity_version` указывается версия сущности, с которой был сделан снимок.

Эта схема поддерживает три операции: `find()`, `create()` и `update()`. Операция `find()` обращается к таблице `snapshots`, чтобы извлечь последний снимок, если таковой имеется. При наличии снимка `find()` ищет в таблице `events` все события, у которых столбец `event_id` больше, чем `entity_version` снимка. В противном случае `find()` извлекает все события для заданной сущности. Эта операция также обращается к таблице `entity`, чтобы получить текущую версию сущности.

Операция `create()` вставляет запись в таблицу `entity` и события в таблицу `events`. Операция `update()` вставляет события в таблицу `events`. Она также проверяет оптимистичное блокирование, обновляя версию сущности в таблице `entities` с помощью выражения `UPDATE`:

```
UPDATE entities SET entity_version = ?  
WHERE entity_type = ? and entity_id = ? and entity_version = ?
```

Это выражение проверяет, не изменилась ли версия сущности с момента ее последнего извлечения операцией `find()`. Оно также обновляет `entity_version` до новой версии. Операция `update()` выполняет эти обновления в рамках транзакции, чтобы обеспечить атомарность.

Теперь вы знаете, как Eventuate Local хранит события и снимки агрегатов. А сейчас посмотрим, как клиент подписывается на эти события с помощью брокера из состава Eventuate Local.

Потребление событий с помощью подписки на брокер из состава Eventuate Local

Сервис потребляет события, подписываясь на брокер, реализованный с помощью Apache Kafka. У брокера событий есть тематика для каждого типа агрегата. Как объяснялось в главе 3, *тематика* — это секционированный канал сообщений. Благодаря этому потребитель может выполнять горизонтальное масштабирование, сохраняя порядок следования сообщений. ID агрегата используется в качестве ключа секционирования, который сохраняет порядок следования событий, публикуемых заданным агрегатом. Для потребления событий агрегата сервис подписывается на его тематику.

Теперь поговорим о ретрансляторе — прослойке, соединяющей базу данных и брокер событий.

Ретранслятор из состава Eventuate Local доставляет события из базы данных к брокеру сообщений

Ретранслятор доставляет события, вставленные в базу данных, к брокеру. Он использует отслеживание транзакционного журнала, если оно поддерживается, или опрашивание в случае с другими базами данных. Например, версия ретранслятора на основе MySQL применяет протокол репликации «ведущий/ведомый».

Она подключается к серверу MySQL в роли ведомой стороны и считывает его двоичный журнал, в который записываются обновления базы данных. Записи о событиях, вставленные в таблицу **EVENTS**, публикуются в подходящую тематику Apache Kafka. Ретранслятор событий игнорирует изменения любых других видов.

Ретранслятор событий разворачивается в виде автономного процесса. Для корректного перезапуска он периодически сохраняет текущую позицию двоичного журнала (имя файла и смещение) в специальной тематике Apache Kafka. В ходе запуска он извлекает оттуда последнюю записанную позицию и начинает чтение двоичного журнала MySQL из соответствующего места.

База данных, брокер сообщений и ретранслятор вместе формируют хранилище событий. Теперь рассмотрим фреймворк, с помощью которого Java-приложения обращаются к этому хранилищу.

6.2.2. Клиентский фреймворк Eventuate для Java

Фреймворк Eventuate Client (рис. 6.10) позволяет разработчикам писать приложения на основе порождения событий, которые используют хранилище Eventuate Local. Он предоставляет фундамент для разработки агрегатов, сервисов и обработчиков с поддержкой порождения событий.

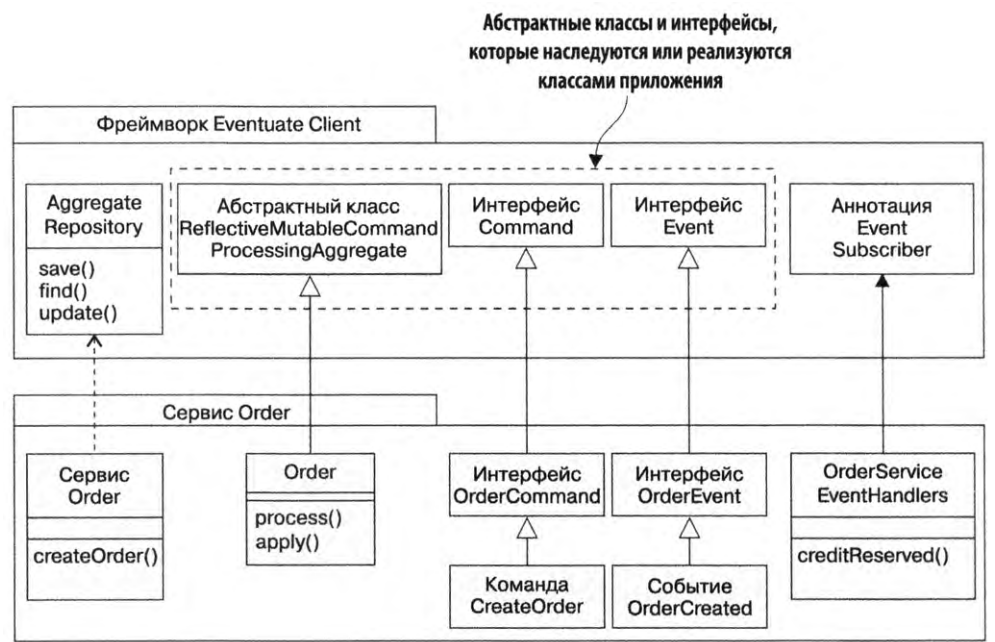


Рис. 6.10. Основные классы и интерфейсы, предоставляемые фреймворком Eventuate Client для языка Java

Этот фреймворк предоставляет базовые классы для агрегатов, команд и событий. Есть также класс `AggregateRepository`, который реализует функции CRUD. Кроме того, у него есть API для подписки на события.

Кратко рассмотрим каждый из типов, представленных на рис. 6.10.

Определение агрегатов с помощью класса `ReflectiveMutableCommandProcessingAggregate`

`ReflectiveMutableCommandProcessingAggregate` — это базовый обобщенный класс для агрегатов. У него есть два типа параметров: конкретный класс агрегата и родитель класса его команды. По довольно длинному имени можно догадаться, что он использует отражение для передачи команд и событий подходящим методам. Команды отправляются методу `process()`, а события — методу `apply()`.

`ReflectiveMutableCommandProcessingAggregate` наследуется классом `Order`, который вы видели ранее. Его код показан в листинге 6.3.

Листинг 6.3. Класс `Order` в версии фреймворка `Eventuate`

```
public class Order extends ReflectiveMutableCommandProcessingAggregate<Order,
    OrderCommand> {

    public List<Event> process(CreateOrderCommand command) { ... }

    public void apply(OrderCreatedEvent event) { ... }

    ...
}
```

Классы `ReflectiveMutableCommandProcessingAggregate` передаются два параметра: `Order` и `OrderCommand`. Последний является базовым интерфейсом для команд агрегата `Order`.

Определение команд агрегата

Классы команд агрегата должны реализовывать его базовый интерфейс, который, в свою очередь, должен наследовать интерфейс `Command`. Например, команды агрегата `Order` реализуют интерфейс `OrderCommand`:

```
public interface OrderCommand extends Command {
}

public class CreateOrderCommand implements OrderCommand { ... }
```

Интерфейс `OrderCommand` наследует `Command`, а класс `CreateOrderCommand` реализует `OrderCommand`.

Определение доменных событий

Классы событий агрегата должны реализовывать интерфейс-маркер `Event`, у которого нет методов. Также не помешает определить общий базовый интерфейс для всех классов событий агрегата — он будет наследовать `Event`. Например, далее показано определение класса `OrderCreated`:

```
interface OrderEvent extends Event {

}

public class OrderCreated extends OrderEvent { ... }
```

Класс событий `OrderCreated` наследует интерфейс `OrderEvent`, который является базовым для классов событий агрегата `Order`. `OrderEvent` наследует `Event`.

Создание, поиск и обновление агрегатов с помощью класса `AggregateRepository`

Фреймворк предоставляет несколько способов создания, поиска и обновления агрегатов. Самый простой способ, который здесь описывается, состоит в использовании обобщенного класса `AggregateRepository`, который параметризуется классом агрегата и базовым классом его команд. Он предоставляет три перегруженных метода:

- ❑ `save()` — создает агрегат;
- ❑ `find()` — ищет агрегат;
- ❑ `update()` — обновляет агрегат.

Особенно полезны методы `save()` и `update()`, так как они инкапсулируют в себе шаблонный код, необходимый для создания и обновления агрегатов. Например, `save()` принимает в качестве параметра командный объект и выполняет следующие действия.

1. Создает экземпляр агрегата с помощью его конструктора по умолчанию.
2. Вызывает метод `process()`, чтобы обработать команду.
3. Применяет сгенерированные события путем вызова `apply()`.
4. Сохраняет сгенерированные события в хранилище.

Метод `update()` работает похожим образом. Он принимает два параметра, ID агрегата и команду, и выполняет следующие действия.

1. Извлекает агрегат из хранилища событий.
2. Вызывает метод `process()`, чтобы обработать команду.
3. Применяет сгенерированные события путем вызова `apply()`.
4. Сохраняет сгенерированные события в хранилище.

Класс `AggregateRepository` в основном используют сервисы, которые создают и обновляют агрегаты в ответ на внешние запросы. Например, в листинге 6.4 показано, как `OrderService` создает `Order` с помощью `AggregateRepository`.

Листинг 6.4. `OrderService` использует `AggregateRepository`

```
public class OrderService {
    private AggregateRepository<Order, OrderCommand> orderRepository;

    public OrderService(AggregateRepository<Order, OrderCommand> orderRepository)
    {
        this.orderRepository = orderRepository;
    }

    public EntityWithIdAndVersion<Order> createOrder(OrderDetails orderDetails) {
        return orderRepository.save(new CreateOrder(orderDetails));
    }
}
```

`AggregateRepository` внедряется в сервис `OrderService` для работы с экземплярами `Order`. Метод сервиса `create()` делает вызов `AggregateRepository.save()`, передавая команду `CreateOrder`.

Подписка на доменные события

Фреймворк `Eventuate Client` также предоставляет API для написания обработчиков событий. В листинге 6.5 показан обработчик событий типа `CreditReserved`. Аннотация `@EventSubscriber` указывает идентификатор долговременной подписки. Если в момент публикации событий подписчик недоступен, он получит их при запуске. Аннотация `@EventHandlerMethod` помечает метод `creditReserved()` в качестве обработчика событий.

Листинг 6.5. Обработчик событий типа `OrderCreatedEvent`

```
@EventSubscriber(id="orderServiceEventHandlers")
public class OrderServiceEventHandlers {

    @EventHandlerMethod
    public void creditReserved(EventHandlerContext<CreditReserved> ctx) {
        CreditReserved event = ctx.getEvent();
        ...
    }
}
```

Обработчик событий принимает параметр типа `EventHandlerContext`, который содержит само событие и его метаданные.

Итак, вы узнали, как писать бизнес-логику на основе порождения событий с помощью фреймворка `Eventuate Client`. Теперь посмотрим, как эту логику можно применять совместно с повествованиями.

6.3. Совместное использование повествований и порождения событий

Представьте, что вы реализовали один или несколько сервисов на основе порождения событий. Они, вероятно, похожи на пример, показанный в листинге 6.4. Но если вы прочитали главу 4, то должны знать, что сервисам часто приходится инициировать *повествования* — последовательности локальных транзакций, которые обеспечивают согласованность данных между сервисами, и участвовать в них. Например, сервис `Order` применяет для проверки заказов повествование, в котором участвуют также сервисы `Kitchen`, `Consumer` и `Accounting`. Следовательно, вам нужно интегрировать повествования и бизнес-логику на основе порождения событий.

Порождение событий облегчает использование повествований, основанных на хореографии. Участники обмениваются доменными событиями, которые генерируются их агрегатами. Агрегаты каждого участника реагируют на события, обрабатывая команды и генерируя новые события. Вам необходимо написать классы агрегатов и обработчиков событий, которые будут обновлять агрегаты.

Однако интеграция бизнес-логики на основе порождения событий и повествований с поддержкой оркестрации может создать дополнительные трудности. Дело в том, что транзакции в понятии хранилища событий могут быть довольно ограниченными. Некоторые хранилища позволяют создавать или обновлять лишь один агрегат с последующей публикацией события (-ий). Но каждый этап повествования состоит из нескольких действий, которые необходимо выполнять атомарно.

- ❑ *Создание повествования.* Сервис, иницирующий повествование, должен создать его оркестратор и выполнить атомарное создание или обновление агрегата. Например, метод `createOrder()` из сервиса `Order` должен создать агрегат `Order` и повествование `CreateOrderSaga`.
- ❑ *Оркестрация повествования.* Оркестратор повествования должен атомарно потреблять ответы, обновлять его состояние и отправлять командные сообщения.
- ❑ *Участники повествования.* Участники повествования, такие как сервисы `Kitchen` и `Order`, должны атомарно потреблять сообщения, обнаруживать и отклонять дубликаты, создавать или обновлять агрегаты и отправлять ответные сообщения.

Учитывая несоответствие между требованиями к хранилищу событий и его транзакционными возможностями, интеграция повествований на основе оркестрации и порождения событий может создать некоторые интересные проблемы.

Ключевой фактор в интеграции этих двух технологий — выбор базы данных для хранилища событий: СУРБД или NoSQL. Фреймворк `Eventuate Tram`, описанный в главе 4, и фреймворк обмена сообщениями `Tram`, на котором он основан (см. главу 3), полагаются на гибкие ACID-транзакции, предоставляемые СУРБД. Оркестратор и участники повествования используют ACID-транзакции для атомарного обновления своих баз данных и обмена сообщениями. Если вы задействуете хранилище событий на основе СУРБД, такое как `Eventuate Local`, то можете *сжульничать*

и обновить его в рамках ACID-транзакции, обратившись к фреймворку Eventuate Tram. Но если хранилище событий основано на базе данных NoSQL, которая не может участвовать в одной транзакции с фреймворком Eventuate Tram, следует применить другой подход.

Подробно рассмотрим разные сценарии и проблемы, которые вам, возможно, придется решать.

- ❑ Реализация повествований на основе хореографии.
- ❑ Создание повествований на основе оркестрации.
- ❑ Реализация участника повествования, основанного на порождении событий.
- ❑ Реализация оркестраторов повествований с помощью порождения событий.

Начнем с обсуждения того, как реализовать повествование на основе хореографии, используя порождение событий.

6.3.1. Реализация повествований на основе хореографии с помощью порождения событий

Событийная природа шаблона «Порождение событий» делает реализацию повествований на основе хореографии довольно прямолинейной. При обновлении агрегат генерирует событие. У другого агрегата может быть обработчик, который обновляет его в результате получения этого события. Фреймворк для порождения событий автоматически делает все обработчики идемпотентными.

В главе 4 обсуждалось, как с помощью хореографии реализовать повествование `CreateOrder`. `ConsumerService`, `KitchenService` и `AccountingService` подписываются на события сервиса `OrderService`, и наоборот. У каждого сервиса есть обработчик событий, аналогичный показанному в листинге 6.5. Он обновляет соответствующий агрегат, который генерирует другое событие.

Порождение событий и повествования на основе хореографии отлично сочетаются друг с другом. Порождение событий предоставляет механизмы, которые нужны повествованиям, включая IPC на основе обмена сообщениями, дедупликацию сообщений, а также атомарное обновление состояния и отправку сообщений. Несмотря на свою простоту, повествования на основе хореографии имеют несколько недостатков. Некоторые из них перечислены в главе 4, но есть и такие, которые относятся непосредственно к порождению событий.

Проблема использования событий для хореографии повествований состоит в их двойном назначении. В шаблоне «Порождение событий» они описывают изменение состояния, но в хореографии повествований должны генерироваться агрегатом, даже если состояние не меняется. Например, если обновление агрегата нарушает бизнес-правило, тот должен сгенерировать событие, чтобы сообщить об ошибке. Еще более серьезная проблема может возникнуть, когда участнику повествования не удастся создать агрегат, в этом случае вернуть ошибку попросту некому.

Учитывая эти трудности, более сложные повествования лучше реализовывать с помощью оркестрации. В следующем разделе объясняется, как интегрировать

повествования на основе оркестрации и порождение событий. Как вы увидите, это потребует решения довольно интересных проблем.

Сначала посмотрим, как метод сервиса, такой как `OrderService.createOrder()`, создает оркестратор повествования.

6.3.2. Создание повествования на основе оркестрации

Оркестраторы повествований создаются некими методами сервиса. Другие методы, такие как `OrderService.createOrder()`, делают две вещи: создают или обновляют агрегат *и* создают оркестратор повествований. Сервис должен выполнять эти два действия так, чтобы второе гарантированно завершалось в случае успешного выполнения первого. То, как сервис этого добивается, зависит от того, какого рода хранилище событий он использует.

Создание оркестратора повествований с помощью хранилища событий на основе СУРБД

Если сервис использует хранилище событий на основе СУРБД, он может его обновить и создать оркестратор повествований в рамках одной ACID-транзакции. Представьте, к примеру, что сервис `OrderService` задействует хранилище `Eventuate Local` и фреймворк `Eventuate Tram`. В этом случае его метод `createOrder()` будет выглядеть так:

```
class OrderService

    @Autowired
    private SagaManager<CreateOrderSagaState> createOrderSagaManager;

    @Transactional
    public EntityWithIdAndVersion<Order> createOrder(OrderDetails orderDetails) {
        EntityWithIdAndVersion<Order> order =
            orderRepository.save(new CreateOrder(orderDetails));

        CreateOrderSagaState data =
            new CreateOrderSagaState(order.getId(), orderDetails);

        createOrderSagaManager.create(data, Order.class, order.getId());

        return order;
    }
    ...
```

Делаем так, чтобы метод `createOrder()` выполнялся в рамках транзакции БД

Создаем агрегат Order

Создаем CreateOrderSaga

Это комбинация листинга 6.4 и сервиса `OrderService`, описанного в главе 4. Поскольку хранилище `Eventuate Local` использует СУРБД, оно может участвовать в той же транзакции, что и фреймворк `Eventuate Tram`. Но если сервис задействует хранилище на основе `NoSQL`, создание оркестратора повествований оказывается не таким простым.

Создание оркестратора повествований при использовании хранилища событий на основе NoSQL

Сервис, применяющий хранилище событий на основе NoSQL, скорее всего, не сможет атомарно обновить его и создать оркестратор повествований. Фреймворк оркестрации может использовать совершенно другую базу данных. Но даже если это та же БД, из-за ограниченной транзакционной модели NoSQL приложение не сможет создать или обновить два разных объекта атомарно. Вместо этого у сервиса должен быть предусмотрен обработчик, который создает оркестратор повествований в ответ на доменное событие, сгенерированное агрегатом.

Например, на рис. 6.11 показано, как сервис **Order** создает **CreateOrderSaga** с помощью обработчика событий типа **OrderCreated**. Сначала он создает агрегат **Order** и сохраняет его в хранилище событий. Хранилище публикует событие **OrderCreated**, которое потребляется обработчиком. Затем обработчик обращается к фреймворку Eventuate Tram, чтобы создать **CreateOrderSaga**.

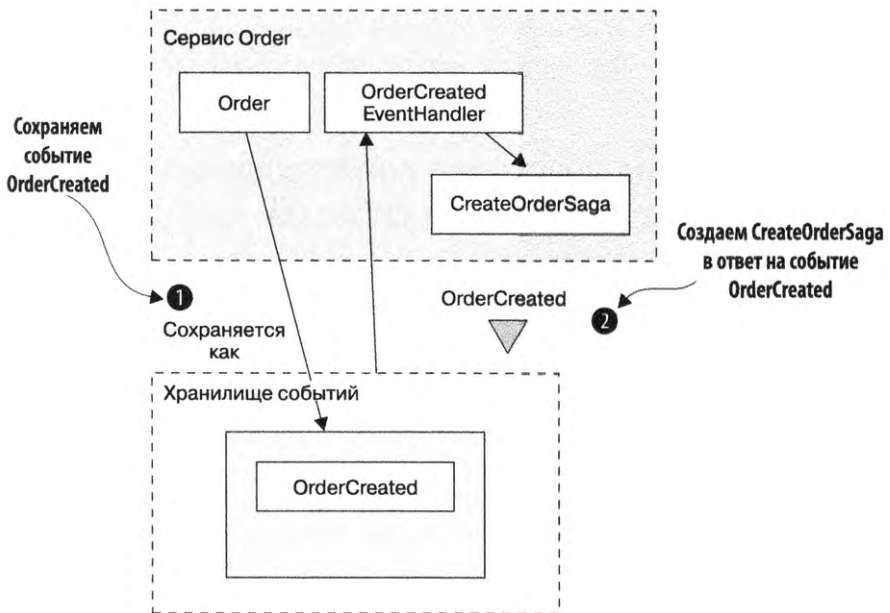


Рис. 6.11. Использование обработчика для надежного создания повествования после того, как сервис создаст агрегат на основе порождения событий

При написании обработчика, который создает оркестратор повествований, следует помнить, что он должен уметь справляться с повторяющимися событиями. Доставка сообщения не менее одного раза означает, что обработчик событий, создающий повествование, может быть вызван несколько раз. Важно сделать так, чтобы повествование создавалось в единственном экземпляре.

Одно из простых решений заключается в формировании идентификатора повествования на основе уникального атрибута события. Это можно сделать несколькими разными способами. Например, в качестве идентификатора повествования можно использовать ID агрегата, который генерирует событие. Это хорошо подходит для повествований, которые создаются в ответ на генерацию событий со стороны агрегата.

Вместо этого для идентификации повествования можно воспользоваться ID события. Поскольку эти ID уникальные, это гарантирует уникальность идентификатора повествования. Если событие окажется дубликатом, попытка обработчика создать повествование завершится неудачно, потому что этот ID уже существует. Рассмотренный вариант подходит в ситуациях, когда для заданного экземпляра агрегата могут существовать несколько копий одного и того же повествования.

Сервис, использующий хранилище событий на основе СУРБД, тоже может применять этот событийный подход к созданию повествований. Его преимущество в том, что он поощряет слабое связывание, поскольку сервисы наподобие `OrderService` больше не занимаются созданием экземпляров повествований напрямую.

Итак, вы научились создавать оркестраторы повествований. Теперь посмотрим, как сервисы, основанные на порождении событий, могут участвовать в оркестрируемых повествованиях.

6.3.3. Реализация участника повествования на основе порождения событий

Представьте, что вы используете порождение событий для реализации сервиса, которому нужно участвовать в повествовании на основе оркестрации. Неудивительно, что, если хранилище событий вашего сервиса (например, `Eventuate Local`) основано на СУРБД, вы гарантируете атомарную обработку командных сообщений и возвращение ответов. Сервис может обновлять хранилище в рамках ACID-транзакции, инициированной фреймворком `Eventuate Tram`. Но если вы выбрали для своего сервиса хранилище событий, которое не может участвовать в одной транзакции с фреймворком `Eventuate Tram`, то придется использовать совершенно другой подход.

Вы должны решить две разные проблемы, обеспечив:

- ❑ идемпотентность обработки командных сообщений;
- ❑ атомарную отправку ответного сообщения.

Сначала посмотрим, как реализовать идемпотентные обработчики командных сообщений.

Идемпотентная обработка командных сообщений

Первая проблема, которую нужно решить, — то, как участник повествования, основанного на порождении событий, будет обнаруживать и отклонять повторяющиеся сообщения. Это необходимо для реализации идемпотентной обработки командных

сообщений. К счастью, эту проблему легко решить с помощью механизма идиоматичной обработки сообщений, описанного ранее. Участник повествования записывает ID сообщения в события, которые генерируются при его обработке. Прежде чем обновлять агрегат, участник сверяет ID сообщения в событиях и убеждается в том, что он его еще не обрабатывал.

Атомарная отправка ответных сообщений

Вторая проблема связана с атомарной отправкой ответов участником повествования, основанного на порождении событий. В принципе, оркестратор может подписаться на события, генерируемые агрегатом, но у этого решения есть два недостатка. Во-первых, команда повествования может и не поменять состояние агрегата. В этом случае он не сгенерирует событие, поэтому оркестратору не будет отправлено никакого ответа. Во-вторых, этот подход требует, чтобы оркестратор различал участников в зависимости от того, используют они порождение событий или нет. Это связано с тем, что для получения доменных событий оркестратор должен подписаться не только на собственный канал ответов, но и на канал событий агрегата.

У этой проблемы есть более подходящее решение. Нужно сделать так, чтобы участник повествования продолжал отправлять ответные сообщения в канал ответов оркестратора. Но вместо того, чтобы выполнять это напрямую, он должен реализовать двухшаговый процесс.

1. Когда обработчик команд повествования создает или обновляет агрегат, он делает так, чтобы псевдособытие `SagaReplyRequested` сохранялось в хранилище вместе с настоящими событиями, сгенерированными агрегатом.
2. Обработчик псевдособытия `SagaReplyRequested` использует содержащиеся в нем данные для формирования ответного сообщения, которое затем записывает в канал ответов оркестратора повествований.

Рассмотрим пример, чтобы понять, как это работает.

Пример участника повествования на основе порождения событий

В этом примере мы сосредоточимся на сервисе `Accounting` — одном из участников повествования `Create Order`. На рис. 6.12 показано, как он обрабатывает команду `Authorize`, отправленную повествованием. Сервис `Accounting` реализован с помощью фреймворка `Eventuate Saga`, который имеет открытый исходный код и предназначен для написания повествований с использованием порождения событий. Он основан на фреймворке `Eventuate Client`.

На этом рисунке показано, как взаимодействуют повествование `Create Order` и сервис `Accounting`. Наблюдается такая последовательность событий.

1. Повествование `Create Order` шлет сервису `Accounting` команду `AuthorizeAccount` через канал сообщений. `SagaCommandDispatcher` из фреймворка `Eventuate Saga`

вызывает `AccountingServiceCommandHandler`, чтобы обработать командное сообщение.

2. `AccountingServiceCommandHandler` отправляет команду заданному агрегату `Account`.
3. Агрегат генерирует два события — `AccountAuthorized` и `SagaReplyRequestedEvent`.
4. `SagaReplyRequestedEventHandler` обрабатывает `SagaReplyRequestedEvent` и отправляет ответное сообщение повествованию `Create Order`.

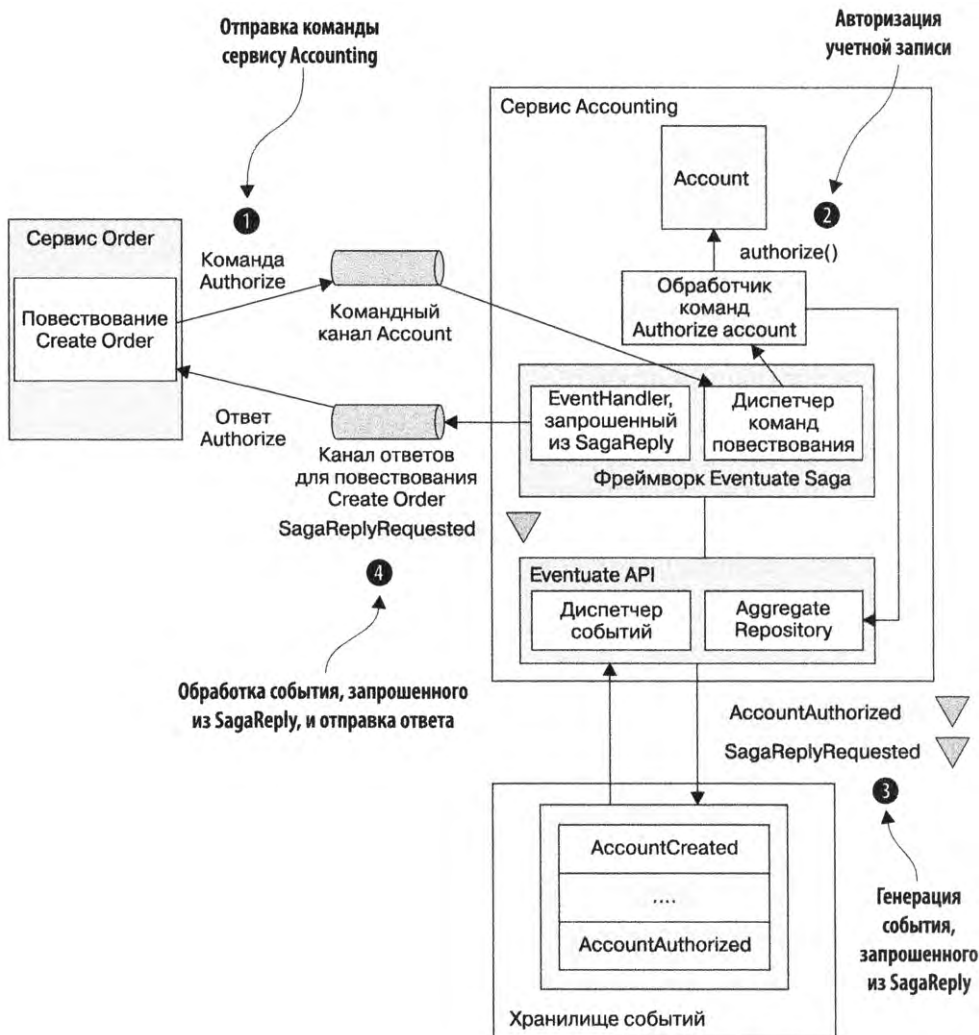


Рис. 6.12. Как сервис Accounting, основанный на порождении событий, участвует в повествовании Create Saga

Класс `AccountingServiceCommandHandler`, показанный в листинге 6.6, обрабатывает командное сообщение `AuthorizeAccount`. Для этого он вызывает метод `AggregateRepository.update()`, чтобы обновить агрегат `Account`.

Листинг 6.6. Обрабатывает командные сообщения, отправленные повествованиями

```
public class AccountingServiceCommandHandler {

    @Autowired
    private AggregateRepository<Account, AccountCommand> accountRepository;

    public void authorize(CommandMessage<AuthorizeCommand> cm) {
        AuthorizeCommand command = cm.getCommand();
        accountRepository.update(command.getOrderId(),
                                command,
                                replyingTo(cm)
                                    .catching(AccountDisabledException.class,
                                            () -> withFailure(new AccountDisabledReply()))
                                    .build());
    }

    ...
}
```

Метод `authorize()` использует `AggregateRepository`, чтобы обновить агрегат `Account`. Третий аргумент метода `update()` (`UpdateOptions`) вычисляется с помощью следующего выражения:

```
replyingTo(cm)
    .catching(AccountDisabledException.class,
              () -> withFailure(new AccountDisabledReply()))
    .build()
```

Параметры `UpdateOptions` конфигурируют метод `update()` для выполнения следующих действий.

1. *Идентификатор сообщения* используется в качестве ключа идемпотентности, чтобы сообщение всегда обрабатывалось только один раз. Как упоминалось ранее, фреймворк Eventuate хранит ключ идемпотентности во всех сгенерированных событиях, благодаря чему может обнаруживать и игнорировать любые повторные попытки обновления агрегата.
2. Псевдособытие `SagaReplyRequestedEvent` добавляется в список событий, помещенный в хранилище. Когда обработчик `SagaReplyRequestedEventHandler` получает псевдособытие `SagaReplyRequestedEvent`, он отправляет ответ в канал ответов `CreateOrderSaga`.
3. Когда агрегат генерирует исключение `AccountDisabledException`, вместо стандартной ошибки отправляется ответ `AccountDisabledReply`.

Итак, вы увидели, как реализовать участников повествования с помощью порождения событий. Теперь рассмотрим реализацию оркестраторов.

6.3.4. Реализация оркестраторов повествований с помощью порождения событий

До сих пор в этой главе мы обсуждали то, как сервисы, основанные на порождении событий, могут участвовать в повествованиях. Но порождение событий можно использовать и для реализации оркестраторов. Это позволит вам разрабатывать приложения, полностью основанные на хранилище событий.

При разработке оркестратора повествований необходимо решить три ключевые проблемы, связанные с проектированием.

1. Как будет храниться оркестратор?
2. Как атомарно изменять состояние оркестратора и слать командные сообщения?
3. Как убедиться в том, что оркестратор обрабатывает ответные сообщения ровно один раз?

В главе 4 мы обсуждали реализацию оркестратора на основе СУРБД. Теперь посмотрим, как решить эти проблемы с помощью порождения событий.

Сохранение оркестратора повествований с помощью порождения событий

У оркестратора повествований очень простой жизненный цикл. Сначала он создается, затем обновляется в ответ на сообщения, возвращаемые участниками повествования. Таким образом, мы можем сохранить оркестратор, используя следующие события:

- ❑ `SagaOrchestratorCreated` — оркестратор повествований создан;
- ❑ `SagaOrchestratorUpdated` — оркестратор повествований обновлен.

Оркестратор генерирует событие `SagaOrchestratorCreated`, когда его создают, и `SagaOrchestratorUpdated` — когда обновляют. Эти события содержат данные, необходимые для воссоздания состояния оркестратора. Например, события для `CreateOrderSaga`, описанные в главе 4, содержали бы сериализованную версию `CreateOrderSagaState` (например, в формате JSON).

Надежная отправка командных сообщений

Еще одной ключевой проблемой проектирования является обновление состояния повествования и отправка команд атомарным образом. Как описывалось в главе 4, при реализации повествований на основе Eventuate Tram обновление оркестратора и вставка командного сообщения в таблицу `message` происходят в рамках одной транзакции. Приложение, использующее хранилище событий на основе СУРБД, такое как Eventuate Local, может применять этот же подход. Но вы можете задействовать аналогичное решение, даже если хранилище основано на NoSQL (как, например, в Eventuate SaaS) и имеет ограниченную транзакционную модель.

Главное здесь — сохранить событие `SagaCommandEvent`, представляющее команду, которую нужно отправить. После этого обработчик событий подпишется на `SagaCommandEvent` и отправит каждое командное сообщение в подходящий канал. На рис. 6.13 показано, как это работает.

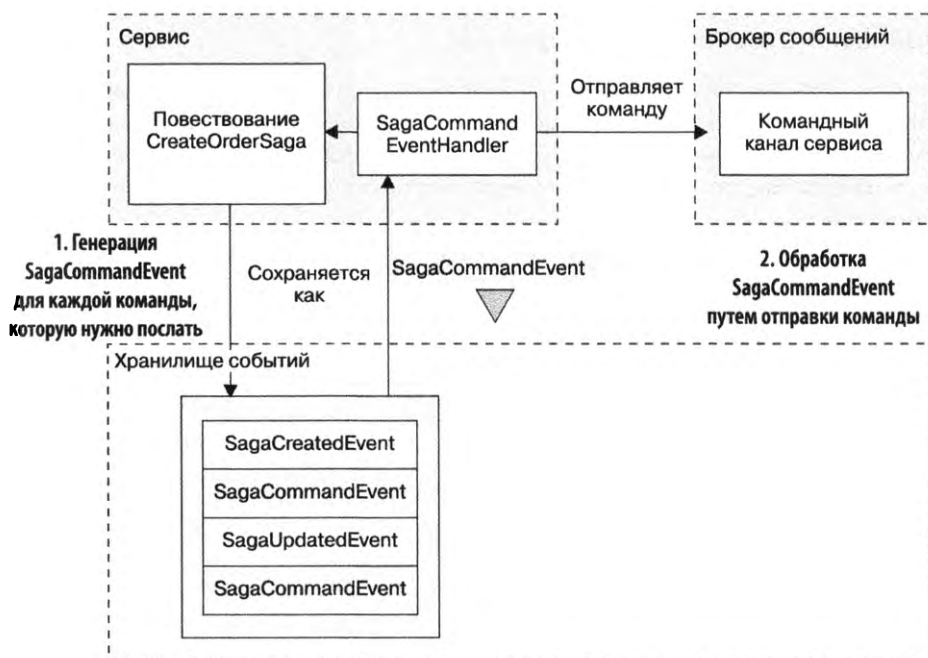


Рис. 6.13. Как оркестратор на основе порождения событий рассылает команды участникам повествования

Команды отправляются в два этапа.

1. Оркестратор генерирует событие `SagaCommandEvent` для каждой команды, которую он хочет отправить. `SagaCommandEvent` содержит все данные, необходимые для отправки команды, включая канал назначения и командный объект. Эти события находятся в хранилище.
2. Обработчик принимает события `SagaCommandEvent` и отправляет командные сообщения в заданный канал.

Двухшаговый процесс гарантирует, что команда будет отправлена хотя бы один раз.

Поскольку хранилище обеспечивает как минимум одноразовую доставку, обработчик может быть вызван несколько раз с одним и тем же событием. Если это случится, обработчик событий `SagaCommandEvent` пошлет несколько одинаковых командных сообщений. К счастью, участник повествования может с легкостью обнаружить и отклонить повторяющиеся команды с помощью следующего механизма.

Идентификатор `SagaCommandEvent`, уникальность которого гарантируется, используется в качестве ID командного сообщения. В итоге у дубликатов будет один и тот же ID. Участник повествования, который получает повторяющееся командное сообщение, отклонит его при помощи механизма, описанного ранее.

Обработка ответов ровно один раз

Оркестратор повествований должен обнаруживать и отклонять повторяющиеся ответные сообщения. Для этого он может использовать описанный ранее механизм. Оркестратор сохраняет ID ответного сообщения в события, которые он генерирует при обработке ответа. Затем он легко может определить, является ли сообщение дубликатом.

Как видите, порождение событий служит хорошей основой для реализации повествований. У этого шаблона есть и другие преимущества, включая изначально надежную генерацию событий при любом изменении данных, надежное ведение журнала аудита и выполнение временных запросов. Но нужно понимать, что это не панацея. Порождение событий имеет высокий порог вхождения. Изменение структуры событий не всегда проходит гладко. Но, несмотря на эти недостатки, порождение событий играет важную роль в микросервисной архитектуре. В следующей главе мы сменим тему и рассмотрим другой аспект, связанный с управлением распределенными данными в микросервисах, — запросы. Я покажу, как реализовать запросы, которые извлекают данные из разных сервисов.

Резюме

- ❑ Порождение событий сохраняет агрегат в виде последовательности событий. Каждое событие описывает либо создание агрегата, либо изменение его состояния. Для воссоздания состояния агрегата приложение воспроизводит события. Этот шаблон сохраняет историю доменного объекта, предоставляет точный журнал аудита и делает возможной надежную публикацию доменных событий.
- ❑ Снимки улучшают производительность, уменьшая количество событий, которые нужно воспроизводить.
- ❑ События размещаются в хранилище — гибриде базы данных и брокера сообщений. Когда сервис помещает событие в хранилище, он тем самым доставляет его подписчикам.
- ❑ Eventuate Local — это хранилище событий с открытым исходным кодом, основанное на MySQL и Apache Kafka. Разработчики используют фреймворк Eventuate Client для написания агрегатов и обработчиков событий.
- ❑ Одна из проблем порождения событий связана с их развитием. При воспроизведении событий приложение может обрабатывать разные их версии. Хорошим решением является приведение к базовому типу, когда события обновляются до последней версии во время загрузки из хранилища.

- ❑ Удаление данных в приложении на основе порождения событий связано с определенными трудностями. Существуют нормативно-правовые требования, например Общий регламент по защите данных в Европейском союзе. Для их соблюдения вы должны использовать такие методики, как шифрование и псевдоанонимизация, чтобы иметь возможность удалять данные пользователей.
- ❑ Порождение событий упрощает реализацию повествований, основанных на хореографии. У сервисов есть обработчики, которые отслеживают события, публикуемые агрегатами.
- ❑ Порождение событий — хороший подход к реализации оркестраторов повествований. Благодаря ему вы можете писать приложения, использующие исключительно хранилище событий.

Реализация запросов в микросервисной архитектуре

В этой главе

- Трудности запрашивания данных в микросервисной архитектуре.
- Когда и как реализовывать запросы с помощью объединения API.
- Когда и как реализовывать запросы с помощью шаблона CQRS.

Мэри и ее команда начали привыкать к идее использования повествований для обеспечения согласованности данных. Но затем они обнаружили, что управление транзакциями было не единственной проблемой, связанной с распределенными данными, которую нужно решить при переводе приложения FTGO на микросервисы. Помимо прочего, им следовало разобраться с тем, как реализовывать запросы.

Для поддержки пользовательского интерфейса приложение FTGO реализует целый ряд запросов. Их реализация в существующей монолитной архитектуре была довольно прямолинейной, потому что они использовали единую базу данных. Разработчикам FTGO в основном было достаточно написать SQL-выражения вроде `SELECT` и определить необходимые индексы. Но, как обнаружила Мэри, написание запросов в микросервисных системах сопряжено с определенными трудностями. Запросы часто должны извлекать данные, разбросанные по базам данных, принадлежащим разным сервисам. При этом нельзя задействовать традиционный механизм распределенных запросов — даже если бы это было технически возможно, это нарушало бы инкапсуляцию.

Возьмем, к примеру, запросы для приложения FTGO, описанные в главе 2. Некоторые из них извлекают данные, принадлежащие только одному сервису.

Например, запрос `findConsumerProfile()` возвращает данные из сервиса `Consumer`. Но такие операции, как `findOrder()` и `findOrderHistory()`, возвращают информацию, которой владеют несколько сервисов. Их реализация будет не такой простой.

Существует два шаблона для написания запросов в микросервисной архитектуре.

- ❑ *Объединение API*. Это самый простой подход, который следует использовать везде, где возможно. Он заключается в том, что за обращение к сервисам и объединение результатов отвечают клиенты.
- ❑ *Разделение ответственности командных запросов*. У этого шаблона, известного как CQRS, больше возможностей по сравнению с предыдущим, но при этом он сложнее. Он требует наличия одной или нескольких баз данных, единственное назначение которых заключается в поддержке запросов.

После обсуждения этих шаблонов поговорим о проектировании представлений CQRS, один из примеров которых будет реализован чуть позже. Начнем с объединения API.

7.1. Выполнение запросов с помощью объединения API

Приложение FTGO реализует целый ряд запросов. Некоторые из них, как упоминалось ранее, извлекают информацию из одного сервиса. Их реализация обычно не вызывает проблем, хотя позже в этой главе при рассмотрении шаблона CQRS вы увидите пример того, как обращение к одному сервису создает определенные трудности.

Существуют также запросы, которые извлекают данные из нескольких сервисов. В этом разделе я опишу операцию `findOrder()` — пример такого запроса. Я объясню, какие проблемы часто возникают при реализации подобных операций в микросервисной архитектуре. Затем опишу шаблон объединения API и покажу, как с его помощью можно реализовать запросы наподобие `findOrder()`.

7.1.1. Запрос `findOrder()`

Операция `findOrder()` извлекает заказ по его первичному ключу. Она принимает `orderId` в качестве параметра и возвращает объект `OrderDetails`, который содержит информацию о заказе. Как показано на рис. 7.1, эта операция вызывается клиентским модулем, который реализует представление *Order Status*, — например, мобильным устройством или веб-приложением.

Сведения, которые выводит *Order Status*, включают в себя основную информацию о состоянии заказа, платежа, выполнении заявки с точки зрения ресторана, а также местоположении и предполагаемом времени доставки, если заказ уже в пути.

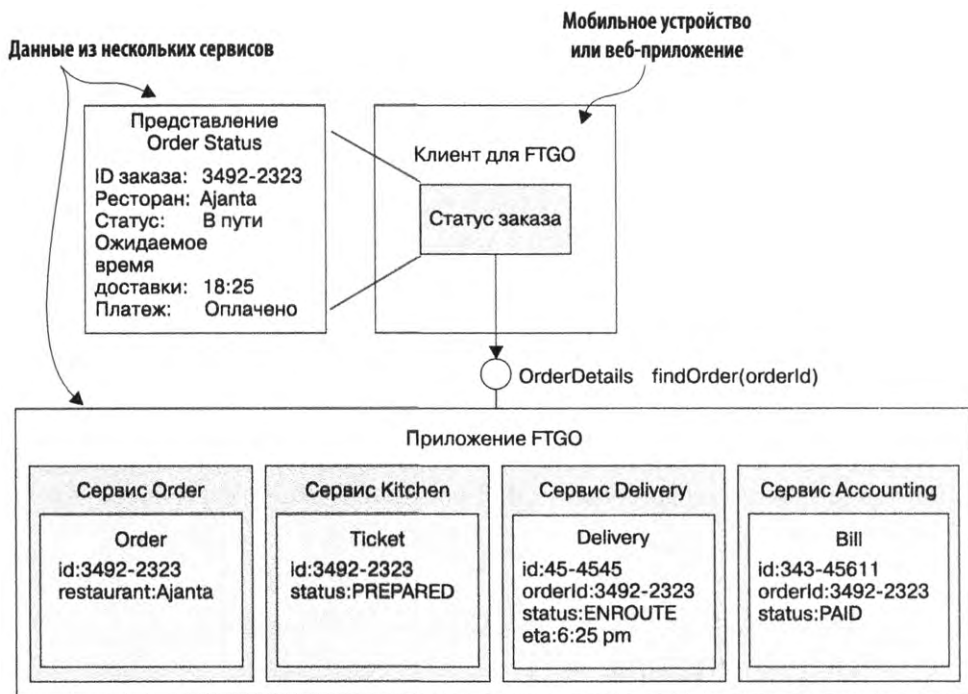


Рис. 7.1. Операция findOrder() вызывается клиентским модулем FTGO и возвращает подробности о заказе

Поскольку монолитная версия FTGO хранит свои данные в одной БД, она может легко извлечь подробности о заказе с помощью единственного выражения **SELECT**, которое объединяет различные таблицы. Тогда как в микросервисном приложении FTGO информация разбросана по следующим сервисам:

- ❑ **Order** — основная информация, включая подробности и статус;
- ❑ **Kitchen** — статус заказа с точки зрения ресторана и то, когда он должен быть готов к доставке;
- ❑ **Delivery** — состояние доставки заказа, предполагаемая информация о доставке и местоположение курьера;
- ❑ **Accounting** — состояние оплаты заказа.

Любой клиент, которому нужны подробности о заказе, должен опросить все эти сервисы.

7.1.2. Обзор шаблона «Объединение API»

Для реализации таких запросов, как findOrder(), которые извлекают данные, принадлежащие разным сервисам, можно использовать шаблон «Объединение API». Чтобы выполнить запрос, он обращается к сервисам, которым принадлежат нужные

данные, и объединяет результаты. Структура этого шаблона показана на рис. 7.2. Он предусматривает два вида участников:

- ❑ *API-композитор* — реализует операцию запроса, обращаясь к сервисам-провайдерам;
- ❑ *сервис-провайдер* — сервис, которому принадлежат данные, возвращаемые запросом.

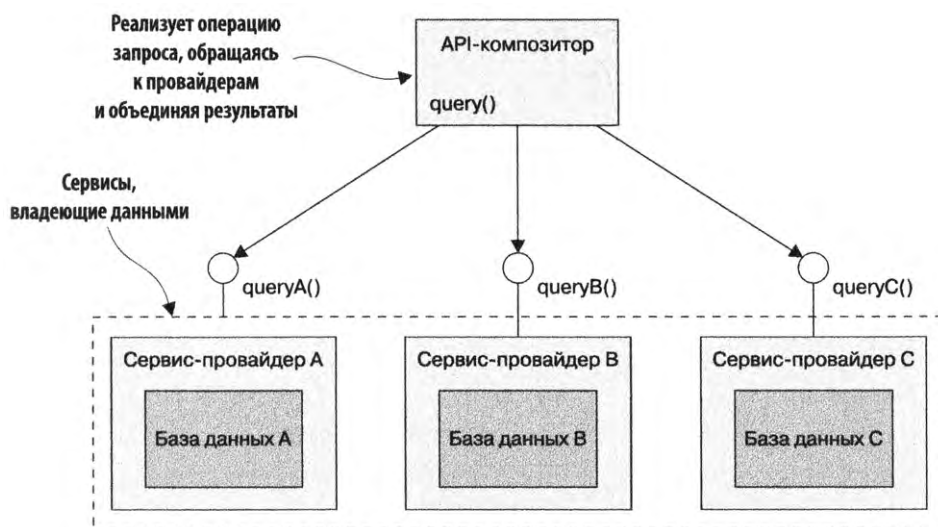


Рис. 7.2. Шаблон «Объединение API» состоит из API-композитора и двух или более сервисов-провайдеров. API-композитор реализует запрос, обращаясь к провайдерам и объединяя результаты

На рис. 7.2 показаны три сервиса-провайдера. API-композитор реализует запрос, извлекая данные из провайдеров и объединяя результаты. Он может быть как клиентом, таким как веб-приложение, которому нужны данные для отрисовки страницы, так и сервисом — например, API-шлюзом с серверами. Последний вариант описан в главе 8 на примере операции запроса, которая делается доступной в виде конечной точки API.

Шаблон «Объединение API»

Реализует запрос, который извлекает данные из разных сервисов, обращаясь к ним через их API и объединяя результаты. См. microservices.io/patterns/data/api-composition.html.

Подходит ли этот шаблон для реализации конкретного запроса, зависит от нескольких факторов, включая способ сегментирования информации, возможности API, которые предоставляют владельцы данных, а также возможности БД,

используемых сервисами. Например, даже если у сервиса-провайдера предусмотрен API для извлечения нужной информации, агрегатору, возможно, придется выполнить неэффективное соединение крупных наборов данных в памяти. Позже вы познакомитесь с примерами запросов, которые нельзя реализовать с помощью этого шаблона. К счастью, существует множество ситуаций, к которым его можно применить. Чтобы увидеть его в действии, рассмотрим пример.

7.1.3. Реализация запроса findOrder() путем объединения API

Операция findOrder() соответствует простому запросу по первичному ключу с объединением на основе равенства (equijoin). Логично предположить, что в API каждого из сервисов-провайдеров есть конечная точка для извлечения нужных данных по orderId. Следовательно, запрос findOrder() — отличный кандидат для того, чтобы быть реализованным с помощью шаблона «Объединение API». API-композитор обращается к четырем сервисам и объединяет полученные результаты. Структура композитора Find Order показана на рис. 7.3.

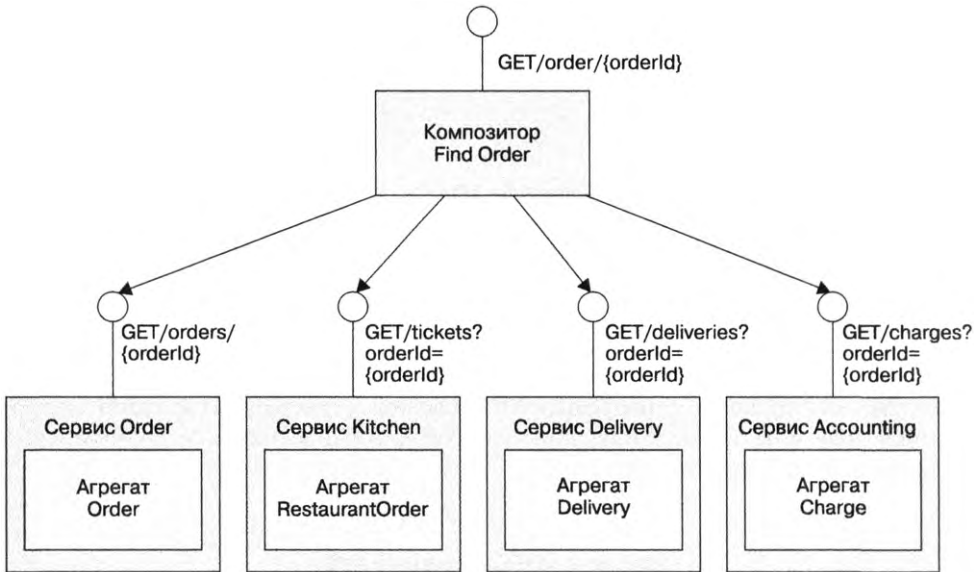


Рис. 7.3. Реализация findOrder() путем объединения API

В этом примере в качестве API-композитора выступает сервис, который делает запрос доступным в виде конечной точки REST. Сервисы-провайдеры тоже реализуют REST API. Но, даже если бы сервисы общались не по HTTP, а по какому-то другому протоколу межпроцессного взаимодействия, например gRPC, принцип работы

был бы тем же самым. Композитор `Find Order` реализует конечную точку REST `GET /order/{orderId}`. Он обращается к четырем сервисам и объединяет их ответы с помощью поля `orderId`. Каждый *сервис-провайдер* реализует конечную точку REST, которая возвращает ответ, соответствующий отдельному агрегату. `OrderService` извлекает свою версию `Order` по первичному ключу, а другие сервисы используют `orderId` в качестве внешнего ключа для извлечения собственных агрегатов.

Как видите, в объединении API нет ничего сложного. Теперь обсудим несколько архитектурных проблем, которые необходимо решить с помощью этого шаблона.

7.1.4. Архитектурные проблемы объединения API

При использовании этого шаблона необходимо решить две архитектурные проблемы.

- ❑ Какой компонент вашей архитектуры будет выступать *API-композитором* для операции запроса.
- ❑ Как написать эффективную логику агрегации.

Рассмотрим их.

Кто играет роль API-композитора

Одно из решений, которые вы должны принять, связано с тем, кто будет выступать *API-композитором* для операции запроса. У вас есть три кандидата на эту роль. Первый, клиент сервисов, показан на рис. 7.4.

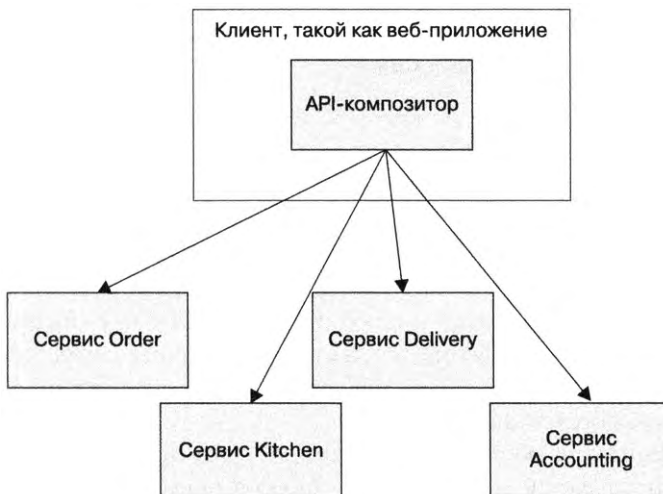


Рис. 7.4. Реализация объединения API на уровне клиента. Клиент запрашивает данные у сервисов-провайдеров

Такой клиент, как веб-приложение, реализующий представление *Order Status* и работающий в той же локальной сети, может эффективно извлечь подробности о заказе, используя данный шаблон. Но, как вы увидите в главе 8, этот вариант может не подойти для клиентов, которые находятся по другую сторону брандмауэра и обращаются к сервисам по медленной сети.

Второй кандидат на роль *API-композитора* — API-шлюз (рис. 7.5), реализующий внешний API приложения.

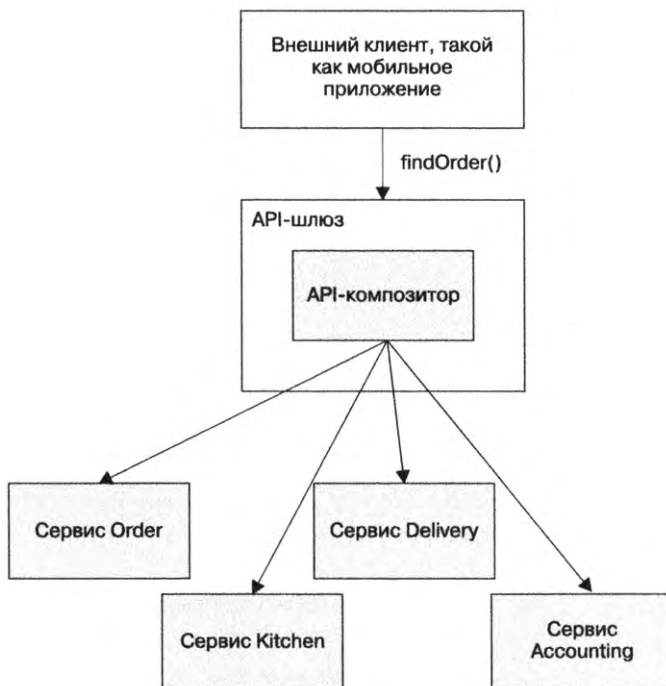


Рис. 7.5. Реализация объединения API внутри API-шлюза. Шлюз обращается к сервисам-провайдерам, чтобы извлечь данные, объединяет результаты и возвращает ответ клиенту

Этот вариант имеет смысл, если операция запроса входит в состав внешнего API. Вместо перенаправления запросов к другому сервису API-шлюз реализует логику объединения API. Такой подход позволяет клиенту (например, мобильному устройству), который находится за пределами брандмауэра, эффективно извлекать данные из многочисленных сервисов с помощью единственного API-вызова. API-шлюз обсуждается в главе 8.

Третий кандидат на роль *API-композитора* — отдельный сервис (рис. 7.6).

Этот вариант следует использовать для запросов, применяемых разными внутренними сервисами. Эту операцию могут задействовать также запросы, доступные извне, чья логика агрегации слишком сложна для того, чтобы делать ее частью API-шлюза.

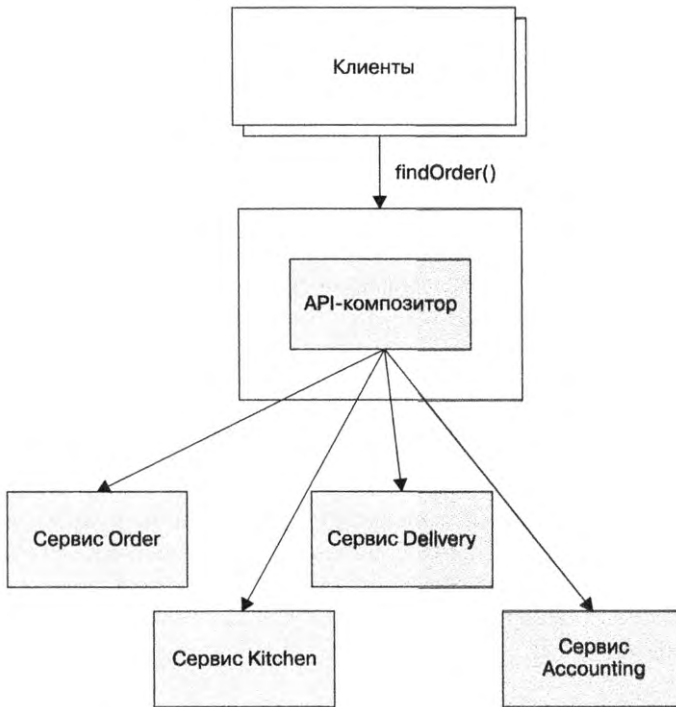


Рис. 7.6. Реализация операции запроса, которую используют разные клиенты и сервисы, в виде отдельного сервиса

API-композиторы должны использовать реактивную модель программирования

При разработке распределенных систем всегда нужно минимизировать латентность. Чтобы сократить время ответа на запрос, *API-компонитор* должен по возможности распараллеливать вызовы к сервисам-провайдерам. Например, агрегатор *Find Order* должен параллельно обратиться к четырем сервисам, поскольку между этими обращениями нет никаких зависимостей. Однако иногда *API-компонитору* нужен ответ одного из *сервисов-провайдеров*, чтобы обратиться к другому. В этом случае некоторые (но, надеюсь, не все) сервисы необходимо вызывать последовательно.

Логика сочетания последовательных и параллельных обращений к сервисам может оказаться довольно сложной. Чтобы *API-компонитор* был прост в обслуживании, но при этом быстро работал и хорошо масштабировался, он должен использовать методы реактивного проектирования на основе Java-класса `CompletableFuture`, наблюдаемых объектов из состава RxJava или аналогичных абстракций. Мы подробно обсудим эту тему в главе 8 при рассмотрении шаблона «API-шлюз».

7.1.5. Преимущества и недостатки объединения API

Этот шаблон — простой и интуитивно понятный способ реализации запросов в микросервисной архитектуре. Но у него есть некоторые недостатки:

- ❑ дополнительные накладные расходы;
- ❑ риск снижения доступности;
- ❑ нехватка транзакционной согласованности данных.

Рассмотрим их.

Дополнительные накладные расходы

Одной из отрицательных сторон этого шаблона являются накладные расходы при обращении к нескольким сервисам и базам данных. В монолитном приложении клиент может извлечь данные с помощью одного запроса, который выполняет лишь одно обращение к БД. Для сравнения: объединение API подразумевает выполнение нескольких запросов и обращений к БД. Это требует больше вычислительных и сетевых ресурсов, из-за чего обслуживание приложения становится более накладным.

Риск снижения доступности

Еще один недостаток шаблона связан со снижением уровня доступности. Как говорилось в главе 3, доступность операции снижается с увеличением количества вовлеченных в нее сервисов. Поскольку реализация запроса разделена между тремя или более участниками (*API-композитор* и по меньшей мере два сервиса-провайдера), ее доступность будет куда ниже, чем при использовании одного сервиса. Например, если доступность отдельного сервиса 99,5 %, то доступность конечной точки `findOrder()`, которая обращается к четырем сервисам-провайдерам, снижается до $99,5\%^{(4+1)} = 97,5\%$!

Существует несколько стратегий для улучшения доступности. Если *сервис-провайдер* недоступен, *API-композитор* может вернуть предварительно закэшированные данные. Иногда данные, возвращаемые *сервисом-провайдером*, кэшируются для повышения производительности, но с их помощью можно также улучшить доступность. Если провайдер не отвечает, *API-композитор* может вернуть данные из кэша, хотя при этом они могут оказаться устаревшими.

Еще одна стратегия улучшения доступности заключается в возвращении неполной информации. Представьте, к примеру, что сервис `Kitchen` временно недоступен. *API-композитор* для операции `findOrder()` может просто убрать данные этого сервиса из ответа, поскольку даже без них пользовательскому интерфейсу будет что показать. Больше подробностей о проектировании, кэшировании и надежности API вы найдете в главе 8.

Нехватка транзакционной согласованности данных

Еще одним недостатком объединения API является нехватка транзакционной согласованности данных. В монолитных приложениях запросы обычно выполняются в рамках одной транзакции базы данных. ACID-транзакции (при условии использования определенных уровней изоляции) гарантируют, что для приложения данные будут выглядеть согласованными даже во время выполнения нескольких запросов к БД. Для сравнения: при объединении API разные запросы направляются к разным базам данных. Поэтому существует риск того, что запрос вернет несогласованную информацию.

Например, заказ, извлеченный из сервиса Order, может находиться в состоянии CANCELLED, тогда как соответствующую заявку, полученную из сервиса Kitchen, еще не успели отменить. *API-композитор* должен устранить это несоответствие, что делает его код более сложным. Что еще хуже, в некоторых случаях *API-композитор* может не обнаружить, что данные не согласованы, и вернуть их клиенту.

Несмотря на эти недостатки, объединение API чрезвычайно полезно. Его можно применять для реализации множества запросов. Но есть такие операции, которые делают невозможной эффективную реализацию этого шаблона. Например, операция запроса может требовать, чтобы *API-композитор* объединил в памяти большие наборы данных.

Запросы такого рода лучше проектировать с использованием шаблона CQRS, о котором речь пойдет в дальнейшем.

7.2. Применение шаблона CQRS

Многие промышленные приложения используют СУРБД в качестве транзакционной системы записей, оставляя полнотекстовые запросы таким базам данных, как Elasticsearch или Solr. Некоторые приложения для поддержания синхронности одновременно записывают информацию в разные БД. Другие периодически копируют данные из СУРБД в поисковую систему. Такая архитектура использует преимущества нескольких баз данных: транзакционные свойства СУРБД и возможности полнотекстового поиска.

Шаблон «Разделение ответственности командных запросов»

Реализует запрос, которому нужны данные из нескольких сервисов. Для поддержания представления, реплицирующего данные из разных источников и доступного только для чтения, используются события. См. microservices.io/patterns/data/cqrs.html.

Обобщенной версией этой архитектуры является CQRS (command query responsibility segregation — разделение ответственности командных запросов). Она действует одно или несколько представлений базы данных (не только поисковой

системы), которые реализуют как минимум один запрос приложения. Чтобы понять, в чем польза этого подхода, рассмотрим несколько запросов, которые нельзя эффективно реализовать с помощью объединения API. Я объясню принцип работы CQRS и расскажу о достоинствах и недостатках этого шаблона. Давайте посмотрим, в каких ситуациях его нужно применять.

7.2.1. Потенциальные причины использования CQRS

Объединение API хорошо подходит для реализации многих запросов, которые должны извлекать данные из разных сервисов. К сожалению, в микросервисной архитектуре это лишь частичное решение проблемы. Существует множество запросов, которые нельзя эффективно спроектировать с помощью этого шаблона.

Кроме того, трудности могут возникнуть и с запросами, не выходящими за пределы одного сервиса. Возможно, база данных сервиса не поддерживает эффективные запросы, а иногда запрос лучше спроектировать таким образом, чтобы он запрашивал информацию, принадлежащую другому сервису. Обсудим эти проблемы. Начнем с многосервисных запросов, которые нельзя эффективно реализовать с помощью объединения API.

Реализация операции запроса `findOrderHistory()`

Операция `findOrderHistory()` извлекает историю заказов клиента. Она имеет несколько параметров.

- ❑ `consumerId` — идентифицирует клиента.
- ❑ `pagination` — страница результатов, которую нужно вернуть.
- ❑ `filter` — фильтр по критериям, таким как максимальная давность возвращаемых заказов, необязательный статус заказа и опциональные ключевые слова, которые соответствуют названию ресторана и пунктам меню.

Этот запрос возвращает объект `OrderHistory`, который содержит краткую информацию о подходящих заказах, отсортированных по давности в порядке возрастания. Он вызывается модулем, реализующим представление `OrderHistory`. Это представление выводит краткое описание каждого заказа, включая его номер, статус, итоговую сумму и ожидаемое время доставки.

На первый взгляд эта операция похожа на `findOrder()`. Единственное отличие в том, что она возвращает несколько заказов вместо одного. Вам может показаться, что API-компоненту достаточно выполнить один и тот же запрос для каждого сервиса-провайдера и затем объединить результаты. К сожалению, не все так просто.

Дело в том, что не все сервисы хранят атрибуты, используемые для фильтрации или сортировки. Например, одним из критериев фильтрации в запросе `findOrderHistory()` служит ключевое слово, которое совпадает с пунктом меню. Только два сервиса, `Order` и `Kitchen`, хранят заказанные позиции меню. Сервисы `Delivery` и `Accounting` этого не делают, поэтому не могут отфильтровать данные по ключевому слову. По этой же причине сервисы `Kitchen` и `Delivery` не могут сортировать значения по атрибуту `orderCreationDate`.

У *API-композитора* есть два варианта решения этой проблемы. Он может выполнить слияние в памяти (рис. 7.7). Для этого он извлекает все заказы клиента из сервисов **Delivery** и **Accounting**, после чего объединяет их с заказами, полученными из сервисов **Order** и **Kitchen**.

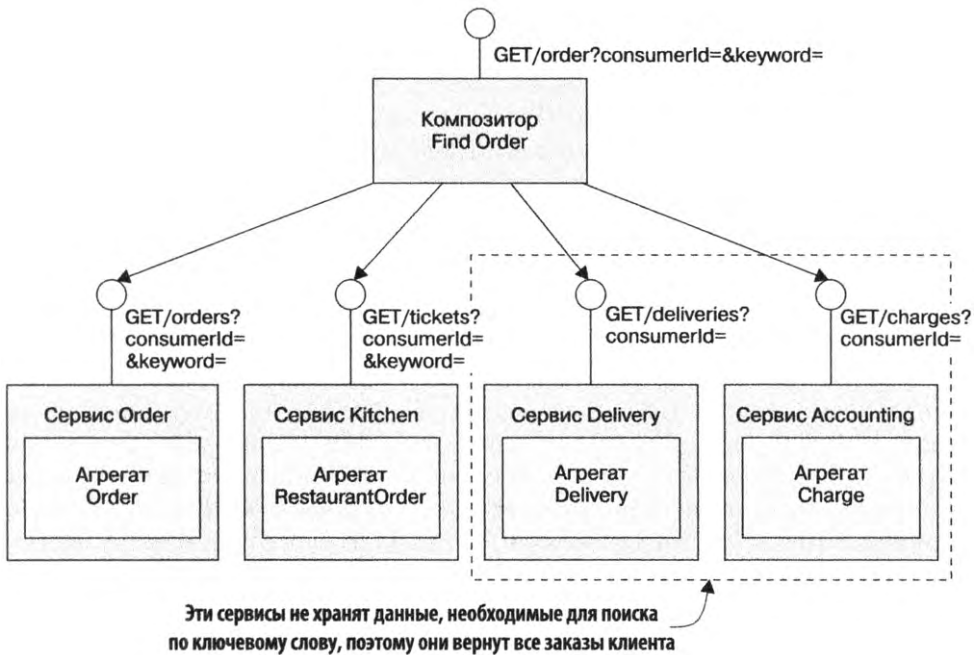


Рис. 7.7. При объединении API нельзя эффективно извлечь заказы клиента, поскольку некоторые провайдеры, такие как сервис **Delivery**, не хранят атрибуты, используемые для фильтрации

Недостаток этого подхода в том, что *API-композитору*, возможно, придется извлекать и объединять большие наборы данных. Это было бы неэффективно.

Есть еще одно решение. *API-композитор* может извлечь подходящие заказы из сервисов **Order** и **Kitchen** и затем запросить заказы из других сервисов по их ID. Но это имеет смысл только в том случае, если API этих сервисов поддерживает массовое извлечение. Запрашивать заказы по отдельности, скорее всего, будет неэффективно из-за лишнего сетевого трафика.

Такие операции, как `findOrderHistory()`, требуют, чтобы *API-композитор* дублировал функции системы выполнения запросов СУРБД. С одной стороны, это может сместить нагрузку с менее масштабируемой базы данных на более масштабируемое приложение. Но с другой — это окажется не так эффективно. К тому же разработчики должны реализовывать бизнес-возможности, а не систему выполнения запросов.

Чуть позже я покажу, как применять шаблон CQRS и использовать отдельное хранилище, которое подходит для эффективной реализации запроса `findOrderHistory()`. Но сначала рассмотрим пример операции, которую непросто реализовать, несмотря на то что она не выходит за рамки одного сервиса.

Непростой односервисный запрос `findAvailableRestaurants()`

Как вы только что видели, реализация запросов, извлекающих данные из разных сервисов, может создать определенные трудности. Но проблемы способны возникнуть даже с запросами, локальными для одного сервиса. Вызвать их могут несколько причин. Во-первых, как вы вскоре узнаете, запрос не должен реализовываться сервисом, который владеет данными. Во-вторых, определенные запросы могут оказаться неэффективными с точки зрения базы (или модели) данных сервиса.

Возьмем, к примеру, операцию `findAvailableRestaurants()`. Она находит рестораны, которые могут доставить еду по заданному адресу и в заданное время. В ее основе лежит геопространственный (основанный на местоположении) поиск ресторанов, расположенных на определенном расстоянии от адреса доставки. Это важная часть процесса заказа, инициируемая модулем пользовательского интерфейса, отображающим доступные рестораны.

Ключевой аспект этого запроса — выполнение эффективного геопространственного поиска. То, как вы реализуете операцию `findAvailableRestaurants()`, зависит от возможностей базы данных, которая хранит сведения о ресторанах. Например, это легко сделать в MongoDB или с помощью геопространственных расширений для Postgres и MySQL. Эти БД поддерживают геопространственные типы данных, индексы и запросы. При их использовании сервис `Restaurant` сохраняет информацию о ресторане в виде записи с атрибутом `location`. Чтобы найти доступные рестораны, он применит запрос, оптимизированный благодаря геопространственному индексу по атрибуту `location`.

Если приложение FTGO хранит сведения о ресторанах в другой базе данных, реализация запроса `findAvailableRestaurant()` окажется более трудной. Мы должны хранить копию данных о ресторанах в формате, который поддерживает геопространственный поиск. Приложение, к примеру, могло бы использовать библиотеку Geospatial Indexing для DynamoDB (github.com/awslabs/dynamodb-geo), в которой таблица выступает в роли геопространственного индекса. Как вариант, копию данных о ресторанах можно хранить в совершенно другой базе данных. Это очень похоже на ситуацию, когда для текстовых запросов применяется система полнотекстового поиска.

Сложность репликации данных связана с тем, что их необходимо поддерживать в актуальном состоянии при изменении оригинала. Как вы вскоре узнаете, проблема синхронизации реплик решается с помощью CQRS.

Необходимость в разделении ответственности

Еще одна проблема с односервисными запросами связана с тем, что иногда за их реализацию должен отвечать не тот сервис, который владеет данными. Операция `findAvailableRestaurants()` извлекает данные, принадлежащие сервису `Restaurant`. Этот сервис позволяет владельцам ресторанов управлять своими профилями и меню. Он хранит различные атрибуты, включая название ресторана, его адрес, кулинарную направленность, меню и время работы. Сервис владеет данными, поэтому, по

крайней мере на первый взгляд, было бы логично поручить ему реализацию запроса. Однако принадлежность данных — не единственный фактор, который следует учитывать.

Вы также должны помнить о необходимости разделения ответственности и не возлагать на сервисы слишком много обязанностей. Например, основная обязанность команды разработки сервиса `Restaurant` состоит в том, чтобы позволить администрации управлять своим рестораном. Это имеет мало общего с реализацией интенсивной и критически важной операции. Кроме того, если бы эти разработчики отвечали за запрос `findAvailableRestaurants()`, им пришлось бы постоянно беспокоиться о том, что любое развертываемое ими изменение может нарушить функцию размещения заказов.

Лучше сделать так, чтобы сервис `Restaurant` предоставлял данные, а реализацию запроса `findAvailableRestaurants()` возложить на другую команду — скорее всего, это будут разработчики сервиса `Order`. Как в случае с операцией `findOrderHistory()`, если вам нужно поддерживать геопространственный индекс, вы должны хранить копию данных с отложенной согласованностью. Посмотрим, как этого добиться с помощью CQRS.

7.2.2. Обзор CQRS

Примеры, описанные в подразделе 7.2.1, продемонстрировали три проблемы, которые часто встречаются при реализации запросов в микросервисной архитектуре.

- ❑ Объединение API для извлечения данных, разбросанных по разным сервисам, приводит к затратным малоэффективным операциям JOIN, выполняемым в памяти.
- ❑ Сервис, владеющий данными, хранит их в формате или базе данных, которые не имеют эффективной поддержки нужного запроса.
- ❑ Необходимость разделения ответственности означает, что реализацией запроса должен заниматься не тот сервис, который владеет данными.

Шаблон CQRS решает все эти проблемы.

CQRS разделяет команды и запросы

CQRS расширяется как разделение ответственности командных запросов. Как следует из названия, этот шаблон предназначен для *разделения обязанностей*. На рис. 7.8 показано, как он разделяет хранимую модель данных и использующие ее модули на две части: команды и запросы. Командные модули и модель данных реализуют операции создания, обновления и удаления (`create`, `update` и `delete`, CUD), которые соответствуют HTTP-командам POST, PUT и DELETE. Модули запросов и модель данных реализуют запросы, соответствующие HTTP-команде GET. Сторона запросов синхронизирует свою модель данных с моделью данных командной стороны, подписываясь на события, которые та публикует.

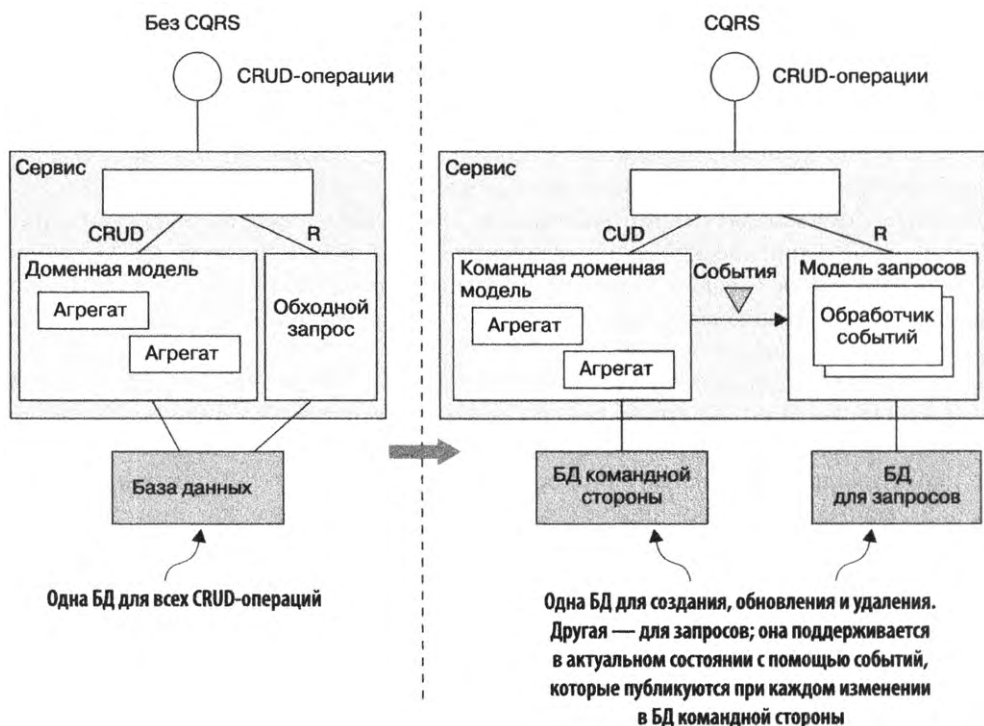


Рис. 7.8. Сервис справа поддерживает CQRS, а слева — нет. CQRS разделяет сервис на модули команд и запросов, которые имеют отдельные базы данных

У обеих версий сервиса (с CQRS и без него) есть API, состоящий из различных CRUD-операций. В сервисе, не основанном на CQRS, эти операции обычно реализуются доменной моделью, привязанной к базе данных. Для улучшения производительности некоторые запросы могут миновать доменную модель и обращаться к базе данных напрямую. Единая хранимая модель данных поддерживает и команды, и запросы.

В сервисе, основанном на CQRS, доменная модель командной стороны обрабатывает CRUD-операции и привязана к собственной базе данных. Она может обрабатывать также простые запросы, использующие первичные ключи и не содержащие операций слияния. Командная сторона публикует события при каждом изменении своих данных. Для этого может задействоваться фреймворк, такой как Eventuate Tram, или порождение событий.

За нетривиальные запросы отвечает отдельная модель. Она намного проще по сравнению с командной стороной, потому что ей не нужно реализовывать бизнес-правила. Чтобы поддерживать необходимые запросы, эта модель использует подходящую для этого базу данных. Она содержит обработчики, которые подписываются на доменные события и обновляют базу (-ы) данных. Таких моделей может быть несколько, по одной для каждого вида запросов.

CQRS и сервисы, предназначенные только для запросов

Помимо использования внутри сервиса, CQRS можно применять для описания запрашивающих сервисов. API запрашивающего сервиса состоит только из запросов и не поддерживает командные операции. Для реализации запросов он обращается к базе данных, которую поддерживает в актуальном состоянии, подписываясь на события, публикуемые одним или несколькими сервисами. Сервис стороны запросов — это хороший способ реализовать представление, которое формируется благодаря подписке на события, генерируемые разными сервисами. Такое представление имеет смысл реализовать отдельно, так как оно не относится ни к одному из существующих сервисов. Хороший пример — запрашивающий сервис `Order History`, который реализует запрос `findOrderHistory()`. Он подписывается на события, публикуемые несколькими сервисами, включая `Order`, `Delivery` и т. д. (рис. 7.9).

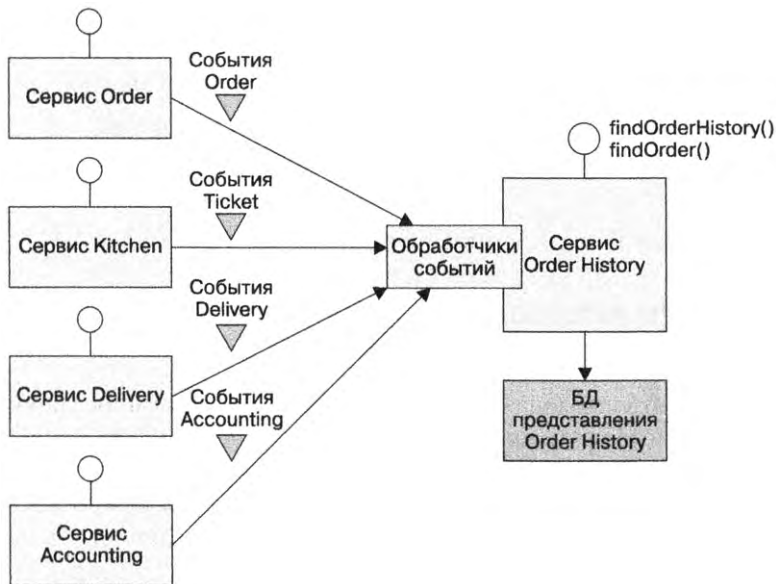


Рис. 7.9. Структура запрашивающего сервиса `Order History`. Он реализует операцию запроса `findOrderHistory()`, обращаясь к базе данных, актуальность которой поддерживается путем подписки на события, публикуемые другими сервисами

У сервиса `Order History` есть обработчики, которые подписываются на события, публикуемые несколькими сервисами, и обновляют базу данных представления `Order history`. Реализация этого сервиса подробнее рассматривается в разделе 7.4.

Запрашивающий сервис также хорошо подходит для реализации представления, которое реплицирует данные, принадлежащие одному сервису, но из-за разделения ответственности не являющиеся его частью. Например, разработчики FTGO могут определить сервис `Available Restaurants`, который реализует операцию `findAvailableRestaurants()`, описанную ранее. Он подписывается на события,

публикуемые сервисом **Restaurant**, и обновляет базу данных с эффективной поддержкой геопространственного поиска.

По большому счету, **CQRS** — это обобщенная разновидность популярной методики, которая заключается в том, что СУРБД используется в качестве системы записей, а поисковая система, такая как **Elasticsearch**, отвечает за полнотекстовый поиск. Но есть одно отличие: в **CQRS** применяется более широкий диапазон баз данных, а не только система полнотекстового поиска. Кроме того, благодаря подписке на события представления стороны запросов в **CQRS** обновляются почти в режиме реального времени.

Взвесим достоинства и недостатки **CQRS**.

7.2.3. Преимущества **CQRS**

CQRS имеет как сильные, так и слабые стороны. Начнем с сильных.

- ❑ Возможность эффективной реализации запросов в микросервисной архитектуре.
- ❑ Возможность эффективной реализации разнородных запросов.
- ❑ Возможность выполнения запросов в приложении, основанном на порождении событий.
- ❑ Улучшенное разделение ответственности.

Возможность эффективной реализации запросов в микросервисной архитектуре

Одно из преимуществ шаблона **CQRS** — эффективная реализация запросов, которые извлекают данные из нескольких сервисов. Как упоминалось ранее, запросы, основанные на объединении **API**, иногда приводят к неэффективному слиянию больших наборов данных прямо в памяти. В таких случаях нужно использовать легкодоступное **CQRS**-представление, которое заранее объединяет данные из двух или более сервисов.

Возможность эффективной реализации разнородных запросов

Еще одно преимущество шаблона **CQRS** состоит в том, что он позволяет приложению или сервису эффективно реализовывать разнородные запросы. Поддержка всех запросов в рамках одной хранимой модели данных часто трудна, а иногда и просто невозможна. Некоторые базы данных **NoSQL** умеют выполнять только очень ограниченные запросы. Но даже если у БД есть расширение для поддержки запросов определенного вида, использование специализированной базы данных часто более эффективно. Шаблон **CQRS** позволяет избежать ограничений конкретного хранилища данных за счет определения одного или нескольких представлений, каждое из которых эффективно реализует тот или иной запрос.

Возможность выполнения запросов в приложении, основанном на порождении событий

CQRS позволяет преодолеть основное ограничение порождения событий. Хранилище событий поддерживает только запросы по первичному ключу. CQRS устраняет эту проблему, создавая для агрегатов одно или несколько представлений, поддерживаемых в актуальном состоянии. Для этого обработчики подписываются на потоки событий, публикуемые агрегатами. В итоге приложения, основанные на порождении событий, неизменно используют CQRS.

Улучшенное разделение ответственности

Еще одной сильной стороной CQRS является разделение ответственности. Доменная модель и соответствующая модель данных занимаются либо командами, либо запросами. Шаблон CQRS предназначает отдельные программные модули и структуру базы данных для двух разных частей сервиса. Разделение командной стороны и стороны запросов во многих случаях упрощает код и облегчает его поддержку.

Более того, благодаря CQRS за реализацию запроса и хранение данных могут отвечать разные сервисы. Например, ранее я продемонстрировал, что, хотя сервис `Restaurant` и владеет данными, к которым обращается `findAvailableRestaurants`, такую важную и интенсивную операцию лучше вынести за его пределы. В CQRS, чтобы поддерживать представление в актуальном состоянии, запрашивающий сервис подписывается на события, публикуемые другими сервисами, которым принадлежат сами данные.

7.2.4. Недостатки CQRS

Наряду с преимуществами CQRS имеет и существенные недостатки:

- ❑ более сложную архитектуру;
- ❑ отставание репликации.

Рассмотрим эти проблемы, начав с возрастающей сложности.

Более сложная архитектура

Один из недостатков шаблона CQRS связан с повышением сложности. Разработчикам приходится писать запрашивающие сервисы, которые отвечают за обновление представлений и обращение к ним. Код усложняется также за счет администрирования и обслуживания дополнительных хранилищ данных. Более того, приложение может использовать разные виды баз данных, что добавляет головной боли как разработчикам, так и администраторам.

Отставание репликации

Еще один недостаток CQRS связан с последствиями рассинхронизации между представлениями для команд и запросов. Как можно было бы ожидать, между публикацией события командной стороной, его обработкой запрашивающим сервисом и обновлением представления проходит некоторое время. Клиентское приложение, которое обновляет агрегат и сразу же обращается к представлению, может получить предыдущую версию агрегата. Подобный код часто пишут таким образом, чтобы пользователь не сталкивался с потенциальной несогласованностью.

Одно из решений заключается в том, чтобы API для команд и запросов предоставляли клиентам сведения о версии. Это позволит определить, устарел ли результат запроса. Клиент может периодически опрашивать представление, пока не получит актуальную информацию. Чуть позже я объясню, как API сервисов могут предоставить клиентам такую возможность.

Скомпилированное мобильное приложение или одностраничный веб-сайт на JavaScript, которые реализуют пользовательский интерфейс, могут справиться с отставанием репликации за счет обновления своей локальной модели в ответ на успешное выполнение команды без применения запроса. Для обновления модели они могут, к примеру, использовать данные, возвращенные командой, и надеяться на то, что к моменту, когда пользователь инициирует запрос, представление успеет синхронизироваться. Один из недостатков этого подхода — то, что для обновления своей модели пользовательский интерфейс, возможно, должен будет дублировать серверный код.

Как видите, у CQRS есть сильные и слабые стороны. Ранее уже упоминалось, что объединение API следует использовать везде, где это возможно, а CQRS — только там, где необходимо.

Итак, вы познакомились с преимуществами и недостатками CQRS. Теперь посмотрим, как проектировать представления с помощью этого шаблона.

7.3. Проектирование CQRS-представлений

Модуль CQRS-представления обладает API, состоящим из одного или нескольких запросов. Для реализации этих запросов CQRS обращается к базе данных, которая обновляется за счет подписки на события, публикуемые одним или несколькими сервисами. Этот модуль содержит базу данных представления и три дочерних модуля (рис. 7.10).

Модуль доступа к данным реализует логику обращения к БД. Модули обработчиков событий и API для запросов используют модуль доступа к данным для обновления и обращения к БД. Модуль обработчиков событий подписывается на события и обновляет базу данных. Модуль с API для запросов реализует эти API.

При разработке модуля представления необходимо принять несколько важных решений по поводу архитектуры.

- ❑ Выбрать базу данных и спроектировать ее структуру.
- ❑ При проектировании модуля доступа к данным решить ряд проблем, включая поддержку идемпотентных и конкурентных обновлений.

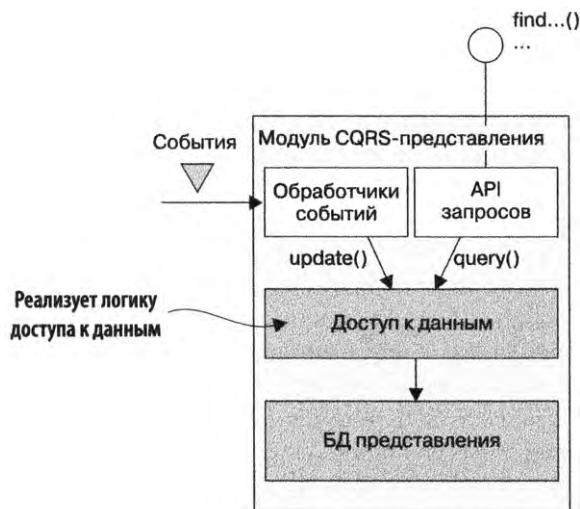


Рис. 7.10. Структура модуля CQRS-представления. Обработчики событий обновляют базу данных представления, доступ к которой выполняется через модуль API запросов

- ❑ При реализации нового представления в существующем приложении или изменении структуры готового проекта реализовать механизм эффективного построения (или перестраивания) представлений.
- ❑ Решить, каким образом клиент будет справляться с отставанием репликации, описанным ранее.

Рассмотрим каждый из этих аспектов.

7.3.1. Выбор хранилища данных для представления

Ключевым архитектурным решением является выбор базы данных и проектирование ее структуры. Основная задача базы и модели данных заключается в эффективной реализации запросов модуля представления. Именно характеристики этих запросов становятся основным фактором при выборе базы данных. Но БД также должна эффективно реализовывать операции обновления, выполняемые обработчиками событий.

Выбор между SQL и NoSQL

Не так давно СУРБД на основе SQL были единственным видом баз данных, которые использовались для всего на свете. Но с ростом популярности Интернета разные компании начали замечать, что СУРБД больше не отвечают их требованиям к масштабируемости. Это привело к созданию так называемых NoSQL-хранилищ. Базы данных NoSQL обычно поддерживают ограниченный набор транзакций и запросов. В некоторых сценариях они имеют определенные преимущества перед БД

на основе SQL, включая более гибкую модель данных, а также лучшие производительность и масштабируемость.

Базы данных NoSQL обычно хорошо подходят для CQRS-представлений, которые способны использовать их сильные стороны и игнорировать недостатки. CQRS-представлению идут на пользу более развитая модель данных и высокая производительность NoSQL. Ему не страшны ограничения этой технологии, поскольку оно применяет лишь простые транзакции и выполняет фиксированный набор запросов.

Несмотря на все сказанное, в некоторых случаях CQRS-представления лучше реализовывать с помощью баз данных с поддержкой SQL. Современные СУРБД, запущенные на современном оборудовании, демонстрируют отличную производительность. У разработчиков, администраторов баз данных и DevOps обычно больше опыта работы с SQL, чем с NoSQL. Как упоминалось ранее, у СУРБД часто есть расширения для нереляционных функций, таких как геопространственные типы данных и запросы. Кроме того, CQRS-представлению может понадобиться база данных на основе SQL для поддержки системы отчетов.

Как показано в табл. 7.1, вам есть из чего выбирать. Выбор осложняется тем, что границы между разными видами баз данных становятся все менее четкими. Например, сервер MySQL, который является СУРБД, имеет прекрасную поддержку JSON, в то же время это одна из сильных сторон MongoDB — документно-ориентированной БД, основанной на этом формате.

Таблица 7.1. Хранилища для представлений на стороне запросов

Если вам нужно	Используйте	Пример
Поиск JSON-объектов по первичному ключу	Документное хранилище наподобие MongoDB либо DynamoDB или хранилище типа «ключ — значение» вроде Redis	Реализация истории заказов посредством хранения документов MongoDB для каждого клиента
Поиск JSON-объектов на основе запросов	Документное хранилище наподобие MongoDB или DynamoDB	Реализация пользовательского интерфейса для клиентов с помощью MongoDB или DynamoDB
Текстовые запросы	Система полнотекстового поиска вроде Elasticsearch	Реализация текстового поиска по заказам путем хранения заказов в виде документов Elasticsearch
Графовые запросы	Графовая база данных, такая как Neo4j	Реализация обнаружения мошенничества посредством хранения графа клиентов, заказов и других данных
Традиционные SQL-отчеты или BI ¹	СУРБД	Стандартные бизнес-отчеты и аналитика

¹ https://ru.wikipedia.org/wiki/Business_Intelligence.

Мы обсудили разные виды баз данных, с помощью которых можно реализовать CQRS-представление. Теперь поговорим о том, как его эффективно обновлять.

Вспомогательные операции обновления

Помимо эффективной реализации запросов, модель данных представления должна предоставлять эффективные операции обновления, которые выполняются обработчиками событий. Обычно для обновления или удаления записей в БД представления обработчики используют первичные ключи. Например, чуть позже я опишу структуру CQRS-представления для запроса `findOrderHistory()`. Каждый заказ в нем хранится в виде записи базы данных с `orderId` в качестве первичного ключа. При получении события `Order Service` представление просто обновляет соответствующую запись.

Однако иногда ему придется обновлять или удалять записи, используя эквивалент внешнего ключа. Возьмем, к примеру, обработчики событий `Delivery*`. Если между `Delivery` и `Order` существует отношение вида «один к одному», то `Delivery.id` может совпадать с `Order.id`. В этом случае обработчики событий `Delivery*` могут легко обновить запись о заказе.

Но представьте, что у агрегата `Delivery` есть собственный первичный ключ или что `Order` и `Delivery` имеют отношение вида «один ко многим». Некоторые события типа `Delivery*`, такие как `DeliveryCreated`, будут содержать `orderId`, но другие, наподобие `DeliveryPickedUp`, — нет. В этом случае обработчику событий `DeliveryPickedUp` придется обновлять запись о заказе, используя `deliveryId` в качестве аналога внешнего ключа.

Некоторые виды баз данных имеют эффективную поддержку обновлений на основе внешнего ключа. Например, при задействовании СУРБД или MongoDB вам нужно создать индекс для соответствующего столбца. Однако в других NOSQL-хранилищах выполнить обновления без применения первичных ключей может оказаться затруднительно. Приложению придется специально поддерживать некую связь между внешними и первичными ключами, чтобы знать, какую запись следует обновить. Например, DynamoDB поддерживает обновления и удаления только по первичному ключу, поэтому приложение, использующее эту БД, сначала должно запросить ее вторичный индекс (о нем чуть позже), чтобы определить первичные ключи обновляемых или удаляемых элементов.

7.3.2. Структура модуля доступа к данным

Обработчики событий и модуль API запросов не обращаются к хранилищу данных напрямую. Вместо этого они задействуют специальный модуль, который состоит из объекта доступа к данным (DAO) и его вспомогательных классов. У DAO есть несколько обязанностей. Он реализует операции обновления, инициируемые обработчиками событий, и операции запросов, которые вызываются модулем запросов. DAO накладывает типы данных, которые применяются в высокоуровневом коде,

на API БД. Кроме того, он должен поддерживать конкурентные и идемпотентные обновления.

Рассмотрим все эти аспекты, начиная с конкурентных обновлений.

Поддержка конкурентности

Иногда объект DAO должен уметь справляться с конкурентными обновлениями одной и той же записи базы данных. Если представление подписывается на события, публикуемые агрегатами одного типа, никаких проблем с конкурентностью возникнуть не может. Дело в том, что события, которые публикуются определенным экземпляром агрегата, обрабатываются последовательно. Благодаря этому запись, относящаяся к экземпляру агрегата, не будет обновляться параллельно. Но если события, на которые подписано представление, публикуются агрегатами разных типов, существует вероятность того, что несколько обработчиков одновременно попытаются обновить одну и ту же запись.

Например, обработчики событий **Order*** и **Delivery*** могут быть вызваны в один и тот же момент для одного и того же заказа. В этом случае они одновременно обратятся к DAO, чтобы обновить запись базы данных для этого экземпляра **Order**. Объект DAO должен быть написан так, чтобы такие ситуации улаживались корректно. Он не должен позволять одному обновлению перезаписывать другое. Если для реализации обновления объект DAO считывает и затем записывает измененную запись, он должен использовать пессимистичное или оптимистичное блокирование. В следующем разделе вы увидите пример поддержки конкурентных обновлений за счет изменения записи без ее предварительного считывания.

Идемпотентные обработчики событий

Как упоминалось в главе 3, обработчик может быть вызван больше одного раза для одного и того же события. Это не составляет проблемы, если обработчик на стороне запросов идемпотентный. В этом случае результат обработки повторяющихся событий будет корректным. Худшее, что может произойти, — это временная рассинхронизация хранилища данных представления. Например, обработчик, отвечающий за представление **Order History**, может получить маловероятную последовательность событий **DeliveryPickedUp**, **DeliveryDelivered**, **DeliveryPickedUp** и **DeliveryDelivered** (рис. 7.11). Допустим, после изначальной доставки событий **DeliveryPickedUp** и **DeliveryDelivered** брокер сообщений столкнулся с сетевыми проблемами. В результате доставка возобновилась с более раннего момента времени, что привело к повторной передаче **DeliveryPickedUp** и **DeliveryDelivered**.

После того как событие **DeliveryPickedUp** пройдет обработку во второй раз, представление **Order History** будет какое-то время иметь неактуальное состояние заказа. Это продлится до тех пор, пока не закончится обработка **DeliveryDelivered**. Если такое поведение нежелательно, обработчик должен уметь определять и отклонять повторяющиеся события, как это делают неидемпотентные обработчики.

Обработчик событий неидемпотентный, если повторяющиеся события приводят к некорректным результатам. Например, обработчик, который инкрементирует баланс на банковском счету, не поддерживает идемпотентность. Как объяснялось в главе 3, неидемпотентные обработчики должны обнаруживать и отклонять дубликаты, записывая идентификаторы уже обработанных событий в хранилище данных представления.



Рис. 7.11. События `DeliveryPickedUp` и `DeliveryDelivered` доставляются два раза, из-за чего в представлении временно рассинхронизируется состояние заказа

Для надежной работы обработчик должен записывать ID событий и обновлять хранилище данных атомарным образом. То, как это сделать, зависит от используемой базы данных. Если хранилище данных представления основано на SQL, обработчик может вставлять обработанные события в таблицу `PROCESSED_EVENTS` в рамках транзакции, обновляющей представление. Но если вы применяете NoSQL-хранилище с ограниченной транзакционной моделью, обработчик должен сохранять события внутри записей (например, документа MongoDB или элемента таблицы DynamoDB), которые он обновляет.

Следует отметить, что обработчику не нужно записывать ID каждого события. Если, как в случае с Eventuate, события имеют монотонно растущие идентификаторы, достаточно хранить в каждой записи значение `max(eventId)`, полученное из заданного экземпляра агрегата. Это справедливо для тех случаев, когда одна запись соответствует одному экземпляру агрегата. Если записи представляют собой слияние событий из разных агрегатов, они должны содержать словарь, связывающий `[aggregate type, aggregate id]` с `max(eventId)`.

Например, вскоре вы увидите, что реализация представления `Order History` на основе DynamoDB содержит элементы с атрибутами для отслеживания событий, которые имеют следующий вид:

```
{...
  "Order3949384394-039434903" : "0000015e0c6fc18f-0242ac1100e50002",
  "Delivery3949384394-039434903" : "0000015e0c6fc264-0242ac1100e50002",
}
```

Это представление является объединением событий, публикуемых разными сервисами. Имя каждого атрибута для отслеживания событий выглядит как "aggregateType""aggregateId", а значение равно eventId. Позже я подробнее объясню, как это работает.

Клиентские приложения могут использовать представления с отложенной согласованностью

Как упоминалось ранее, одна из проблем CQRS состоит в том, что клиент, который обновляет командную сторону и затем немедленно выполняет запрос, может не увидеть собственное обновление. Ввиду неизбежных задержек в инфраструктуре обмена сообщениями представление является отложено согласованным.

API модулей команд и запросов позволяют клиенту обнаружить несогласованность с помощью следующего подхода. Операция командной стороны возвращает клиенту токен с идентификатором опубликованного события. Клиент указывает этот токен в операции запроса. Если представление не обновилось в результате этого события, операция вернет ошибку. Модуль представления может реализовать этот механизм, используя систему обнаружения повторяющихся событий.

7.3.3. Добавление и обновление CQRS-представлений

CQRS-представления добавляются и обновляются на протяжении жизненного цикла приложения. Иногда дополнительное представление требуется для поддержки нового запроса. Временами из-за изменения структуры БД или необходимости исправить ошибку в коде, который занимается обновлением, представление приходится создавать заново.

Добавление и обновление представлений сами по себе довольно просты. Чтобы создать новое представление, нужно разработать модуль стороны запросов, подготовить хранилище данных и развернуть сервис. Обработчики в модуле стороны запросов пропускают через себя все события, благодаря чему представление рано или поздно актуализируется. В обновлении существующих представлений тоже нет ничего сложного: вам нужно изменить обработчики событий и перестроить набор данных с нуля. Однако проблема этого подхода в том, что он вряд ли будет работать в реальных условиях. Посмотрим, что с ним не так.

Построение CQRS-представлений с помощью заархивированных событий

Одна из проблем связана с тем, что брокер не может хранить сообщения бесконечно. Традиционные брокеры, такие как RabbitMQ, удаляют сообщения, обработанные потребителем. И даже более современные аналоги наподобие Apache Kafka хранят сообщения на протяжении ограниченного времени, указанного в конфигурации. Так что представление нельзя построить, считав все необходимые события из бро-

кера сообщений. Вместо этого приложению нужно прочитать более старые события, заархивированные, скажем, в AWS S3. Это можно сделать с помощью масштабируемой технологии для хранения больших данных, такой как Apache Spark.

Инкрементальное построение CQRS-представлений

Еще одна проблема, связанная с созданием представлений, состоит в том, что для обработки всех событий требуется все больше времени и ресурсов. В какой-то момент этот процесс становится слишком медленным и затратным. В качестве решения можно воспользоваться двухэтапным инкрементальным алгоритмом. Первый этап периодически вычисляет снимок экземпляров каждого агрегата с учетом предыдущего снимка и событий, произошедших с момента его создания. На втором этапе с помощью снимков и любых последующих событий создается представление.

7.4. Реализация CQRS с использованием AWS DynamoDB

Итак, мы обсудили различные архитектурные проблемы, которые необходимо решить с помощью CQRS. Теперь рассмотрим пример. В этом разделе описывается реализация CQRS-представления для операции `findOrderHistory()` с применением DynamoDB. AWS DynamoDB — это масштабируемая база данных типа NoSQL, доступная в виде сервиса в облаке Amazon. Ее модель данных состоит из таблиц, которые содержат элементы, представляющие собой набор иерархических пар «ключ — значение» (по примеру JSON-объектов). База данных AWS DynamoDB полностью управляема, поэтому вы можете динамически масштабировать пропускную способность отдельных таблиц в обе стороны.

CQRS-представление для операции `findOrderHistory()` потребляет события из нескольких сервисов, поэтому оно реализуется в виде отдельного сервиса `Order View`. Этот сервис имеет API, который реализует две операции: `findOrderHistory()` и `findOrder()`. И хотя последнюю можно реализовать с помощью объединения API, в этом представлении она предоставляется фактически даром. Структура сервиса `Order History` показана на рис. 7.12. Он состоит из набора модулей, каждый из которых имеет определенную обязанность, что упрощает разработку и тестирование. Далее перечислены обязанности модулей.

- ❑ `OrderHistoryEventHandlers` — подписывается на события, публикуемые различными сервисами, и вызывает `OrderHistoryDAO`.
- ❑ *Модуль API* `OrderHistoryQuery` — реализует конечные точки REST, описанные ранее.
- ❑ `OrderHistoryDataAccess` — содержит объект `OrderHistoryDAO`, который определяет методы для обновления и обращения к таблице DynamoDB `ftgo-order-history`, а также его вспомогательные классы.
- ❑ *Таблица DynamoDB* `ftgo-order-history` — хранит заказы.

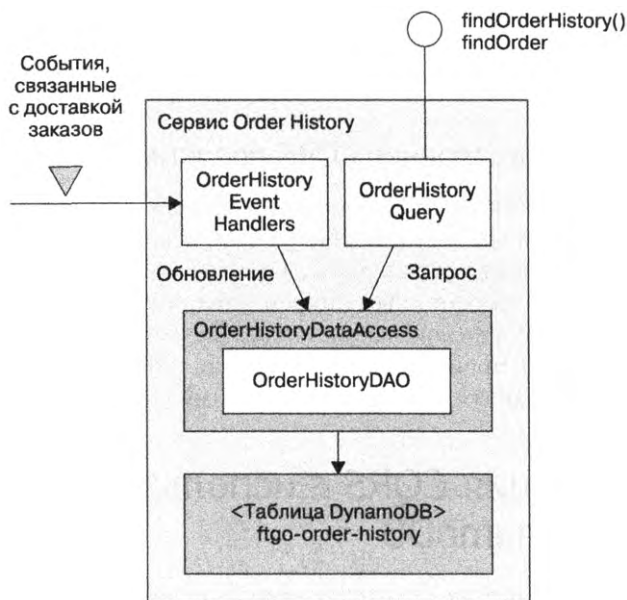


Рис. 7.12. Структура OrderHistoryService. Модуль OrderHistoryEventHandlers обновляет базу данных в ответ на события. Модуль OrderHistoryQuery реализует запросы, обращаясь к базе данных. Оба они используют модуль OrderHistoryDataAccess для доступа к БД

Давайте подробнее рассмотрим структуру обработчиков событий, объекта DAO и таблицы DynamoDB.

7.4.1. Модуль OrderHistoryEventHandlers

Модуль состоит из обработчиков, которые потребляют события и обновляют таблицу DynamoDB. Как можно видеть в листинге 7.1, эти обработчики представляют собой простые однострочные методы, которые вызывают OrderHistoryDao с аргументами, сформированными на основе события.

Листинг 7.1. Обработчики событий, которые вызывают OrderHistoryDao

```

public class OrderHistoryEventHandlers {

    private OrderHistoryDao orderHistoryDao;

    public OrderHistoryEventHandlers(OrderHistoryDao orderHistoryDao) {
        this.orderHistoryDao = orderHistoryDao;
    }

    public void handleOrderCreated(DomainEventEnvelope<OrderCreated> dee) {
        orderHistoryDao.addOrder(makeOrder(dee.getAggregateId(), dee.getEvent()),
            makeSourceEvent(dee));
    }
}

```

```
}

private Order makeOrder(String orderId, OrderCreatedEvent event) {
    ...
}

public void handleDeliveryPickedUp(DomainEventEnvelope<DeliveryPickedUp>
                                   dee) {
    orderHistoryDao.notePickedUp(dee.getEvent().getOrderId(),
                                  makeSourceEvent(dee));
}

...
}
```

У каждого обработчика есть параметр типа `DomainEventEnvelope`, который содержит само событие и определенные метаданные с его описанием. Например, метод `handleOrderCreated()` вызывается для обработки события `OrderCreated`. Он вызывает `orderHistoryDao.addOrder()`, чтобы создать заказ в базе данных. Точно так же метод `handleDeliveryPickedUp()` вызывается для обработки события `DeliveryPickedUp`. Чтобы обновить состояние заказа в базе данных, он вызывает `orderHistoryDao.notePickedUp()`.

В обоих случаях применяется вспомогательный метод `makeSourceEvent()`, который создает объект `SourceEvent`, содержащий идентификатор события, а также тип и ID агрегата, который его сгенерировал. В следующем разделе вы увидите, что объект `OrderHistoryDao` использует `SourceEvent`, чтобы обеспечить идемпотентность операции обновления.

Теперь взглянем на структуру таблицы DynamoDB, после чего обследуем объект `OrderHistoryDao`.

7.4.2. Моделирование данных и проектирование запросов с помощью DynamoDB

Операции доступа к данным, предоставляемые DynamoDB, как и многими другими БД типа NoSQL, куда менее гибки по сравнению с теми, которые можно встретить в СУРБД. В связи с этим вы должны тщательно подходить к выбору модели хранения данных. В частности, структура таблиц во многих случаях определяется нужными вам запросами. Вы должны решить несколько архитектурных задач.

- ❑ Проектирование таблицы `ftgo-order-history`.
- ❑ Определение индекса для запроса `findOrderHistory`.
- ❑ Реализация запроса `findOrderHistory`.
- ❑ Разбиение результатов запроса на страницы.
- ❑ Обновление заказов.
- ❑ Обнаружение повторяющихся событий.

Далее поговорим о каждой из них.

Проектирование таблицы `ftgo-order-history`

Модель хранилища `DynamoDB` состоит из таблиц, которые содержат элементы и индексы, предоставляющие альтернативные способы доступа к этим элементам (об этом чуть позже). *Элемент* — это набор именных атрибутов. *Значением атрибута* может быть скалярная величина, такая как строка, коллекция из нескольких строк или набор именных атрибутов. И хотя элемент является эквивалентом строки в СУРБД, он куда более гибок и может хранить целый агрегат.

Эта гибкость `DynamoDB` позволяет модулю `OrderHistoryDataAccess` хранить каждый заказ в виде отдельного элемента в таблице `ftgo-order-history`. Каждое поле класса `Order` накладывается на атрибут элемента (рис. 7.13). Простые поля, такие как `orderCreationTime` и `status`, соответствуют атрибутам с единственным значением. Поле `lineItems` привязывается к атрибуту, содержащему список ассоциативных массивов — по одному на каждую строку. В JSON это можно было бы назвать массивом объектов.

Таблица ftgo-order-history						
Первичный ключ	orderId	consumerId	orderCreationTime	status	lineItems	...
	...	xyz-abc	22939283232	CREATED	[{...}, {...},]	...

Рис. 7.13. Предварительная структура таблицы `OrderHistory` в `DynamoDB`

Важной частью определения таблицы является ее первичный ключ. В `DynamoDB` он используется для вставки, обновления и извлечения элементов. Было бы логично выбрать в качестве первичного ключа поле `orderId`. Это позволит сервису `OrderHistory` вставлять, обновлять и извлекать заказы по их идентификаторам. Но прежде, чем окончательно принять это решение, посмотрим, как первичный ключ таблицы влияет на то, какого рода операции доступа к данным она поддерживает.

Определение индекса для запроса `findOrderHistory`

Определение этой таблицы поддерживает чтение и запись заказов по первичному ключу. Но в нем отсутствует поддержка некоторых запросов, например `findOrderHistory()`, который возвращает несколько подходящих заказов, отсортированных по тому, как давно они сделаны. Как вы позже увидите в данном разделе, это связано с тем, что запросы в `DynamoDB` задействуют операцию `query()`, которая требует, чтобы первичный ключ таблицы состоял из двух скалярных атрибутов. Первый атрибут — это *ключ секции*. Он так называется из-за того, что `DynamoDB` использует его при масштабировании по оси *Z* (см. главу 1), чтобы выбрать для эле-

мента секцию хранилища. Вторым атрибутом служит ключ *сортировки*. Операция `query()` возвращает элементы с заданным ключом секции, при этом они должны соответствовать выражению фильтрации (если таковое имеется), а их ключи сортировки должны относиться к указанному диапазону. Кроме того, ключ сортировки определяет порядок, в котором возвращаются элементы.

Операция запроса `findOrderHistory()` возвращает заказы клиента, отсортированные по давности в порядке возрастания. В связи с этим ей нужен первичный ключ, который состоит из ключа секции `consumerId` и ключа сортировки `orderCreationDate`. Однако ключ `(consumerId, orderCreationDate)` неуникальный, поэтому применять его в качестве первичного в таблице `ftgo-order-history` бессмысленно.

Чтобы решить эту проблему, операция `findOrderHistory()` при обращении к таблице `ftgo-order-history` должна использовать то, что в DynamoDB называется *вторичным индексом*. Он содержит `(consumerId, orderCreationDate)` в качестве неуникального ключа. Как и СУРБД, DynamoDB автоматически обновляет свои индексы при изменении таблицы. Однако в DynamoDB атрибуты индексов могут не быть ключами, что нехарактерно для реляционных БД. *Неключевые атрибуты* улучшают производительность, так как их возвращает запрос, благодаря чему приложение может не запрашивать их из таблицы. К тому же, как вы вскоре увидите, с их помощью можно выполнять фильтрацию. Структура таблицы и упомянутый ранее индекс показаны на рис. 7.14.

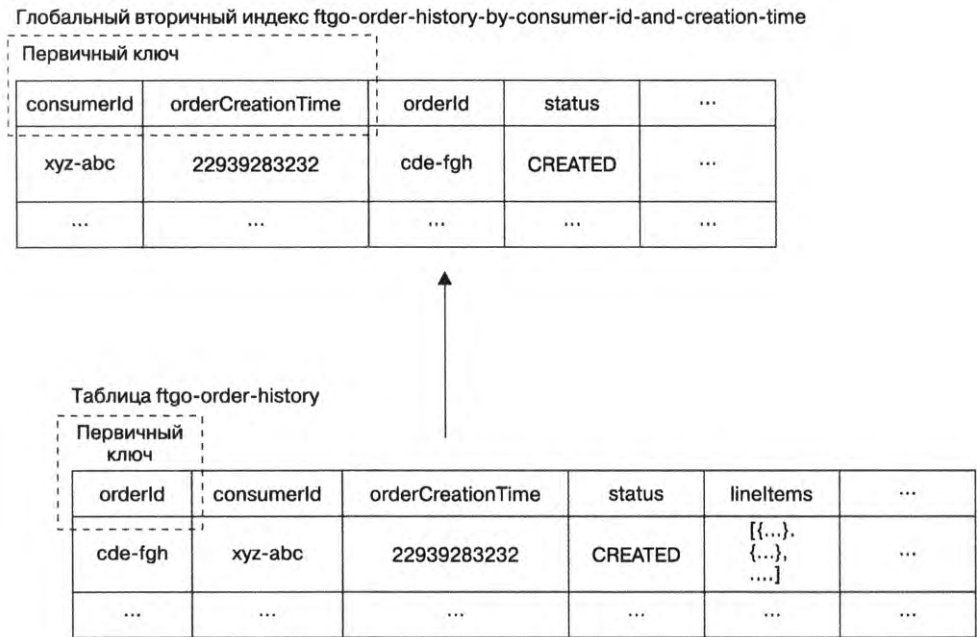


Рис. 7.14. Структура и индекс таблицы OrderHistory

Данный индекс является частью определения таблицы `ftgo-order-history` и называется `ftgo-order-history-by-consumer-id-and-creation-time`. Он состоит из атрибутов первичного ключа, `consumerId` и `orderCreationTime`, а также неключевых атрибутов, таких как `orderId` и `status`.

Индекс `ftgo-order-history-by-consumer-id-and-creation-time` позволяет объекту `OrderHistoryDaoDynamoDb` эффективно извлекать заказы клиента, отсортированные по давности в порядке возрастания.

Теперь посмотрим, как извлечь только заказы, соответствующие критериям фильтра.

Реализация запроса `findOrderHistory`

У операции запроса `findOrderHistory()` есть параметр `filter`, который определяет критерии поиска. Одним из критериев фильтрации является максимальная давность возвращаемых заказов. Это легко реализовать, поскольку в `DynamoDB` *ключевое условное выражение* запроса поддерживает ограничение по диапазону для ключа сортировки. Еще один критерий относится к неключевым атрибутам и может быть реализован с помощью *выражения фильтрации* булева типа. Операция запроса `DynamoDB` возвращает только те элементы, которые удовлетворяют выражению фильтрации. Например, чтобы найти заказы с состоянием `CANCELLED`, объект `OrderHistoryDaoDynamoDb` может использовать выражение `orderStatus = :orderStatus`, где `:orderStatus` — подставляемый параметр.

Реализация критериев фильтрации по ключевым словам не так проста. Она выбирает заказы, у которых название ресторана или пункты меню совпадают с одним из заданных ключевых слов. Чтобы выполнить поиск по ключевым словам, объект `OrderHistoryDaoDynamoDb` разбивает названия ресторанов и пункты меню на термины, которые хранятся в массиве внутри атрибута `keywords`. Для поиска заказов, соответствующих ключевым словам, он задействует функцию `contains()`, например `contains(keywords, :keyword1) OR contains(keywords, :keyword2)`, где `:keyword1` и `:keyword2` — подставляемые параметры для заданных ключевых слов.

Разбиение результатов запроса на страницы

У некоторых клиентов будет много заказов. Поэтому логично сделать так, чтобы операция `findOrderHistory()` возвращала их постранично. В `DynamoDB` операции запросов имеют параметр `pageSize`, который определяет максимальное количество возвращаемых элементов. Если элементов оказывается больше, результат запроса будет содержать ненулевой атрибут `LastEvaluatedKey`. Чтобы извлечь следующую страницу, объект DAO может указать параметру запроса `exclusiveStartKey` значение `LastEvaluatedKey`.

Как видите, `DynamoDB` не поддерживает секционное разбиение на страницы. Следовательно, сервис `Order History` возвращает своему клиенту непрозрачный токен, с помощью которого тот может запросить следующую страницу с результатами.

Обновление заказов

DynamoDB поддерживает операции `PutItem()` и `UpdateItem()` для добавления и обновления элементов соответственно. `PutItem()` создает или заменяет по первичному ключу целый элемент. Теоретически объект `OrderHistoryDaoDynamoDb` мог бы использовать эту операцию для обновления заказов. Но применение данного подхода затрудняет то, что требуется обеспечить корректную обработку одновременных обновлений.

Представьте, к примеру, что два обработчика событий одновременно пытаются обновить один и тот же элемент. Каждый из них обращается к `OrderHistoryDaoDynamoDb`, чтобы загрузить этот элемент из DynamoDB, изменить его в памяти и обновить запись, взяв `PutItem()`. Один обработчик событий потенциально может перезаписать изменения, внесенные другим. Чтобы предотвратить потерю изменений, `OrderHistoryDaoDynamoDb` может воспользоваться механизмом оптимистичного блокирования из состава DynamoDB. Однако проще и эффективнее применить операцию `UpdateItem()`.

Операция `UpdateItem()` обновляет отдельные атрибуты элемента или создает его целиком, если это необходимо. Ее использование имеет смысл, поскольку разные обработчики изменяют разные атрибуты заказа. К тому же эта операция более эффективна, потому что не требует предварительного извлечения заказа из таблицы.

Как упоминалось ранее, одной из проблем обновления базы данных в ответ на события является обнаружение дубликатов. Посмотрим, как это сделать с помощью DynamoDB.

Обнаружение повторяющихся событий

Все обработчики событий в сервисе `Order History` идемпотентны. Каждый из них устанавливает один или несколько атрибутов для элемента `Order`. Таким образом, сервис `Order History` может просто игнорировать проблему повторяющихся событий. Однако в этом случае элемент `Order` будет периодически устаревать. Дело в том, что обработчик, который принимает повторяющееся событие, назначит атрибутам элемента `Order` предыдущие значения. То есть элемент окажется неактуальным, пока не наступят следующие события.

Как упоминалось ранее, чтобы предотвратить устаревание данных, повторяющиеся события можно обнаруживать и отклонять. Для этого объект `OrderHistoryDaoDynamoDb` может записывать в элементы события, которые инициировали их обновление. Затем он может воспользоваться механизмом условного обновления из операции `UpdateItem()`, чтобы изменять элементы, только если событие не является дубликатом.

Условное обновление производится только при выполнении *условного выражения*. Это выражение проверяет существование атрибута или наличие в нем определенного значения. DAO-объект `OrderHistoryDaoDynamoDb` может отслеживать события, полученные из каждого экземпляра агрегата, для этого он использует атрибут `"aggregateType"` `"aggregateId"`, чье значение равно наивысшему ID принятого события. Если атрибут существует и его значение меньше или равно этому ID,

событие является дубликатом. `OrderHistoryDaoDynamoDb` применяет следующее условное выражение:

```
attribute_not_exists("aggregateType""aggregateId")
OR "aggregateType""aggregateId" < :eventId
```

Условное выражение позволяет обновить элемент, только если атрибута не существует или `eventId` больше, чем ID последнего обработанного события.

Представьте, к примеру, что обработчик получает из агрегата `Delivery` с ID `3949384394-039434903` событие `DeliveryPickup`, чей идентификатор равен `123323-343434`. Отслеживающий атрибут называется `Delivery3949384394-039434903`. Обработчик должен рассматривать событие в качестве дубликата, если значение этого атрибута больше или равно `123323-343434`. Операция `query()`, вызываемая обработчиком событий, обновляет элемент `Order` с помощью условного выражения:

```
attribute_not_exists(Delivery3949384394-039434903)
OR Delivery3949384394-039434903 < :eventId
```

Вы познакомились с моделью данных и структурой запросов `DynamoDB`. Теперь рассмотрим класс `OrderHistoryDaoDynamoDb`, который обновляет таблицу `ftgo-order-history` и запрашивает из нее данные.

7.4.3. Класс `OrderHistoryDaoDynamoDb`

Класс `OrderHistoryDaoDynamoDb` реализует методы для чтения и записи элементов в таблице `ftgo-order-history`. Его операции обновления вызываются из `OrderHistoryEventHandlers`, а запросы иницируются `API OrderHistoryQuery`. Рассмотрим примеры некоторых методов, начиная с `addOrder()`.

Метод `addOrder()`

Метод `addOrder()`, представленный в листинге 7.2, добавляет заказ в таблицу `ftgo-order-history`. У него есть два параметра — `order` и `sourceEvent`. Параметр `order` — это заказ, который нужно добавить, он извлекается из события `OrderCreated`. Параметр `sourceEvent` содержит `eventId`, а также тип и идентификатор агрегата, который сгенерировал событие. Он используется для реализации условных обновлений.

Листинг 7.2. Метод `addOrder()` добавляет или обновляет заказ

```
public class OrderHistoryDaoDynamoDb ...

@Override
public boolean addOrder(Order order, Optional<SourceEvent> eventSource) {
    UpdateItemSpec spec = new UpdateItemSpec()
        .withPrimaryKey("orderId", order.getOrderId())
        .withUpdateExpression("SET orderStatus = :orderStatus, " +
            "creationDate = :cd, consumerId = :consumerId, lineItems = " +
            ":lineItems, keywords = :keywords, restaurantName = " +
            ":restaurantName")
}
```

Первичный ключ обновляемого элемента Order

Выражение, обновляющее атрибуты

Значения
подставляемых
параметров
в выражении
обновления

```

    .withValueMap(new Maps()
        .add(":orderStatus", order.getStatus().toString())
        .add(":cd", order.getCreationDate().getMillis())
        .add(":consumerId", order.getConsumerId())
        .add(":lineItems", mapLineItems(order.getLineItems()))
        .add(":keywords", mapKeywords(order))
        .add(":restaurantName", order.getRestaurantName())
        .map())
    .withReturnValues(ReturnValue.NONE);
    return idempotentUpdate(spec, eventSource);
}

```

Метод `addOrder()` создает объект `UpdateSpec`, который входит в состав AWS SDK и описывает операцию обновления. После этого он вызывает вспомогательный метод `idempotentUpdate()`, который добавляет условное выражение, предохраняющее от дубликатов, и выполняет обновление.

Метод `notePickedUp()`

Метод `notePickedUp()`, показанный в листинге 7.3, вызывается обработчиком событий типа `DeliveryPickedUp`. Он меняет состояние `deliveryStatus` элемента `Order` на `PICKED_UP`.

Листинг 7.3. Метод `notePickedUp()` меняет состояние заказа на `PICKED_UP`

```

public class OrderHistoryDaoDynamoDb ...

@Override
public void notePickedUp(String orderId, Optional<SourceEvent> eventSource) {
    UpdateItemSpec spec = new UpdateItemSpec()
        .withPrimaryKey("orderId", orderId)
        .withUpdateExpression("SET #deliveryStatus = :deliveryStatus")
        .withNameMap(Collections.singletonMap("#deliveryStatus",
            DELIVERY_STATUS_FIELD))
        .withValueMap(Collections.singletonMap(":deliveryStatus",
            DeliveryStatus.PICKED_UP.toString()))
        .withReturnValues(ReturnValue.NONE);
    idempotentUpdate(spec, eventSource);
}

```

Этот метод похож на `addOrder()`. Он создает объект `UpdateItemSpec` и делает вызов `idempotentUpdate()`. О последнем речь пойдет далее.

Метод `idempotentUpdate()`

В листинге 7.4 показан метод `idempotentUpdate()`, который обновляет элемент после возможного добавления условного выражения в объект `UpdateItemSpec`, предохраняющий от повторяющихся обновлений.

Листинг 7.4. Метод `idempotentUpdate()` игнорирует повторяющиеся события

```
public class OrderHistoryDaoDynamoDb ...

private boolean idempotentUpdate(UpdateItemSpec spec, Optional<SourceEvent>
    eventSource) {
    try {
        table.updateItem(eventSource.map(es -> es.addDuplicateDetection(spec))
            .orElse(spec));
        return true;
    } catch (ConditionalCheckFailedException e) {
        // Ничего не делаем
        return false;
    }
}
```

Если указать параметр `sourceEvent`, метод `idempotentUpdate()` сделает вызов `SourceEvent.addDuplicateDetection()`, чтобы добавить в `UpdateItemSpec` условное выражение, описанное ранее. Метод `idempotentUpdate()` перехватывает и игнорирует исключение `ConditionalCheckFailedException`, которое генерируется вызовом `updateItem()` в случае, если событие является дубликатом.

Просмотрев код, который обновляет таблицу, можем перейти к методу запроса.

Метод `findOrderHistory()`

Метод `findOrderHistory()`, представленный в листинге 7.5, извлекает заказы клиента. Для этого он обращается к таблице `ftgo-order-history` по вторичному индексу `ftgo-order-history-by-consumer-id-and-creation-time`. У него есть два параметра: `consumerId`, который идентифицирует клиента, и `filter`, определяющий поисковые критерии. На основе своих параметров этот метод формирует объект `QuerySpec`, который, как и `UpdateSpec`, входит в состав AWS SDK. Затем он делает запрос по индексу и преобразует возвращенные элементы в объект `OrderHistory`.

Листинг 7.5. Метод `findOrderHistory()` извлекает подходящие заказы клиента

```
public class OrderHistoryDaoDynamoDb ...

@Override
public OrderHistory findOrderHistory(String consumerId, OrderHistoryFilter
    filter) {
    QuerySpec spec = new QuerySpec()
        .withScanIndexForward(false)
        .withHashKey("consumerId", consumerId)
        .withRangeKeyCondition(new RangeKeyCondition("creationDate")
            .gt(filter.getSince().getMillis()));

    filter.getStartKeyToken().ifPresent(token ->
        spec.withExclusiveStartKey(toStartingPrimaryKey(token)));
```

Запрос должен вернуть заказы по возрастанию давности

Максимальная давность возвращаемых заказов

```

Map<String, Object> valuesMap = new HashMap<>();

String filterExpression = Expressions.and(
    keywordFilterExpression(valuesMap, filter.getKeywords()),
    statusFilterExpression(valuesMap, filter.getStatus()));

if (!valuesMap.isEmpty())
    spec.withValueMap(valuesMap);

if (StringUtils.isNotBlank(filterExpression)) {
    spec.withFilterExpression(filterExpression);
}

filter.getPageSize().ifPresent(spec::withMaxResultSize);

ItemCollection<QueryOutcome> result = index.query(spec);

return new OrderHistory(
    StreamSupport.stream(result.spliterator(), false)
        .map(this::toOrder)
        .collect(toList()),
    Optional.ofNullable(result
        .getLastLowLevelResult()
        .getQueryResult().getLastEvaluatedKey())
        .map(this::toStartKeyToken));
}

```

Формируем фильтрующее выражение и словарь для подстановочных параметров на основе OrderHistoryFilter

Ограничиваем число результатов, если вызывающая сторона указала размер страницы

Создаем Order из элемента, возвращенного запросом

После построения объекта QuerySpec этот метод выполняет запрос и создает экземпляр OrderHistory со списком заказов на основе возвращенных элементов.

Метод findOrderHistory() реализует разбиение на страницы путем сериализации значения, полученного из getLastEvaluatedKey(), в токен формата JSON. Если клиент укажет в OrderHistoryFilter начальный токен, findOrderHistory() его сериализует и вызовет withExclusiveStartKey(), чтобы установить начальный ключ.

Как видите, в ходе реализации CQRS-представления необходимо решить целый ряд вопросов, включая выбор БД, проектирование модели данных, которая эффективно реализует обновления и запросы, обработку конкурентных обновлений и способность справляться с повторяющимися обновлениями. Единственной сложной частью кода является объект DAO, поскольку он должен иметь корректную поддержку конкурентности и обеспечивать идемпотентность обновлений.

Резюме

- ❑ Реализация запросов, извлекающих данные из разных сервисов, сопровождается определенными затруднениями, так как данные каждого сервиса приватные.
- ❑ Для реализации такого рода запросов можно использовать одну из методик: объединение API или разделение ответственности командных запросов (CQRS).

- ❑ Шаблон объединения API собирает данные от разных сервисов. Это самый простой способ реализации запросов. Его следует применять везде, где это возможно.
- ❑ Ограничение шаблона объединения API связано с тем, что некоторые сложные запросы требуют неэффективного слияния в памяти больших наборов данных.
- ❑ Шаблон CQRS, реализующий запросы с помощью БД представлений, более мощный, но его не так просто создать.
- ❑ Модуль представлений CQRS должен поддерживать конкурентные обновления, а также обнаружение и отклонение повторяющихся событий.
- ❑ CQRS способствует разделению ответственности за счет того, что сервисы могут реализовывать запросы, которые возвращают данные других сервисов.
- ❑ Клиенты должны быть готовы к отложенной согласованности CQRS-представлений.

Шаблоны внешних API

В этой главе

- Трудности проектирования API с поддержкой разнородных клиентов.
- Применение API-шлюза шаблона BFF.
- Проектирование и реализация API-шлюза.
- Упрощение объединения API с помощью реактивного программирования.
- Реализация API-шлюза на основе GraphQL.

У FTGO, как и у многих других приложений, есть REST API. В число его клиентов входят мобильные приложения, код на JavaScript, работающий в браузере, и программы, разработанные партнерами. В такой монолитной архитектуре API, видимый снаружи, и сам монолитный. Но после того, как команда FTGO начала разворачивать микросервисы, единого API больше нет, поскольку у каждого сервиса теперь свой API. Мэри и ее коллеги должны решить, какого рода API приложение FTGO должно сделать доступным для своих клиентов. Например, должны ли клиенты знать о существовании сервисов и обращаться к ним напрямую?

Проектирование внешнего API приложения усложняется разнообразием его клиентов. Обычно разным клиентам нужны различные данные. Настольный пользовательский веб-интерфейс, как правило, должен выводить намного больше информации, чем мобильное приложение. Кроме того, для обращения к сервисам разные клиенты могут пользоваться разными сетями. Например, клиент в пределах брандмауэра работает по высокоскоростной локальной сети, тогда как внешние клиенты

применяют Интернет или мобильную сеть с более низкой производительностью. В итоге, как вы сами увидите, наличие единого универсального API часто оказывается нецелесообразным.

Я начну эту главу с описания различных проблем, связанных с проектированием внешних API. Затем опишу соответствующие шаблоны проектирования. Мы рассмотрим концепцию API-шлюза и шаблон BFF. После этого я покажу, как спроектировать и реализовать API-шлюз. Вам будут представлены доступные варианты, включая готовые продукты, а также фреймворки для разработки собственной реализации. Я опишу архитектуру и реализацию API-шлюза, построенного на основе фреймворка Spring Cloud Gateway. Также вы узнаете, как создать API-шлюз с помощью фреймворка GraphQL, предоставляющего язык запросов, основанный на графах.

8.1. Проблемы с проектированием внешних API

Чтобы исследовать проблемы, связанные с API, рассмотрим приложение FTGO. Его сервисами пользуется целый ряд клиентов четырех типов (рис. 8.1):

- ❑ веб-приложения, реализующие браузерные пользовательские интерфейсы для заказчиков, ресторанов и администраторов. Последний является внутренним;
- ❑ JavaScript-приложение, работающее в браузере;
- ❑ мобильные приложения — одно для заказчиков, второе для курьеров;
- ❑ приложения, написанные сторонними разработчиками.

Веб-приложения работают в пределах брандмауэра, поэтому обращаются к сервисам по высокоскоростной локальной сети с низкой латентностью. Другие клиенты находятся снаружи, поэтому их запросы к сервисам передаются по Интернету или мобильной сети, обладающим меньшей скоростью и более высокой латентностью.

Один из вариантов проектирования API состоит в том, что клиенты обращаются к сервисам напрямую. На первый взгляд это звучит довольно просто — в конце концов, именно так клиенты вызывают API в монолитных приложениях. Но этот подход редко применяется в микросервисной архитектуре из-за следующих недостатков.

- ❑ Для извлечения нужных данных с помощью мелко раздробленных API клиентам приходится выполнять несколько запросов. Это неэффективно, к тому же клиенты могут получить отрицательный опыт взаимодействия с сервисом.
- ❑ Недостаточная инкапсуляция, связанная с тем, что клиенты знают о каждом сервисе и его API, затрудняет внесение изменений в архитектуру и API.
- ❑ Сервисы могут задействовать механизмы IPC, которые нецелесообразно или неудобно использовать на клиентской стороне, особенно клиентам за пределами брандмауэра.

Чтобы узнать больше об этих недостатках, посмотрим, как мобильное потребительское приложение FTGO извлекает данные из сервисов.

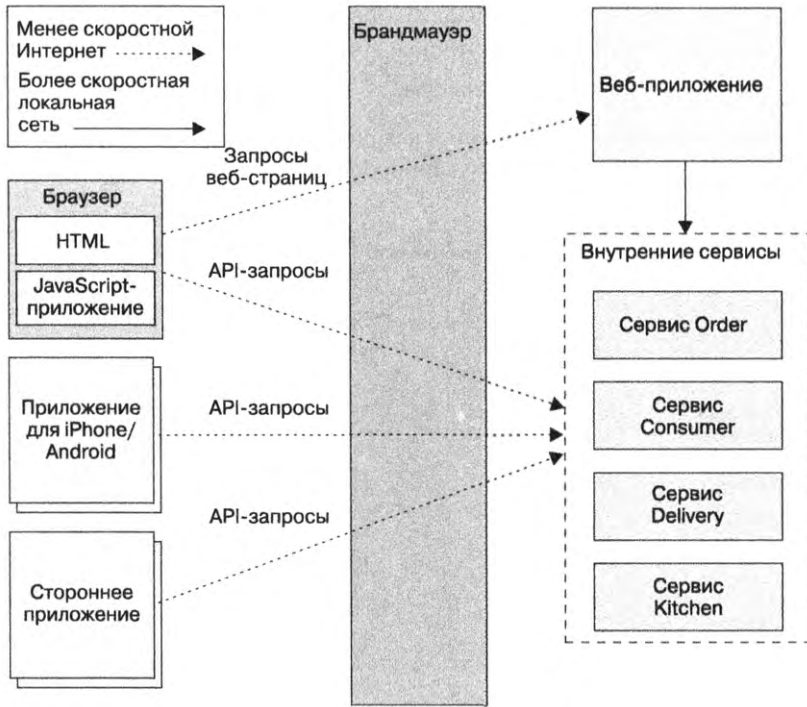


Рис. 8.1. Сервисы приложения FTGO и их клиенты. Есть несколько видов клиентов. Одни работают в пределах брандмауэра, другие — нет. Те, что находятся снаружи, обращаются к сервисам по Интернету или мобильной сети с относительно низкой скоростью. Внутренние клиенты применяют высокоскоростную локальную сеть

8.1.1. Проблемы проектирования API для мобильного клиента FTGO

Для размещения заказов и управления ими потребители используют мобильный клиент FTGO. Представьте, что в ходе его разработки вам нужно написать представление `View Order` для отображения заказа. Как говорилось в главе 7, это представление выводит общую информацию, такую как состояние заказа с точки зрения пользователя и ресторана, а также состояние платежа, местоположение курьера и предположительное время доставки, если она уже в пути.

У API монолитной FTGO есть конечная точка, которая возвращает подробности о заказе. Чтобы извлечь нужную информацию, мобильному клиенту достаточно сделать один запрос. Но в микросервисной архитектуре детали заказа разбросаны по нескольким сервисам, включая следующие:

- ❑ **Order** — основная информация, включая подробности и статус;
- ❑ **Kitchen** — статус заказа с точки зрения ресторана и то, когда он должен быть готов к доставке;

- ❑ **Delivery** — состояние доставки заказа, предполагаемая информация о доставке и местоположение курьера в настоящее время;
- ❑ **Accounting** — состояние оплаты заказа.

Если мобильный клиент обращается к сервисам напрямую, для извлечения данных ему придется сделать несколько запросов (рис. 8.2).

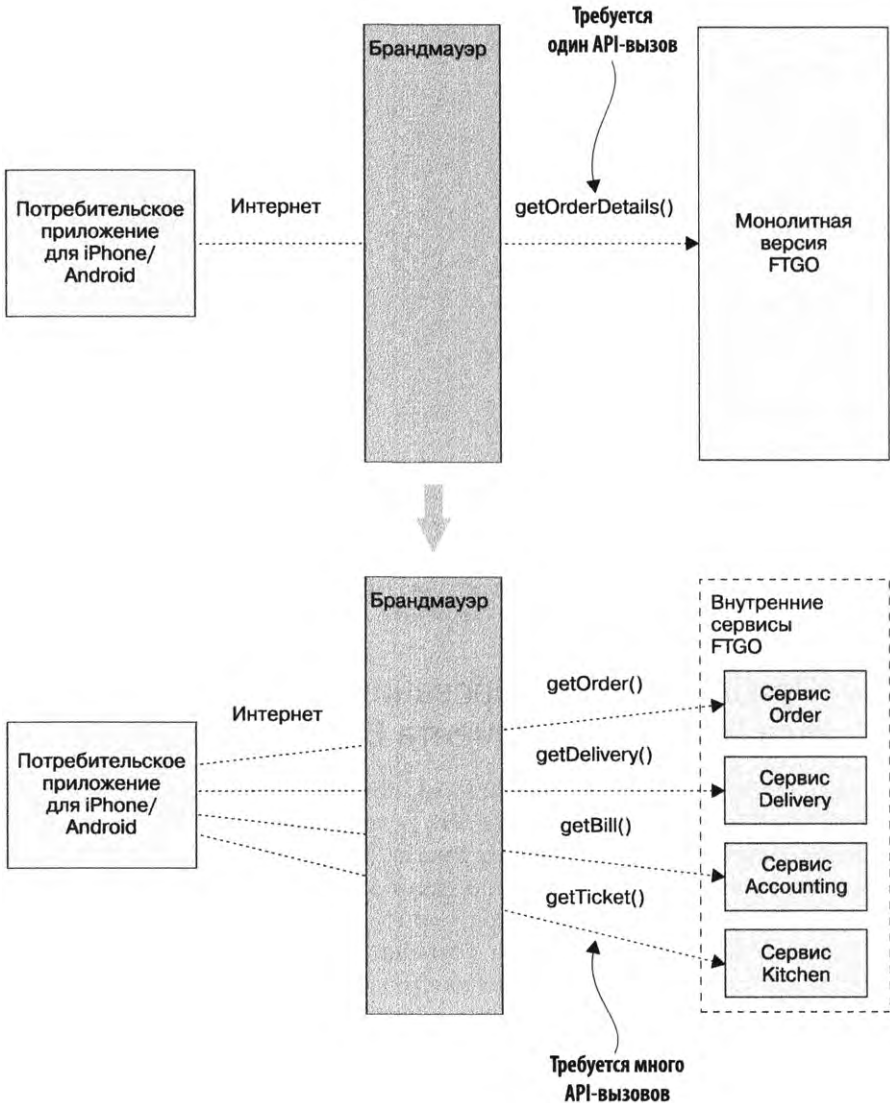


Рис. 8.2. Чтобы извлечь подробности о заказе в монолитной версии FTGO, клиенту достаточно одного запроса. Но для извлечения той же информации в микросервисной архитектуре потребуется несколько запросов

В этой архитектуре мобильное приложение играет роль API-композитора. Оно обращается к нескольким сервисам и объединяет результаты. Этот подход может показаться разумным, но у него есть несколько проблем.

Отрицательный опыт взаимодействия из-за того, что клиент выполняет несколько запросов

Первая проблема состоит в том, что мобильному приложению иногда придется выполнять несколько запросов для извлечения данных, которые нужно отобразить для пользователя. Слишком интенсивное взаимодействие между приложением и сервисами может плохо сказаться на отзывчивости приложения, особенно если оно проходит по Интернету или мобильной сети. Интернет обладает куда меньшей пропускной способностью и более высокой латентностью по сравнению с локальной сетью, а мобильные сети в этом отношении еще хуже. В обоих случаях уровень латентности обычно различается в 100 раз.

Повышенная латентность при извлечении деталей заказа не обязательно вызывает проблемы, так как мобильное приложение минимизирует задержки за счет конкурентного выполнения запросов. Общее время оказывается не больше, чем в случае с одним запросом. Но иногда запросы приходится выполнять последовательно, что уменьшает удобство использования клиента.

Однако негативный опыт взаимодействия из-за сетевых задержек не единственная проблема, присущая интенсивному обращению к API. Мобильным разработчикам иногда приходится писать довольно сложный код для объединения API. Это отвлекает их от создания удобных пользовательских интерфейсов — их основной задачи. Кроме того, на каждый сетевой запрос затрачивается электроэнергия, поэтому интенсивная работа с API быстрее сажает аккумулятор мобильного устройства.

Недостаточная инкапсуляция требует синхронного обновления кода на серверной и клиентской сторонах

Еще один недостаток того, что мобильное приложение напрямую обращается к сервисам, связано с недостаточной инкапсуляцией. По мере развития приложения разработчики сервисов иногда меняют API, нарушая работу существующих клиентов. Изменения могут касаться даже декомпозиции системы. Разработчики могут добавить новые сервисы или разделить/слить уже имеющиеся. И если сведения о сервисах вшиты в мобильное приложение, изменить их API может оказаться непросто.

В отличие от обновления серверных систем, выкатывание новой версии мобильного приложения может занять часы или даже дни. Такие компании, как Apple или Google, должны одобрить ваше обновление и сделать его доступным для загрузки. При этом не факт, что пользователи загрузят его сразу (или вообще когда-нибудь), и вряд ли стоит заставлять их это делать. Открытие API для мобильных устройств создает существенные преграды для его последующего развития.

Сервисы могут использовать механизмы IPC, плохо совместимые с клиентами

У непосредственного взаимодействия между мобильным приложением и сервисами есть еще один недостаток: иногда сервисы используют протоколы, с которыми клиентам сложно работать. Клиентские приложения, находящиеся за пределами брандмауэра, обычно применяют такие протоколы, как HTTP и WebSockets. Но как упоминалось в главе 3, у разработчиков сервисов есть большой выбор протоколов, помимо HTTP. Одни используют gRPC, другие — обмен сообщениями на основе AMQP. Такого рода протоколы хорошо работают внутри, но их совместимость с мобильными клиентами может быть оказаться под вопросом. Некоторые из них даже не умеют обходить брандмауэр.

8.1.2. Проблемы с проектированием API для клиентов другого рода

Я выбрал мобильные клиенты, потому что они отлично демонстрируют недостатки прямого доступа к сервисам. Но эти проблемы не ограничены лишь мобильными устройствами. Они распространяются и на клиенты другого рода, особенно те, что находятся за пределами брандмауэра. Как описывалось ранее, сервисы приложения FTGO потребляются веб-сайтами, программами, основанными на JavaScript, и сторонними системами. Рассмотрим, какие трудности могут возникнуть при проектировании API для таких клиентов.

Проблемы проектирования API для веб-приложений

Традиционные серверные веб-приложения, которые обрабатывают HTTP-запросы браузера и возвращают HTML-страницы, находятся в пределах брандмауэра и обращаются к сервисам по локальной сети. Таким образом, при объединении API пропускная способность и латентность сети не оказываются проблемой. Кроме того, для работы с сервисами веб-приложения могут использовать протоколы, плохо совместимые с веб-технологиями. Их разработчики часто тесно взаимодействуют с командами, отвечающими за серверную сторону, так как все они — части одной организации, поэтому веб-приложение можно легко обновлять при изменениях в сервисах. Из-за этого веб-приложениям стоит работать с сервисами напрямую.

Проблемы проектирования API для браузерных JavaScript-приложений

Современные приложения содержат некоторое количество кода на языке JavaScript. Даже если HTML в основном генерируется серверным веб-приложением, для доступа к сервисам часто используется код на JavaScript, выполняющийся в браузере. Например, все веб-приложения в проекте FTGO (Consumer, Restaurant и Admin) содержат JavaScript-код, который обращается к серверной стороне. Веб-приложение Consumer, например, динамически обновляет страницу Order Details, вызывая API сервисов с помощью JavaScript.

С одной стороны, JavaScript-приложения, запускаемые в браузере, можно легко обновлять при изменениях в API. Но с другой — если они обращаются к сервисам по Интернету, у них могут наблюдаться те же проблемы с сетевой латентностью, что и у мобильных клиентов. Что еще хуже, браузерные пользовательские интерфейсы, особенно предназначенные для настольных компьютеров, обычно сложнее мобильных приложений и должны объединять большее количество сервисов. Вполне вероятно, что приложения *Consumer* и *Restaurant*, которые вызывают сервисы по Интернету, не смогут эффективно объединять разные API.

Проектирование API для сторонних приложений

Компания FTGO, как и многие другие организации, предоставляет для сторонних разработчиков API, с помощью которого те могут создавать приложения для размещения и администрирования заказов. Эти сторонние программы работают через Интернет, поэтому объединение API, скорее всего, будет неэффективным. Но это относительно небольшая проблема по сравнению с трудностями при проектировании API для сторонних клиентов. Ведь сторонним разработчикам нужен стабильный интерфейс.

Очень немногие организации способны заставить сторонних разработчиков обновиться до новой версии API. Если API приложения становится нестабильным, сторонние клиенты могут перестать его поддерживать и перейти к конкурентам. В связи с этим вы должны тщательно продумывать развитие API, которые используются другими организациями. Обычно для этого приходится долго поддерживать старые версии или даже сохранять их навсегда.

Это требование ложится на организацию большим бременем. Возлагать ответственность за долгосрочную поддержку обратной совместимости на разработчиков сервисов было бы непрактично. Вместо того чтобы открывать сторонним клиентам прямой доступ к сервисам, организация должна иметь отдельный публичный API, который разрабатывает отдельная команда. Как вы позже увидите, публичные API реализуются архитектурным компонентом, известным как *API-шлюз*. Давайте посмотрим, как он работает.

8.2. Шаблон «API-шлюз»

Как вы только что узнали, прямой доступ к сервисам чреват целым рядом проблем. Зачастую выполнять объединение API по Интернету на стороне клиента непрактично. Недостаточная инкапсуляция затрудняет изменение декомпозиции сервисов и обновление API. Иногда сервисы применяют коммуникационные протоколы, которые плохо работают за пределами брандмауэра. Учитывая все это, гораздо лучше будет использовать API-шлюз.

Шаблон «API-шлюз»

Реализует сервис, который служит точкой входа в микросервисное приложение для клиентов внешнего API. См. microservices.io/patterns/apigateway.html.

API-шлюз — это сервис, который служит точкой входа в приложение из внешнего мира. Он отвечает за маршрутизацию запросов, объединение API и другие возможности, например аутентификацию. В данном разделе мы рассмотрим его со всеми преимуществами и недостатками. Вы познакомитесь с проблемами проектирования, которые необходимо решить при разработке API-шлюза.

8.2.1. Обзор шаблона «API-шлюз»

В разделе 8.1.1 описывались недостатки подхода, при котором клиенты (например, мобильное приложение FTGO) выполняют множество запросов, чтобы отобразить информацию для пользователя. Намного лучше было бы сделать так, чтобы клиент делал один запрос к API-шлюзу — сервису, который служит единой точкой входа для API-запросов к приложению из-за пределов брандмауэра. Это похоже на шаблон объектно-ориентированного проектирования «Фасад» в том смысле, что API-шлюз инкапсулирует внутреннюю архитектуру приложения и предоставляет API его клиентам. Он может иметь и другие функции, такие как аутентификация, мониторинг и ограничение частоты запросов. Отношения между клиентами, API-шлюзом и сервисами показаны на рис. 8.3.

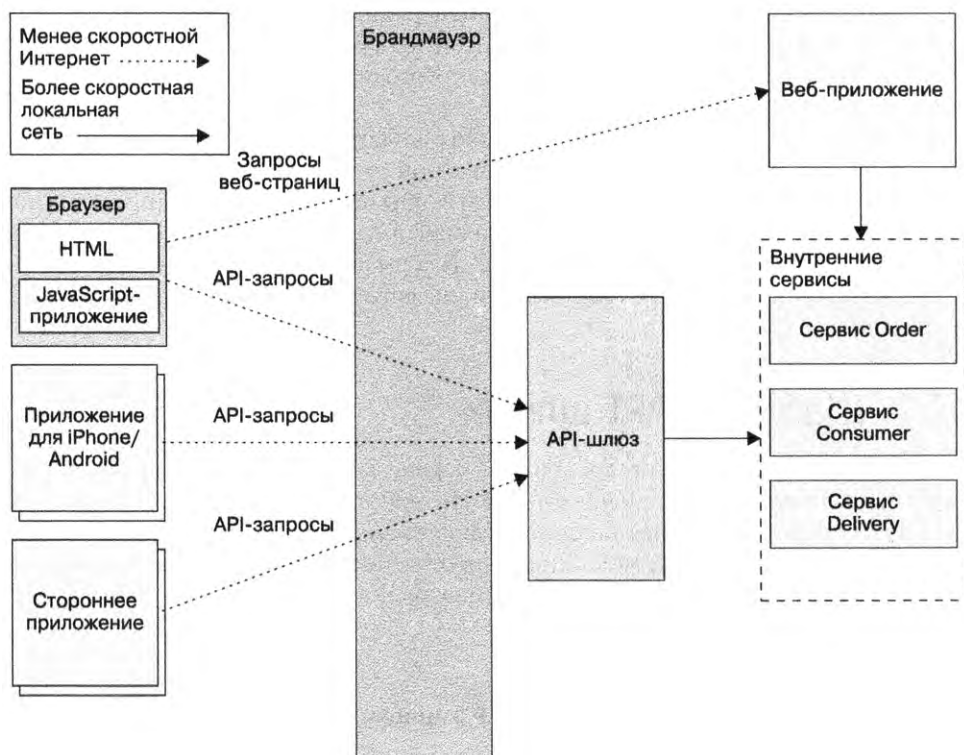


Рис. 8.3. API-шлюз — это единая точка входа в приложение для API-вызовов из-за пределов брандмауэра

API-шлюз отвечает за маршрутизацию запросов, объединение API и преобразование протоколов. Все запросы, выполняемые *внешними* клиентами, сначала поступают на API-шлюз, который направляет их подходящим сервисам. Для обработки других запросов API-шлюз использует объединение API, обращаясь к разным сервисам и агрегируя полученные результаты. Он может также налаживать связь между клиентскими протоколами, такими как HTTP и WebSockets, и внутренними протоколами сервисов, плохо совместимыми с клиентами.

Маршрутизация запросов

Одна из ключевых функций API-шлюза — *маршрутизация запросов*. Некоторые API-вызовы реализуются путем направления запросов подходящим сервисам. Когда API-шлюз получает запрос, он сверяется с картой маршрутизации, которая определяет, какому сервису следует направить этот запрос. Например, карта маршрутизации может привязывать HTTP-метод и путь к URL-адресу сервиса. Эта функция идентична возможностям обратного прокси, которые предоставляют такие серверы, как NGINX.

Объединение API

API-шлюз обычно не ограничивается обратным проксированием. Он может также реализовывать некоторые API-операции, используя объединение API. Например, в FTGO он таким образом выполняет операцию **Get Order Details**. Мобильное приложение делает один запрос к API-шлюзу, а тот извлекает подробности о заказе из нескольких сервисов (рис. 8.4).

API-шлюз приложения FTGO предоставляет обобщенный API, который позволяет клиентам извлекать нужные им данные с помощью лишь одного запроса. Например, мобильный клиент делает единственный запрос `getOrderDetails()`.

Преобразование протоколов

API-шлюз способен также преобразовывать протоколы. Для внешних клиентов он может предоставлять RESTful API, хотя внутри сервисы приложения используют сочетание разных протоколов, включая REST и gRPC. При необходимости реализация некоторых API-операций налаживает связь между внешним RESTful API и внутренними интерфейсами на основе gRPC.

API-шлюз предоставляет каждому клиенту отдельный API

API-шлюз мог бы предоставлять единый универсальный API. Но проблема такого подхода состоит в том, что у разных клиентов разные требования. Например, стороннему приложению может понадобиться, чтобы API-операция **Get Order Details** возвращала все подробности о заказе, тогда как мобильному клиенту нужна лишь часть из них. В качестве решения можно позволить клиентам указывать в запросе, какие поля и сопутствующие объекты им следует вернуть. Это подходит для публичных API, которые должны обслуживать широкий спектр сторонних приложений, притом что клиенты часто лишены необходимого им контроля.

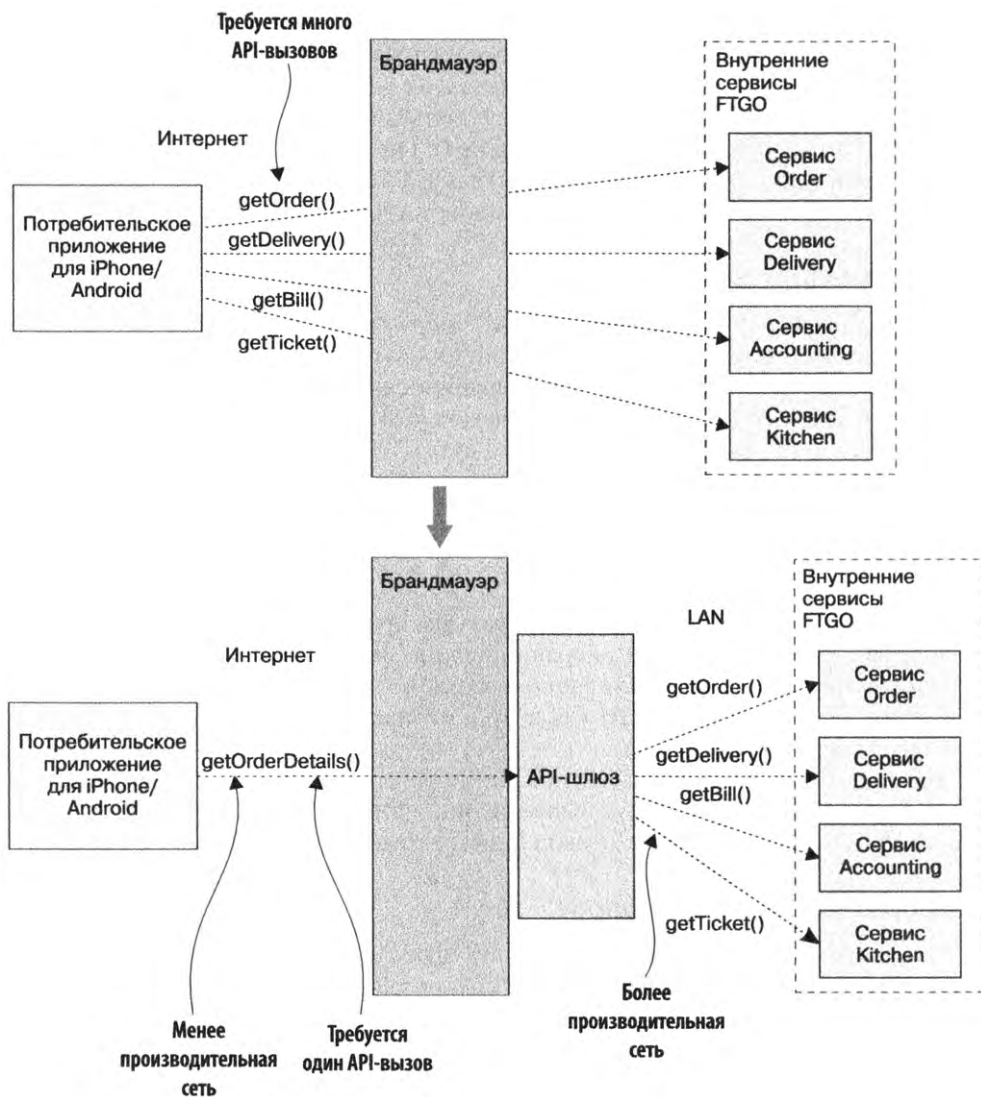


Рис. 8.4. API-шлюз часто выполняет объединение API, что позволяет таким клиентам, как мобильные устройства, эффективно извлекать данные с помощью одного API-запроса

Вместо этого лучше сделать так, чтобы API-шлюз выделял каждому клиенту отдельный API. Например, API-шлюз FTGO может предоставить мобильному клиенту API, специально «заточенный» под его требования. И даже поддерживать разные версии API для Android и iPhone. Отдельный публичный API будет предоставлен также сторонним разработчикам. Позже вы познакомитесь с шаблоном BFF, который выводит эту концепцию на новый уровень, определяя разные API-шлюзы для каждого клиента.

Реализация граничных функций

Основными обязанностями API-шлюза являются маршрутизация и объединение API, но он способен взять на себя также реализацию граничных функций. *Граничная функция*, как понятно из названия, — это операция обработки запросов на границе приложения. Можно привести следующие примеры:

- ❑ *аутентификация* — проверка подлинности клиента, который делает запрос;
- ❑ *авторизация* — проверка того, что клиенту позволено выполнять определенную операцию;
- ❑ *ограничение частоты запросов* — контроль над тем, сколько запросов в секунду могут выполнять определенный клиент и/или все клиенты вместе;
- ❑ *кэширование* — кэширование ответов для снижения количества запросов к сервисам;
- ❑ *сбор показателей* — сбор показателей использования API для анализа, связанного с биллингом;
- ❑ *ведение журнала запросов* — запись запросов в журнал.

В вашем приложении есть три участка, где можно реализовать граничные функции. Прежде всего это внутренние сервисы. Они подходят для реализации таких функций, как кэширование, сбор показателей и, возможно, авторизация. При этом, чтобы увеличить безопасность, аутентификацию запросов лучше выполнять до того, как они достигнут сервисов.

Второй участок — это реализация граничных функций в отдельном граничном сервисе, который находится сразу перед API-шлюзом. Этот сервис будет выступать первой точкой контакта с внешними клиентами. Он аутентифицирует запрос и выполняет другую граничную обработку, прежде чем передать его API-шлюзу.

Важное преимущество от использования отдельного граничного сервиса связано с разделением ответственности. API-шлюз может сосредоточиться на маршрутизации и объединении API. Еще одна положительная сторона состоит в централизации ответственности за критически важные функции, такие как аутентификация. Это особенно полезно в случае, когда у приложения есть несколько API-шлюзов, которые могут быть написаны с помощью разных языков и фреймворков. Мы поговорим об этом позже. Недостаток этого подхода — повышенная сетевая латентность из-за дополнительного сетевого перехода. К тому же это делает приложение более сложным.

В итоге во многих случаях лучше воспользоваться третьим вариантом и реализовать граничные функции, особенно аутентификацию, в самом API-шлюзе. Этим мы устраняем лишний сетевой переход, чем снижаем латентность. К тому же чем меньше компонентов, тем проще приложение. В главе 11 мы поговорим о том, как API-шлюз и сервисы обеспечивают безопасность, взаимодействуя друг с другом.

Архитектура API-шлюза

API-шлюз имеет двухуровневую модульную архитектуру. Она состоит из двух частей: общего уровня и API (рис. 8.5). API состоит из одного или нескольких API-модулей. Каждый модуль реализует API для конкретного клиента. На общем уровне реализованы общие возможности, включая такие граничные функции, как аутентификация.

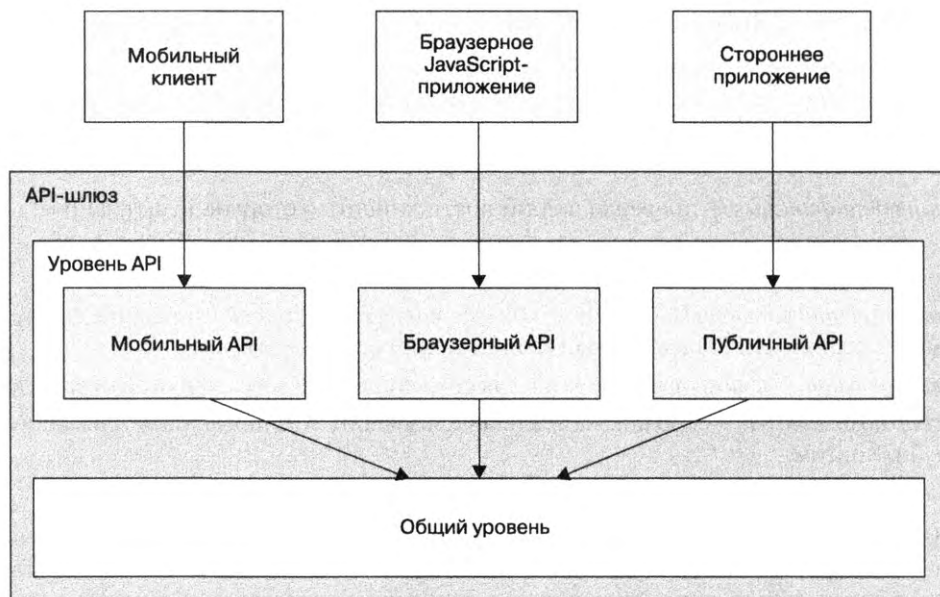


Рис. 8.5. Двухуровневая архитектура API-шлюза. API для каждого клиента реализуется отдельным модулем. Общий уровень реализует возможности, которые используются во всех API, например аутентификацию

В этом примере API-шлюз содержит три API-модуля:

- ❑ *мобильный API* — реализует API для мобильного клиента FTGO;
- ❑ *браузерный API* — реализует API для JavaScript-приложения, которое работает в браузере;
- ❑ *публичный API* — реализует API для сторонних разработчиков.

API-модуль реализует каждую API-операцию одним из двух способов. Некоторые операции накладываются непосредственно на определенную API-операцию сервиса, которой впоследствии направляются соответствующие запросы. Перенаправляются запросы с помощью универсального модуля, который считывает конфигурационный файл с правилами маршрутизации.

Чтобы реализовать более сложные API-операции, API-модуль использует объединение API. Это требует написания дополнительного кода. Каждая реализация API-операции обрабатывает запросы, обращаясь к нескольким сервисам и объединяя результаты.

Модель владения в API-шлюзе

Есть еще один важный вопрос, на который вы должны ответить: кто несет ответственность за разработку API-шлюза и его операций? Есть несколько вариантов. Вы можете выделить для этого отдельную команду. Недостаток такого решения в том, что оно похоже на архитектуру SOA, где одна команда отвечает за разработку

всей сервисной шины предприятия (Enterprise Service Bus, ESB). Если разработчику, который пишет мобильное приложение, нужно получить доступ к определенному сервису, он должен обратиться к коллегам из команды API-шлюза и подождать, когда они предоставят нужный API. Такое централизованное узкое место в организации противоречит философии микросервисной архитектуры, которая поощряет формирование слабо связанных автономных команд.

Вместо этого стоит воспользоваться подходом, который продвигает компания Netflix: отдать API-модуль на откуп клиентским разработчикам, занимающимся мобильными устройствами, веб-приложениями и публичным API. Команда, которая пишет API-шлюз, отвечает за его эксплуатацию и разработку модуля `Common`. Такая модель владения дает командам разработчиков контроль над их API (рис. 8.6).

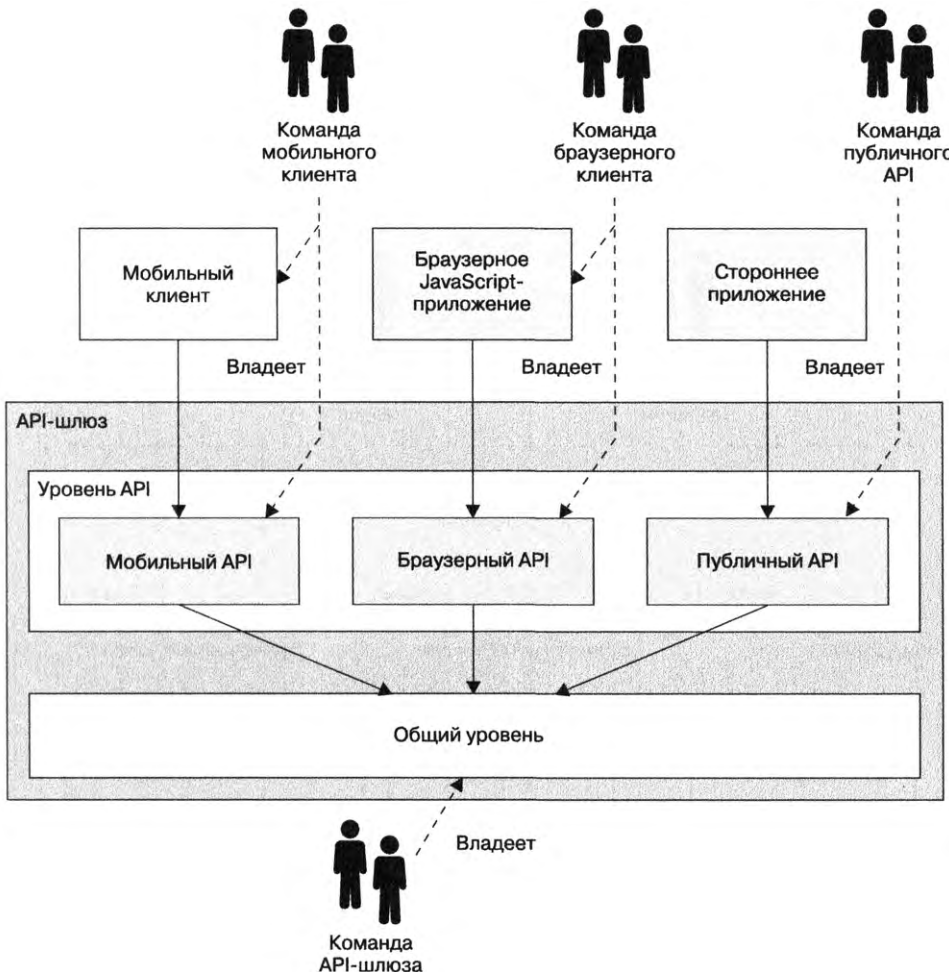


Рис. 8.6. Клиентская команда владеет своим API-модулем. По мере обновления клиента она может изменять API-модуль, не спрашивая разрешения у команды API-шлюза

Когда команде нужно обновить свой API, она вносит изменения в исходный репозиторий API-шлюза. Чтобы сделать процесс разработки API-шлюза надежным, его следует полностью автоматизировать. В противном случае клиентским командам придется часто останавливать свою работу и ждать, пока команда API-шлюза не развернет новую версию.

Использование шаблона BFF

Одна из проблем API-шлюза состоит в нечетком разделении ответственности. Разные команды вносят свой вклад в общую кодовую базу. Команда API-шлюза отвечает за его эксплуатацию. Это не настолько плохо, как в SOA ESB, но размывание ответственности противоречит одному из принципов микросервисной архитектуры: «за компонент отвечает тот, кто его пишет».

Решением этой проблемы является выделение API-шлюза для каждого клиента. Это шаблон проектирования BFF (backends for frontends — серверы для клиентов), который был предложен Филом Кальсадо (Phil Calçado) (philcalzado.com) и его коллегами из SoundCloud. На рис. 8.7 показано, как каждый API-модуль превращается в отдельный API-шлюз, который разрабатывается и администрируется одной клиентской командой.

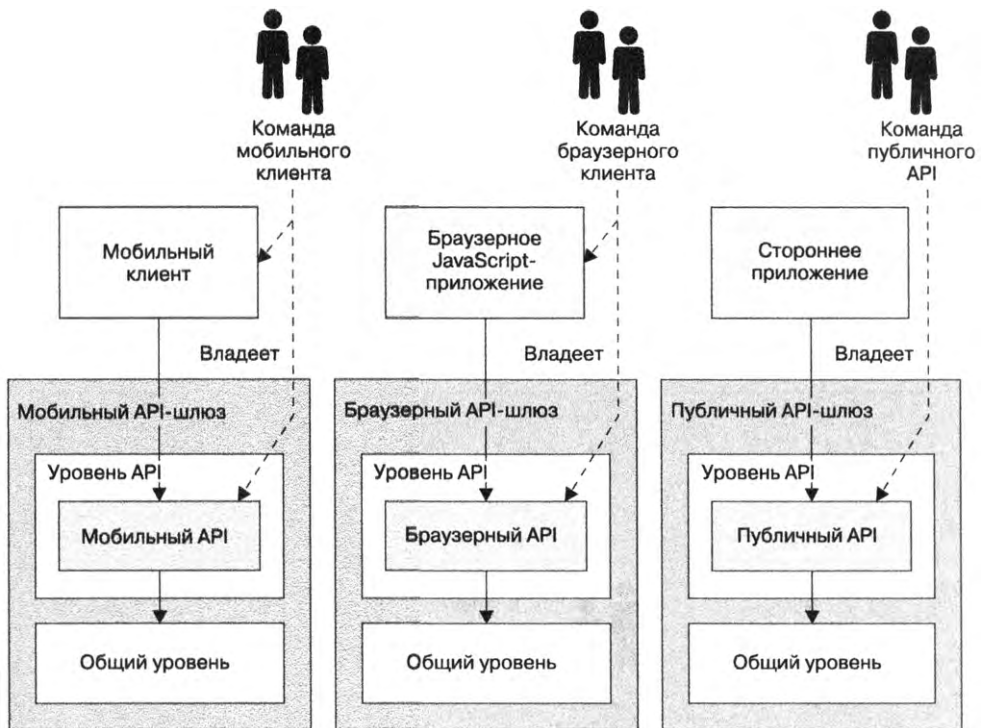


Рис. 8.7. Шаблон BFF отводит каждому клиенту отдельный API-шлюз. Каждая клиентская команда имеет собственный API-шлюз. Команда API-шлюза отвечает за общий уровень

Шаблон BFF

Реализует отдельный API-шлюз для каждого типа клиентов. См. microservices.io/patterns/apigateway.html.

Команда публичного API владеет своим API-шлюзом и отвечает за его администрирование, у мобильной команды — свой API-шлюз, администрированием которого она занимается, и т. д. Теоретически разные API-шлюзы можно разрабатывать с помощью разных наборов технологий. Но это грозит вылиться в дублирование общей функциональности, такой как реализация граничных функций. В идеале все API-шлюзы должны использовать единый технологический стек. Команда API-шлюза выносит общие возможности в разделяемую библиотеку.

Помимо четкого разделения ответственности, шаблон BFF имеет и другие преимущества. API-модули изолированы друг от друга, что повышает надежность. Одному плохо работающему API будет сложно повлиять на другие. Это улучшает наблюдаемость, поскольку API-модули выполняются в разных процессах. Еще одна положительная сторона шаблона BFF — независимое масштабирование каждого API. Он также уменьшает время запуска, из-за того что каждый отдельный API-шлюз становится более компактным и простым.

8.2.2. Преимущества и недостатки API-шлюза

Как вы, наверное, и ожидали, шаблон API-шлюза имеет как положительные, так и отрицательные стороны.

Преимущества API-шлюза

Большое преимущество от использования API-шлюза связано с тем, что он инкапсулирует внутреннюю структуру приложения. Вместо вызова определенных сервисов клиенты общаются со шлюзом. Каждый клиент получает отдельный API, что снижает количество запросов между ним и приложением. К тому же это упрощает клиентский код.

Недостатки API-шлюза

Шаблон API-шлюза имеет определенные недостатки. Это еще один высокодоступный компонент, который нужно разрабатывать, развертывать и администрировать. Вдобавок существует риск того, что API-шлюз начнет тормозить разработку. Его следует обновлять при «выставлении наружу» API очередного сервиса. Важно, чтобы процесс обновления был максимально легковесным. В противном случае разработчики будут вынуждены ждать своей очереди, чтобы обновить API-шлюз. Несмотря на это, шаблон BFF подходит для большинства реальных приложений. При необходимости можно применить его, чтобы команды разрабатывали и развертывали свои API независимо друг от друга.

8.2.3. Netflix как пример использования API-шлюза

Отличным примером API-шлюза является Netflix API. Потокное видео Netflix доступно на сотнях разных устройств, включая телевизоры, проигрыватели Blu-ray, смартфоны и многое другое. Изначально компания Netflix пыталась обойтись единым универсальным API (www.programmableweb.com/news/why-rest-keeps-me-night/2012/05/15). Но вскоре обнаружилось, что это не самая лучшая идея, поскольку разные устройства предъявляют разные требования. Сейчас Netflix использует API-шлюз, который реализует отдельные API для каждого типа устройств, за которые отвечают команды клиентских разработчиков.

В первой версии API-шлюза каждая клиентская команда реализовывала свой API с помощью скриптов на языке Groovy, которые выполняли маршрутизацию и объединение API. Каждый скрипт обращался к API одного или нескольких сервисов, используя клиентские библиотеки на языке Java, предоставляемые командами сервисов. В общем-то, все хорошо работало, а клиентские разработчики написали тысячи скриптов. Каждый день API-шлюз обрабатывал миллиарды запросов, а каждый API-вызов обращался в среднем к шести или семи внутренним сервисам. Однако компании Netflix такая монолитная архитектура показалась несколько громоздкой.

Так что сейчас Netflix переходит на архитектуру API-шлюза, похожую на шаблон BFF. Теперь клиентские команды пишут API-модули с помощью NodeJS. Каждый API-модуль запускает собственный Docker-контейнер, но скрипты не вызывают сервисы напрямую. Вместо этого они обращаются ко второму «API-шлюзу», который делает доступными API сервисов с использованием Netflix Falcor. *Netflix Falcor* — это технология для декларативного динамического объединения API, которая позволяет клиенту вызвать несколько сервисов за один запрос. Эта новая архитектура обладает рядом преимуществ. API-модули изолированы друг от друга, что улучшает надежность и наблюдаемость. При этом клиентские API-модули поддерживают независимое масштабирование.

8.2.4. Трудности проектирования API-шлюза

Итак, вы познакомились с шаблоном API-шлюза, его преимуществами и недостатками. Теперь исследуем трудности, которые могут возникнуть при его проектировании. Есть несколько моментов, на которые следует обратить внимание.

- ☐ Производительность и масштабируемость.
- ☐ Написание поддерживаемого кода с помощью абстракций реактивного программирования.
- ☐ Обработка частичных отказов.
- ☐ Реализация шаблонов, общих для архитектуры приложения.

Обсудим их.

Производительность и масштабируемость

API-шлюз — это парадный вход в приложение. Через него должны поступать все внешние запросы. И хотя большинству компаний далеко до Netflix, где ежедневные запросы исчисляются миллиардами, производительность и масштабируемость API-шлюза обычно имеют большое значение. Ключевое архитектурное решение, которое влияет на эти показатели, заключается в выборе между синхронным и асинхронным вводом/выводом.

В *синхронной* модели ввода/вывода каждое сетевое соединение обрабатывается отдельным потоком. Это простая, неплохо работающая программная модель. Например, она лежит в основе широко используемого фреймворка сервлетов Java EE (правда, в нем есть возможность завершать запросы асинхронно). Однако ограничение синхронного ввода/вывода связано с тяжеловесностью потоков операционной системы, в результате ограничивается количество потоков, а вместе с ним и число параллельных соединений, которые поддерживает API-шлюз.

Альтернативный подход состоит в применении *асинхронной* (неблокирующей) модели ввода/вывода. В ней передачей запросов обработчикам событий занимается один поток с рабочим циклом. Асинхронный ввод/вывод реализован в различных технологиях. В JVM можно задействовать один из фреймворков на основе NIO, включая Netty, Vertx, Spring Reactor и JBoss Undertow. За пределами JVM популярным выбором является NodeJS — платформа, построенная поверх движка JavaScript из браузера Chrome.

Неблокирующий ввод/вывод гораздо лучше масштабируется, потому что не тратит лишние ресурсы на создание множества потоков. Его слабая сторона заключается в намного более сложной модели программирования, основанной на функциях обратного вызова. Это усложняет написание, чтение и отладку кода. Обработчики событий должны быстро возвращать результат, чтобы не блокировать рабочий цикл потока.

Кроме того, окажет ли неблокирующий ввод/вывод общий положительный эффект, зависит от характеристик логики обработки запросов API-шлюза. У Netflix получились неоднозначные результаты после переписывания Zuul — граничного сервера с применением NIO (см. medium.com/netflixtechblog/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c). С одной стороны, как можно было ожидать, это снизило расходы на каждое сетевое подключение: для этого больше не нужно создавать отдельный поток. К тому же при выполнении логики с интенсивным использованием ввода/вывода, такой как маршрутизация запросов, кластер Zuul продемонстрировал увеличение пропускной способности и снижение вычислительной нагрузки на 25 %. С другой стороны, при интенсивной работе с вычислительными операциями, такими как расшифровка и сжатие, кластер Zuul не показал никаких улучшений.

Использование абстракций реактивного программирования

Как уже упоминалось, объединение API подразумевает вызов нескольких внутренних сервисов. Некоторые из этих вызовов полностью зависят от параметров клиентского запроса. Другие могут полагаться на результаты запросов к другим сервисам.

Одно из решений состоит в том, чтобы обработчик конечной точки API вызывал сервисы в порядке, основанном на зависимостях. Например, в листинге 8.1 показан написанный таким образом обработчик запроса `findOrder()`. Он последовательно, один за другим вызывает каждый из четырех сервисов.

Листинг 8.1. Извлечение подробностей о заказе путем последовательного обращения к внутренним сервисам

```
@RestController
public class OrderDetailsController {
    @RequestMapping("/order/{orderId}")
    public OrderDetails getOrderDetails(@PathVariable String orderId) {

        OrderInfo orderInfo = orderService.findOrderById(orderId);

        TicketInfo ticketInfo = kitchenService
            .findTicketByOrderId(orderId);

        DeliveryInfo deliveryInfo = deliveryService
            .findDeliveryByOrderId(orderId);

        BillInfo billInfo = accountingService
            .findBillByOrderId(orderId);

        OrderDetails orderDetails =
            OrderDetails.makeOrderDetails(orderInfo, ticketInfo,
                deliveryInfo, billInfo);

        return orderDetails;
    }
    ...
}
```

Недостаток этого подхода состоит в том, что времена ответа каждого из сервисов суммируются в общее время ответа. Чтобы его минимизировать, логика объединения API должна по возможности обращаться к сервисам параллельно. В этом примере между вызовами нет никаких зависимостей, поэтому все они должны быть выполнены в конкурентном стиле. Это существенно снизит время ожидания. Основная трудность заключается в написании конкурентного кода, который будет несложно поддерживать.

Дело в том, что традиционно для написания масштабируемого конкурентного кода используются функции обратного вызова. Это в наибольшей степени относится к асинхронному событийному вводу/выводу. Функции обратного вызова обычно применяются даже в API-композиторе на основе сервлетов, который вызывает сервисы параллельно. Для конкурентного выполнения запросов он может вызывать метод `ExecutorService.submitCallable()`. Но проблема в том, что этот метод возвращает объект `Future`, который имеет блокирующий API. Чтобы добиться лучшей масштабируемости, API-композитор мог бы воспользоваться методом `ExecutorService.submit(Runnable)` и затем передать результат каждого запроса в функцию обратного вызова. Когда все результаты будут получены, она возвратит ответ клиенту.

Написание кода для объединения API с применением традиционного асинхронного подхода очень быстро приводит к ситуации, которую называют *адом обратных вызовов* (callback hell). Код становится запутанным, сложным для понимания и начинает провоцировать ошибки, особенно когда для объединения требуется сочетание параллельных и последовательных запросов. Намного более удачным подходом является использование декларативного стиля с реактивными методиками. В качестве примера реактивных абстракций в JVM можно привести следующие:

- ❑ `CompletableFuture` в Java 8;
- ❑ `Monos` из Project Reactor;
- ❑ `Observable` из библиотеки RxJava (Reactive Extensions for Java — реактивные расширения для Java), созданной компанией Netflix специально для решения этой проблемы в ее API-шлюзе;
- ❑ `Future` в Scala.

API-шлюз, основанный на NodeJS, может использовать встроенные объекты `Promise` или реактивные расширения для JavaScript, `RxJS`. Любая из этих двух абстракций позволит вам писать конкурентный код, который потом легко будет понять. Позже в этой главе я продемонстрирую этот стиль программирования на примере класса `Monos` из Project Reactor и пятой версии Spring Framework.

Обработка частичных отказов

API-шлюз должен быть не только масштабируемым, но и надежным. Чтобы этого добиться, его можно запускать в нескольких экземплярах, размещенных за балансировщиком нагрузки. Если один из них откажет, балансировщик перенаправит запросы на другие экземпляры.

Еще один способ обеспечения надежности API-шлюза заключается в правильной обработке запросов, которые завершились неудачей или имеют слишком высокую латентность. Когда API-шлюз обращается к сервису, всегда существует вероятность того, что тот окажется медленным или недоступным. Иногда ответа приходится ждать очень долго, даже бесконечно, что отнимает ресурсы и не дает ответить клиенту. Затянувшийся запрос к неисправному сервису может даже расходовать такой ограниченный и ценный ресурс, как системные потоки, из-за чего API-шлюз будет не в состоянии обработать другие запросы. Решение этой проблемы описано в главе 3: при вызове сервисов API-шлюз должен использовать шаблон «Предохранитель».

Реализация шаблонов, общих для архитектуры приложения

В главе 3 я описал шаблоны для обнаружения сервисов, а в главе 11 вы познакомитесь с шаблонами для обеспечения наблюдаемости. Шаблоны обнаружения сервисов позволяют клиенту, например API-шлюзу, определить сетевое местоположение экземпляра сервиса, чтобы обратиться к нему. Шаблоны обеспечения наблюдаемости позволяют разработчикам отслеживать поведение приложения и диагностировать возникающие проблемы. API-шлюз, как и другие сервисы, должен реализовать шаблоны, выбранные для заданной архитектуры.

8.3. Реализация API-шлюза

Теперь посмотрим, как реализовать API-шлюз. Ранее уже упоминалось, что у него есть следующие обязанности:

- ❑ *маршрутизация запросов* — маршрутизация запросов к сервисам на основе таких критериев, как HTTP-метод или путь. Если приложение содержит несколько сервисов для CORS-запросов, API-шлюз должен учитывать HTTP-метод при маршрутизации. Как упоминалось в главе 7, в такой архитектуре команды и запросы обрабатываются разными сервисами;
- ❑ *объединение API* — реализация конечной точки REST с методом GET с помощью объединения API (см. главу 7). Обработчик запросов объединяет результаты вызова нескольких сервисов;
- ❑ *границные функции* — самой примечательной из них является аутентификация;
- ❑ *преобразование протоколов* — взаимодействие между протоколами клиентской стороны и теми, которые используются сервисами и плохо совместимы с клиентами;
- ❑ реализация шаблонов, общих для архитектуры приложения.

API-шлюз можно реализовать несколькими способами:

- ❑ *используя готовый продукт/сервис*. Этот вариант почти (или совсем) не требует разработки, но он менее гибок. Например, готовые API-шлюзы обычно не поддерживают объединение API;
- ❑ путем разработки собственного API-шлюза с применением специального (веб-) фреймворка в качестве отправной точки. Это самый гибкий подход, но он требует от разработчиков определенных усилий.

Рассмотрим оба варианта.

8.3.1. Использование готового API-шлюза

Функции API-шлюза реализованы в нескольких готовых сервисах и продуктах. Вначале обсудим сервисы, предоставляемые в рамках AWS. Затем поговорим о нескольких продуктах, которые вы можете загрузить, сконфигурировать и применять самостоятельно.

AWS API Gateway

AWS API Gateway — это один из множества сервисов, предоставляемый компанией, он предназначен для развертывания и администрирования API. Его API представляет собой набор ресурсов REST, каждый из которых поддерживает один или несколько HTTP-методов. В его конфигурации нужно указать, к какому внутреннему сервису следует направлять каждую пару «метод — ресурс». В качестве внутреннего сервиса выступает либо лямбда-функция AWS (см. главу 12), либо HTTP-сервис, определенный на уровне приложения, либо AWS-сервис. При необходимости API можно скон-

фигурировать так, чтобы он преобразовывал запросы и ответы, используя механизм шаблонов. AWS API Gateway умеет также аутентифицировать запросы.

AWS API Gateway удовлетворяет некоторым требованиям к API-шлюзу, описанным ранее. Это облачный сервис AWS, поэтому вам не нужно беспокоиться о его установке и администрировании — достаточно его сконфигурировать, а AWS позаботится обо всем остальном, включая масштабирование.

К сожалению, у AWS API Gateway есть несколько недостатков и ограничений, которые делают невозможным выполнение остальных требований. Он не поддерживает объединение API, поэтому вам придется реализовать эту возможность на уровне внутренних сервисов. Он поддерживает только HTTP(S) с большим упором на JSON и обнаружение на стороне сервера, описанное в главе 3. Приложения обычно используют Elastic Load Balancer для распределения запросов между серверами EC2 или контейнерами ECS. Но, несмотря на эти ограничения, если вы способны обойтись без объединения API, AWS API Gateway может послужить хорошей реализацией API-шлюза.

AWS Application Load Balancer

У компании Amazon есть еще один сервис, похожий на API-шлюз. Речь идет о балансировщике нагрузки AWS Application, который поддерживает HTTP, HTTPS, WebSocket и HTTP/2 (aws.amazon.com/blogs/aws/new-aws-application-load-balancer/). Чтобы его сконфигурировать, нужно определить правила маршрутизации, которые будут направлять запросы к внутренним сервисам (в данном случае это серверы AWS EC2).

Как и AWS API Gateway, AWS Application Load Balancer удовлетворяет некоторым требованиям, предъявляемым к API-шлюзу. Он реализует базовую функциональность для маршрутизации. Он находится в облаке, поэтому вам не нужно беспокоиться о его установке и администрировании. К сожалению, он довольно ограничен: не поддерживает маршрутизацию на основе HTTP-методов, объединение API и аутентификацию. В связи с этим AWS Application Load Balancer не подходит на роль API-шлюза.

Готовый продукт в качестве API-шлюза

Еще один вариант — применение готовых продуктов, таких как Kong или Traefik. Это пакеты с открытым исходным кодом, которые можно устанавливать и использовать самостоятельно. Kong основан на HTTP-сервере NGINX, а Traefik написан на языке GoLang. Оба продукта позволяют настраивать гибкие правила маршрутизации для выбора внутренних сервисов с учетом HTTP-методов, заголовков и путей. Kong предоставляет расширения для реализации таких граничных функций, как аутентификация. Traefik может даже интегрироваться с некоторыми реестрами сервисов, описанными в главе 3.

Несмотря на поддержку граничных функций и гибкой маршрутизации, эти продукты имеют определенные недостатки. На вас ложатся их установка, настройка и администрирование. К тому же они не поддерживают объединение API, поэтому, если вам необходима эта возможность, придется разработать собственный API-шлюз.

8.3.2. Разработка собственного API-шлюза

В разработке API-шлюза ничего особо сложного. Это, в сущности, веб-приложение, которое проксирует запросы к другим сервисам. Вы можете создать его сами, используя любимый веб-фреймворк. Однако при проектировании API-шлюза вам придется решить две ключевые проблемы:

- ❑ реализовать механизм определения правил маршрутизации, чтобы минимизировать написание сложного кода;
- ❑ правильно реализовать HTTP-проксирование, в том числе обработку HTTP-заголовков.

Таким образом, разработку API-шлюза лучше всего начать с выбора фреймворка, предназначенного для этой задачи. Его встроенные возможности значительно снизят объем кода, который вам придется писать.

Вначале мы рассмотрим проект с открытым исходным кодом Zuul от компании Netflix, а затем познакомимся с открытым проектом Spring Cloud Gateway от Pivotal.

Использование Netflix Zuul

Для реализации таких граничных функций, как маршрутизация, ограничение частоты запросов и аутентификация, компания Netflix разработала фреймворк Zuul (github.com/Netflix/zuul). В нем применяется концепция *фильтров* — перехватчиков запросов с возможностью повторного использования, похожих на фильтры сервлетов или промежуточный слой в NodeJS Express. Для обработки HTTP-запроса Zuul собирает цепочку подходящих фильтров, которые выполняют преобразование, вызывают внутренние сервисы и форматируют ответ, перед тем как вернуть его клиенту. Zuul можно применять напрямую, однако открытый проект Spring Cloud Zuul от компании Pivotal намного упрощает задачу. Этот проект основан на Zuul, но благодаря своей концепции *соглашения по конфигурации* он заметно облегчает разработку Zuul-серверов.

Zuul поддерживает маршрутизацию и граничные функции. Для его расширения можно использовать контроллеры Spring MVC, которые реализуют объединение API. Тем не менее Zuul реализует лишь маршрутизацию на основе путей, и это его основное ограничение. Например, он не может направить запрос `GET /orders` к одному сервису, а `POST /orders` — к другому. По этой причине Zuul не поддерживает архитектуру запросов, описанную в главе 7.

О проекте Spring Cloud Gateway

Ни один из описанных до сих пор вариантов не отвечает всем требованиям. На самом деле я уже было сдался и прекратил поиски подходящего фреймворка, начав разработку API-шлюза с помощью Spring MVC. Но затем открыл для себя проект Spring Cloud Gateway (cloud.spring.io/spring-cloud-gateway/). Он построен на основе нескольких фреймворков — Spring Framework 5, Spring Boot 2 и Spring Webflux. Последний вхо-

дит в состав Spring Framework 5, построен поверх Project Reactor и предоставляет реактивные абстракции. Project Reactor — это реактивный фреймворк для JVM на основе NIO, он реализует абстракцию *Mono*, которой мы воспользуемся позже в этой главе.

Spring Cloud Gateway предоставляет простой, но в то же время комплексный подход к решению следующих задач:

- ❑ направление запросов к внутренним сервисам;
- ❑ реализация обработчиков запросов, которые выполняют объединение API;
- ❑ реализация граничных функций, таких как аутентификация.

Ключевые компоненты API-шлюза, построенного с использованием этого фреймворка, представлены на рис. 8.8.

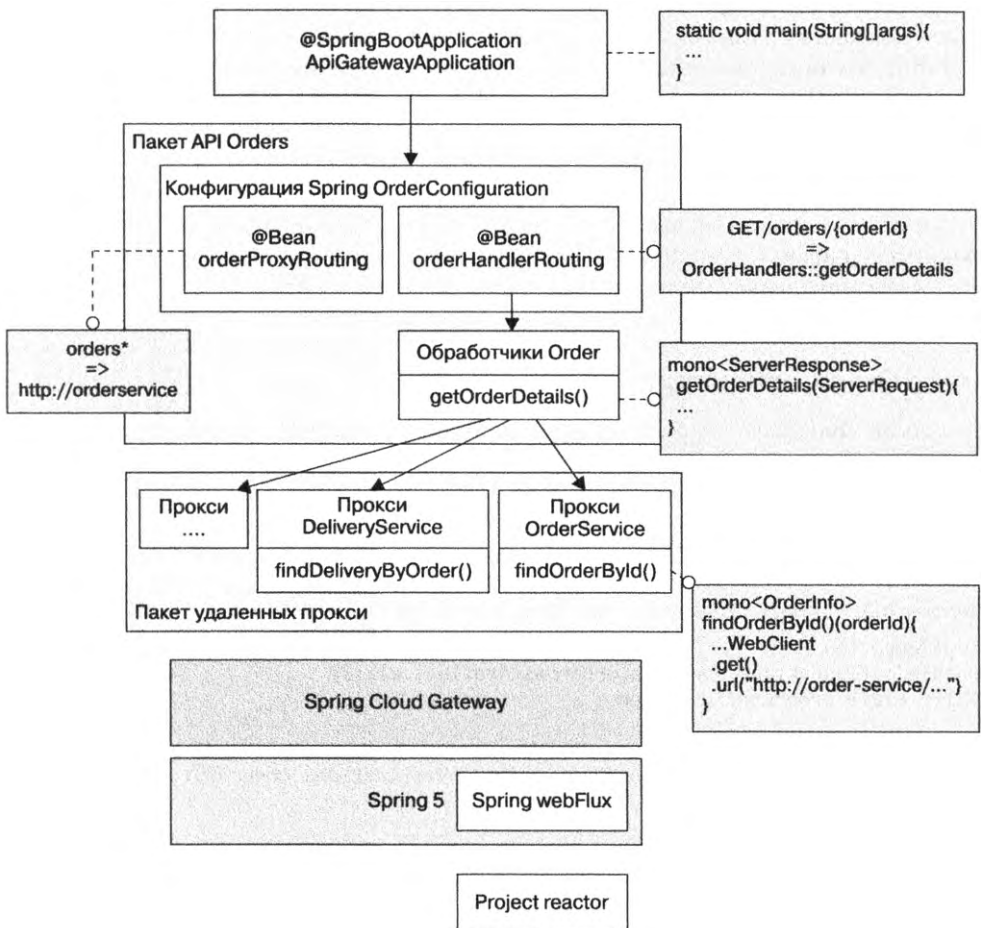


Рис. 8.8. Архитектура API-шлюза, созданного с помощью Spring Cloud Gateway

API-шлюз состоит из следующих пакетов.

- ❑ *Пакет `ApiGatewayMain`* — определяет главную программу для API-шлюза.
- ❑ *Один или несколько API-пакетов* — API-пакет реализует набор конечных точек API. Например, пакет `Order` реализует конечные точки, связанные с заказами.
- ❑ *Прокси-пакеты* — состоит из прокси-классов, с помощью которых API-пакеты обращаются к сервисам.

Класс `OrderConfiguration` определяет объекты Spring Bean, ответственные за маршрутизацию запросов, относящихся к заказам. Правило маршрутизации может соответствовать комбинации HTTP-метода, заголовков и пути. `orderProxyRoutes @Bean` определяет правила, которые связывают API-операции с URL-адресами внутреннего сервиса. Например, этот объект направляет все пути, начинающиеся с `/orders`, к сервису `Order`.

Правила `orderHandlers @Bean` переопределяют те, что были определены объектом `orderProxyRoutes`. Они связывают API-операции с методами-обработчиками Spring WebFlux, которые являются эквивалентом контроллеров в Spring MVC. Например, `orderHandlers` привязывает запрос `GET /orders/{orderId}` к методу `OrderHandlers::getOrderDetails()`.

Класс `OrderHandlers` реализует различные методы для обработки запросов, такие как `OrderHandlers::getOrderDetails()`. Этот метод извлекает подробности о заказе за счет объединения API, как было описано ранее. Обработчики обращаются к внутренним сервисам с помощью удаленных прокси-классов, таких как `OrderService`. Этот класс определяет методы для обращения к сервису `Order`.

Рассмотрим код, начиная с класса `OrderConfiguration`.

Класс `OrderConfiguration`

Класс `OrderConfiguration`, показанный в листинге 8.2, помечен аннотацией `@Configuration` из состава Spring. Он определяет объекты `@Bean`, которые реализуют конечные точки `/orders`. Такие объекты `@Bean`, как `orderProxyRouting` и `orderHandlerRouting`, описывают маршрутизацию запросов с помощью DSL-языка Spring WebFlux. `orderHandlers @Bean` реализует обработчики запросов, выполняющие объединение API.

Листинг 8.2. Объекты `@Bean` из состава Spring, которые реализуют конечные точки `/orders`

```
@Configuration
@EnableConfigurationProperties(OrderDestinations.class)
public class OrderConfiguration {

    @Bean
    public RouterLocator orderProxyRouting(OrderDestinations orderDestinations) {
        return Routes.locator()
            .route("orders")
            .uri(orderDestinations.orderServiceUrl)
            .predicate(path("/orders").or(path("/orders/*")))
            .and()
            ...
            .build();
    }
}
```

По умолчанию направляет все запросы,
начинающиеся с `/orders`,
на URL-адрес `orderDestinations.orderServiceUrl`

```

@Bean
public RouterFunction<ServerResponse>
    orderHandlerRouting(OrderHandlers orderHandlers) {
    return RouterFunctions.route(GET("/orders/{orderId}"),
                                orderHandlers::getOrderDetails);
}

@Bean
public OrderHandlers orderHandlers(OrderService orderService,
                                   KitchenService kitchenService,
                                   DeliveryService deliveryService,
                                   AccountingService accountingService) {
    return new OrderHandlers(orderService, kitchenService,
                             deliveryService, accountingService);
}

```

Направляет GET /orders/{orderId} к orderHandlers::getOrderDetails

Объект @Bean, реализующий пользовательскую логику обработки запросов

Класс `OrderDestinations`, представленный в листинге 8.3, помечен аннотацией `@ConfigurationProperties` из состава Spring, что позволяет использовать внешнюю конфигурацию URL-адресов внутреннего сервиса.

Листинг 8.3. Внешняя конфигурация URL-адресов внутреннего сервиса

```

@ConfigurationProperties(prefix = "order.destinations")
public class OrderDestinations {

    @NotNull
    public String orderServiceUrl;

    public String getOrderServiceUrl() {
        return orderServiceUrl;
    }

    public void setOrderServiceUrl(String orderServiceUrl) {
        this.orderServiceUrl = orderServiceUrl;
    }
    ...
}

```

К примеру, URL-адрес сервиса `Order` можно указать либо в виде свойства `order.destinations.orderServiceUrl` в конфигурационном файле, либо как переменную окружения `ORDER_DESTINATIONS_ORDER_SERVICE_URL`.

Класс `OrderHandlers`

Класс `OrderHandlers`, показанный в листинге 8.4, определяет методы для обработки запросов, которые реализуют нестандартное поведение, включая объединение API. Метод `getOrderDetails()`, к примеру, объединяет API для извлечения информации о заказе. В этот класс внедряется несколько прокси-классов, которые отправляют запросы к внутренним сервисам.

Листинг 8.4. Класс `OrderHandlers` реализует пользовательскую логику для обработки запросов

```
public class OrderHandlers {

    private OrderService orderService;
    private KitchenService kitchenService;
    private DeliveryService deliveryService;
    private AccountingService accountingService;
    public OrderHandlers(OrderService orderService,
                        KitchenService kitchenService,
                        DeliveryService deliveryService,
                        AccountingService accountingService) {
        this.orderService = orderService;
        this.kitchenService = kitchenService;
        this.deliveryService = deliveryService;
        this.accountingService = accountingService;
    }

    public Mono<ServerResponse> getOrderDetails(ServerRequest serverRequest) {
        String orderId = serverRequest.pathVariable("orderId");

        Mono<OrderInfo> orderInfo = orderService.findOrderById(orderId);

        Mono<Optional<TicketInfo>> ticketInfo =
            kitchenService
                .findTicketByOrderId(orderId)
                .map(Optional::of)
                .onErrorReturn(Optional.empty());
        // Преобразуем TicketInfo в Optional<TicketInfo>
        // Если обращение к сервису было
        // неудачным, возвращаем Optional.empty()

        Mono<Optional<DeliveryInfo>> deliveryInfo =
            deliveryService
                .findDeliveryByOrderId(orderId)
                .map(Optional::of)
                .onErrorReturn(Optional.empty());

        Mono<Optional<BillInfo>> billInfo = accountingService
            .findBillByOrderId(orderId)
            .map(Optional::of)
            .onErrorReturn(Optional.empty());
        // Объединяем четыре значения в одно, Tuple4

        Mono<Tuple4<OrderInfo, Optional<TicketInfo>,
                Optional<DeliveryInfo>, Optional<BillInfo>>> combined =
            Mono.when(orderInfo, ticketInfo, deliveryInfo, billInfo);

        Mono<OrderDetails> orderDetails =
            combined.map(OrderDetails::makeOrderDetails);
        // Преобразуем Tuple4 в OrderDetails

        return orderDetails.flatMap(person -> ServerResponse.ok()
            .contentType(MediaType.APPLICATION_JSON)
            .body(fromObject(person)));
        // Преобразуем OrderDetails в ServerResponse
    }
}
```

Метод `getOrderDetails()` выполняет объединение API, чтобы получить подробности о заказе. Он написан в масштабируемом реактивном стиле с помощью абстракции `Mono` из состава Project Reactor. Класс `Mono`, который является более развитой версией `CompletableFuture` из Java 8, содержит результат асинхронной операции — значение или исключение. Он обладает гибким API для объединения значений, возвращаемых асинхронными операциями. Вы можете использовать его для написания конкурентного кода в простом и понятном стиле. В этом примере метод `getOrderDetails()` параллельно обращается к четырем сервисам и объединяет полученные результаты в единый объект `OrderDetails`.

Метод `getOrderDetails()` принимает в качестве параметра объект `ServerRequest`, который в Spring WebFlux представляет собой HTTP-запрос, и делает следующее.

1. Извлекает из пути `orderId`.
2. Асинхронно вызывает четыре сервиса через их прокси, получая в ответ объекты `Mono`. Для улучшения доступности метод `getOrderDetails()` считает ответы всех сервисов, кроме `Order`, необязательными. Если объект `Mono`, возвращенный необязательным сервисом, содержит исключение, вызов `onErrorReturn()` преобразует его в `Mono` с пустым экземпляром `Optional` внутри.
3. Асинхронно объединяет результаты, используя метод `Mono.when()`, который возвращает `Mono<Tuple4>` с четырьмя значениями.
4. Преобразует `Mono<Tuple4>` в `Mono<OrderDetails>` с помощью `OrderDetails::makeOrderDetails`.
5. Преобразует `OrderDetails` в объект `ServerResponse`, который в Spring WebFlux представляет собой ответ в формате JSON/HTTP.

Как видите, благодаря использованию объектов `Mono` метод `getOrderDetails()` обращается к сервисам и объединяет результаты без применения запутанных и сложных для восприятия функций обратного вызова. Давайте отдельно остановимся на прокси одного из сервисов, который возвращает результаты API-вызова, завернутые в объект `Mono`.

Класс `OrderService`

Класс `OrderService`, показанный в листинге 8.5, служит удаленным прокси для сервиса `Order`. Для обращения к этому сервису он задействует реактивный HTTP-клиент `WebClient` из состава Spring WebFlux.

Листинг 8.5. Класс `OrderService` — удаленный прокси для сервиса `Order`

```
@Service
public class OrderService {

    private OrderDestinations orderDestinations;

    private WebClient client;

    public OrderService(OrderDestinations orderDestinations, WebClient client)
```

```

    {
        this.orderDestinations = orderDestinations;
        this.client = client;
    }

    public Mono<OrderInfo> findOrderById(String orderId) {
        Mono<ClientResponse> response = client
            .get()
            .uri(orderDestinations.orderServiceUrl + "/orders/{orderId}",
                orderId)
            .exchange(); // ← Вызываем сервис
        return response.flatMap(resp -> resp.bodyToMono(OrderInfo.class)); // ← Преобразуем тело ответа в OrderInfo
    }
}

```

Метод `findOrder()` извлекает объект `OrderInfo` с заказом. Для выполнения HTTP-запроса к сервису он применяет `WebClient`, а затем десериализует ответ формата JSON в `OrderInfo`. `WebClient` обладает реактивным API и заворачивает ответы в объекты `Mono`. Метод `findOrder()` использует вызов `flatMap()`, чтобы преобразовать `Mono<ClientResponse>` в `Mono<OrderInfo>`. А метод `bodyToMono()`, как понятно из названия, возвращает тело ответа в виде `Mono`.

Класс `ApiGatewayApplication`

Класс `ApiGatewayApplication`, представленный в листинге 8.6, реализует метод `main()` API-шлюза. Это стандартный главный класс в Spring Boot.

Листинг 8.6. Метод `main()` для API-шлюза

```

@SpringBootApplication
@EnableAutoConfiguration
@EnableGateway
@Import(OrdersConfiguration.class)
public class ApiGatewayApplication {

    public static void main(String[] args) {
        SpringApplication.run(ApiGatewayApplication.class, args);
    }
}

```

Аннотация `@EnableGateway` импортирует конфигурацию Spring для фреймворка Spring Cloud Gateway.

Spring Cloud Gateway отлично подходит для реализации API-шлюза. Он позволяет сконфигурировать базовое проксирование, используя простой и лаконичный язык DSL для описания правил маршрутизации. С его помощью можно легко направлять запросы к методам-обработчикам, которые объединяют API и преобразуют протоколы. Этот фреймворк построен поверх масштабируемых и реактивных проектов Spring Framework 5 и Project Reactor. Но в вашем распоряжении есть еще один вариант для написания собственного API-шлюза — фреймворк GraphQL, который предоставляет язык запросов на основе графов. Посмотрим, как это работает.

8.3.3. Реализация API-шлюза с помощью GraphQL

Представьте, что вам доверили реализацию конечной точки `GET /orders/{orderId}` для API-шлюза FTGO, которая возвращает подробности о заказе. На первый взгляд эта задача выглядит несложной. Но, как говорилось в разделе 8.1, эта конечная точка извлекает данные из разных сервисов. Следовательно, вам необходимо использовать шаблон объединения API и написать код, который обращается к сервисам и совмещает их ответы.

Еще одна трудность, о которой упоминалось ранее, состоит в том, что разным клиентам нужны немного разные данные. Например, настольное SPA-приложение, в отличие от мобильного, показывает, какую оценку вы поставили заказу. Чтобы подогнать данные под конкретные потребности, можно позволить клиенту указывать нужную информацию (см. главу 3). Конечная точка, к примеру, может поддерживать параметры `expand` и `fields`, первый будет задавать сопутствующие ресурсы, а второй — поля каждого ресурса, которые следует вернуть. Можно также определить несколько версий данной конечной точки в рамках шаблона BFF. Это довольно хлопотно, учитывая, что рассматриваемый API-вызов является далеко не единственным в API-шлюзе приложения FTGO.

Написание API-шлюза с интерфейсом REST API с хорошей поддержкой широкого спектра клиентов занимает много времени. В связи с этим вам стоит обратить внимание на графовый API-фреймворк GraphQL, который специально создан для эффективного извлечения данных. Суть таких фреймворков заключается в том, что API сервера имеет структуру графа (рис. 8.9). Графовая структура (схема) задает набор *узлов* (типов), имеющих *свойства* (поля) и связи с другими узлами. Чтобы извлечь данные, клиент выполняет запрос, который описывает необходимую информацию в виде узлов графа, их свойств и связей. В итоге клиент может извлечь нужные ему данные за одно обращение к API-шлюзу.

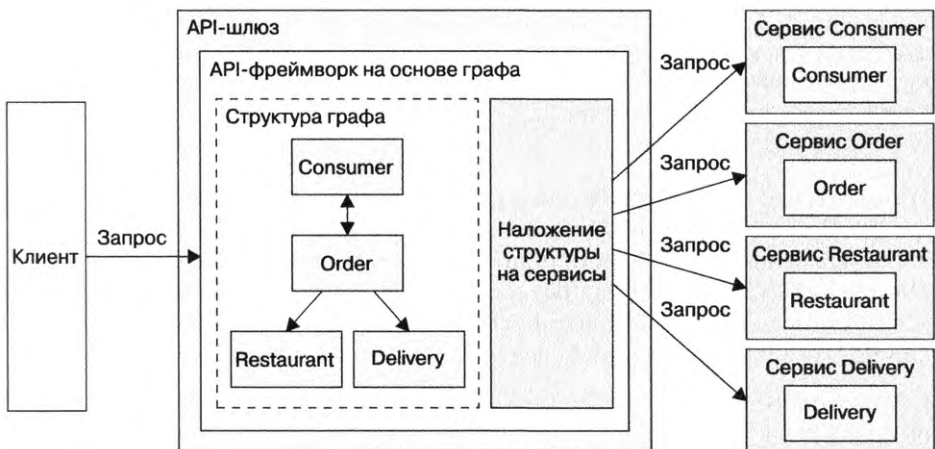


Рис. 8.9. Интерфейс API-шлюза имеет графовую структуру, которая накладывается на сервисы. Клиент выполняет запрос, извлекающий разные узлы графа. API-фреймворк на основе последнего выполняет запрос, извлекая данные из одного или нескольких сервисов

Технология графовых API имеет несколько важных преимуществ. Она позволяет клиенту контролировать возвращаемые данные. Это позволяет создавать единый API, обладающий достаточной гибкостью для поддержки разного рода клиентов. Еще одно преимущество этого подхода — то, что, несмотря на гибкость API, он значительно облегчает разработку. Это связано с тем, что для написания серверного кода используется фреймворк выполнения запросов, специально созданный для поддержки объединения и отображения API. Можно привести такую аналогию: вместо того чтобы заставлять клиенты извлекать данные с помощью хранимых процедур, которые нужно отдельно писать и поддерживать, вы позволяете им выполнять запросы к исходной базе данных.

Технологии структурированных API

Двумя самыми популярными технологиями графовых API являются GraphQL (graphql.org) и Netflix Falcor (netflix.github.io/falcor/). Netflix Falcor моделирует серверные данные в виде виртуального объектного графа в формате JSON. Чтобы извлечь данные из сервера Falcor, клиент выполняет запрос, который возвращает свойства JSON-объекта. Клиент может также обновлять эти свойства. Сервер Falcor накладывает свойства объектного графа на внутренние источники данных, такие как сервисы с REST API. Для задания или получения свойств он обращается к одному или нескольким внутренним источникам данных.

Еще одна популярная технология графовых API — фреймворк GraphQL, разработанный компанией Facebook в 2015 году. Он моделирует серверные данные в виде графа объектов с полями и ссылками на другие объекты. Объектный граф накладывается на внутренние источники данных. Клиенты GraphQL могут выполнять запросы и мутации для извлечения и создания/обновления данных соответственно. В отличие от фреймворка Netflix Falcor, который является реализацией, GraphQL представляет собой стандарт. Его клиенты и серверы доступны для разных языков, включая NodeJS, Java и Scala.

Apollo GraphQL — это популярная реализация для JavaScript/NodeJS (www.apollo-graphql.com). Платформа включает в себя сервер и клиент GraphQL. Этот проект вносит в спецификацию GraphQL мощные дополнения, например подписки для доставки клиентам изменившихся данных.

В этом разделе мы поговорим о том, как разработать API-шлюз с помощью Apollo GraphQL. Я остановлюсь на нескольких ключевых возможностях этого сервера и GraphQL в целом. Подробности ищите в документации к этим проектам.

На рис. 8.10 показан API-шлюз, основанный на GraphQL и написанный на JavaScript с использованием веб-фреймворка NodeJS Express и сервера Apollo GraphQL. Эта архитектура имеет следующие ключевые аспекты.

- ❑ *Схема GraphQL* описывает модель серверных данных и запросы, которые она поддерживает.
- ❑ *Функции сопоставления* накладывают элементы схемы на различные внутренние сервисы.
- ❑ *Прокси-классы* обращаются к сервисам приложения FTGO.

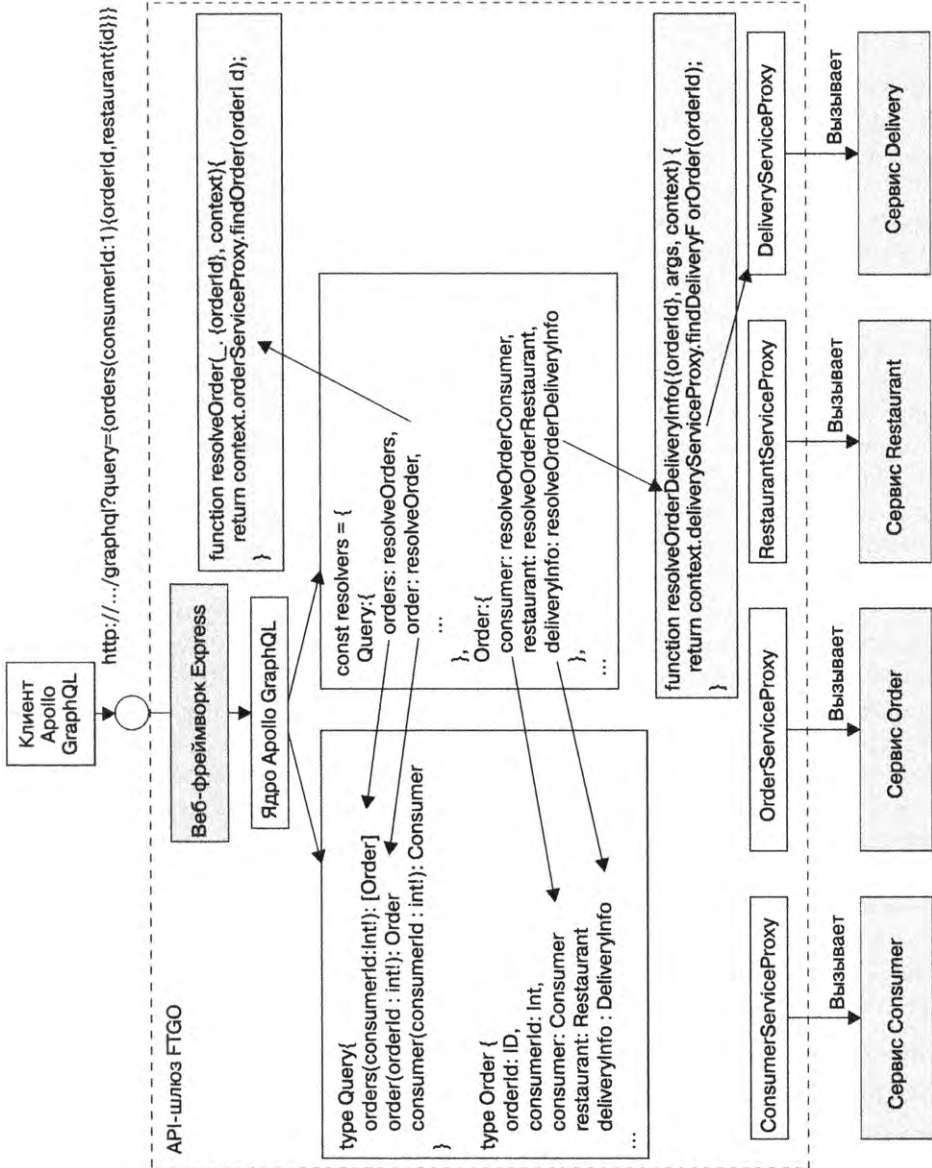


Рис. 8.10. Архитектура API-шлюза FTGO на основе GraphQL

Есть также небольшое количество связующего кода, который интегрирует сервер GraphQL с веб-фреймворком Express. Рассмотрим каждый из этих пунктов, начиная со схемы GraphQL.

Описание схемы GraphQL

Спецификация GraphQL API построена вокруг концепции *схемы* (schema) — набора типов, которые определяют структуру модели серверных данных и операции, доступные для клиента, например запросы. GraphQL поддерживает несколько разных типов. В примере кода, представленного в этом разделе, используются только два: *объектные* типы, которые являются основным способом описания модели данных, и *перечисления*, похожие на тип `enum` в Java. Объектные типы имеют имя и набор типизированных именованных полей. *Поле* может быть скалярным значением (число, строка или перечисление), списком скалярных значений, ссылкой на другой объектный тип или набором ссылок на другой объектный тип. Это напоминает поле традиционного объектно-ориентированного класса, однако в GraphQL поля концептуально являются функциями, которые возвращают значения. Они могут иметь аргументы, что позволяет клиенту подгонять возвращаемые данные под свои нужды.

Поля в GraphQL используются также для описания запросов, которые поддерживает схема. Чтобы определить такой запрос, нужно объявить объектный тип, который принято называть *Query*. Каждое поле объекта *Query* — это именованный запрос с набором опциональных параметров и возвращаемым типом. Когда я впервые столкнулся с таким способом описания запросов, он мне показался слегка запутанным, однако не стоит забывать, что в GraphQL поля являются функциями. Все станет намного понятней, когда мы посмотрим, как эти поля связаны с внутренними источниками данных.

В листинге 8.7 показан фрагмент схемы API-шлюза FTGO на основе GraphQL. В нем определены несколько объектных типов. Большинство из них относятся к элементам приложения FTGO: *Consumer*, *Order* и *Restaurant*. Вы также можете видеть объектный тип *Query*, который описывает запросы схемы.

Несмотря на другой синтаксис, объектные типы *Consumer*, *Order*, *Restaurant* и *DeliveryInfo* по своей структуре напоминают соответствующие Java-классы. Одним из отличий является тип *ID*, который представляет уникальный идентификатор.

Эта схема определяет три запроса:

- ❑ `orders()` — возвращает заказы для заданного клиента;
- ❑ `order()` — возвращает заданный заказ;
- ❑ `consumer()` — возвращает информацию о заданном клиенте.

На первый взгляд эти запросы незначительно отличаются от соответствующих конечных точек REST, однако GraphQL дает клиенту потрясающий контроль над возвращаемыми данными. Чтобы лучше в этом разобраться, посмотрим, как клиент выполняет запросы GraphQL.

Листинг 8.7. Схема GraphQL для API-шлюза FTGO

```

type Query {
  orders(consumerId : Int!): [Order]
  order(orderId : Int!): Order
  consumer(consumerId : Int!): Consumer
}

type Consumer {
  id: ID
  firstName: String
  lastName: String
  orders: [Order]
}

type Order {
  orderId: ID,
  consumerId : Int,
  consumer: Consumer
  restaurant: Restaurant

  deliveryInfo : DeliveryInfo
}

...

type Restaurant {
  id: ID
  name: String
}

...

type DeliveryInfo {
  status : DeliveryStatus
  estimatedDeliveryTime : Int
  assignedCourier :String
}

enum DeliveryStatus {
  PREPARING
  READY_FOR_PICKUP
  PICKED_UP
  DELIVERED
}

```

Определяет запросы, которые может выполнять клиент

Уникальный ID для заказчика

У заказчика есть список заказов

Выполнение запросов GraphQL

Принципиальное преимущество технологии GraphQL — ее язык запросов, который дает клиенту невероятный контроль над возвращаемыми данными. Клиент отправляет серверу документ, содержащий запрос. В простейшем случае в нем указаны название запроса, значения аргументов и поля объекта, который нужно вернуть.

Далее приведен простой запрос, который извлекает поля `firstName` и `lastName`, принадлежащие заказчику с определенным ID:

```
query {
  consumer(consumerId:1) {
    firstName
    lastName
  }
}
```

Указывает запрос `consumer`, который извлекает данные о заказчике

Поля объекта `Consumer`, которые нужно вернуть

Этот запрос возвращает поля заданного объекта `Consumer`.

А вот более сложный запрос, возвращающий сведения о заказчике, его заказы, а также ID и название каждого ресторана, который эти заказы выполнил:

```
query {
  consumer(consumerId:1) {
    id
    firstName
    lastName
    orders {
      orderId
      restaurant {
        id
        name
      }
      deliveryInfo {
        estimatedDeliveryTime
        name
      }
    }
  }
}
```

Этот запрос просит сервер вернуть нечто большее, чем просто поля объекта `Consumer`. Он извлекает заказы, которые сделал клиент, и информацию о ресторане, упомянутом в каждом из заказов. Как видите, клиент GraphQL может указать, какие именно данные следует вернуть, включая поля транзитивно связанных объектов.

Этот язык запросов более гибок, чем кажется на первый взгляд. Дело в том, что запрос находится в поле объекта `Query`, а документ указывает серверу, какие из этих полей он должен вернуть. В этих простых примерах мы извлекаем одно поле, но если в документе задать несколько полей, он выполнит соответствующее число запросов. Документ указывает, в каких полях он заинтересован, и предоставляет им подходящие аргументы. Далее показан запрос, который извлекает двух разных заказчиков:

```
query {
  c1: consumer (consumerId:1) { id, firstName, lastName}
  c2: consumer (consumerId:2) { id, firstName, lastName}
}
```

В этом документе используются так называемые *псевдонимы* `c1` и `c2`. Они позволяют различать в итоговом объекте двух заказчиков, которые сами по себе имеют одно и то же название — `consumer`. В этом примере извлекаются два объекта одного типа, но клиент может запрашивать и объекты разных типов.

Схема GraphQL определяет форму данных и поддерживаемые запросы. Чтобы она имела какое-то практическое применение, ее следует подключить к источнику данных. Посмотрим, как это делается.

Подключение схемы к источнику данных

Когда сервер GraphQL выполняет запрос, он должен извлечь запрошенные данные из одного или нескольких хранилищ. В случае с приложением FTGO ему необходимо обратиться к API сервисов, которые владеют данными. Чтобы подключить схему GraphQL к источнику данных, к полям ее объектных типов следует прикрепить функции сопоставления. Сервер GraphQL реализует шаблон объединения API путем вызова этих функций — сначала для запросов верхнего уровня, а затем рекурсивно для полей итогового объекта или объектов.

То, как функции сопоставления связываются со схемой, зависит от выбранного вами сервера GraphQL. В листинге 8.8 показано, как определить сопоставители при использовании Apollo GraphQL. Вы должны создать двухуровневый объект JavaScript. Каждое свойство верхнего уровня соотносится с объектным типом, таким как `Query` или `Order`. Каждое свойство второго уровня, такое как `Order.consumer`, определяет функцию сопоставления поля.

Листинг 8.8. Прикрепление функций сопоставления к полям схемы в GraphQL

```
const resolvers = {  
  Query: {  
    orders: resolveOrders, ← Сопоставитель для запросов orders  
    consumer: resolveConsumer,  
    order: resolveOrder  
  },  
  Order: {  
    consumer: resolveOrderConsumer, ← Сопоставитель для поля consumer в Order  
    restaurant: resolveOrderRestaurant,  
    deliveryInfo: resolveOrderDeliveryInfo  
  },  
  ...  
};
```

Функция сопоставления имеет три параметра.

- ❑ *Объект* — для поля запроса верхнего уровня это корневой объект, обычно игнорируемый функцией сопоставления. Но это может быть и значение, которое сопоставитель возвращает родительскому объекту. Например, функции сопоставления для поля `Order.consumer` передается значение, возвращенное сопоставителем `Order`.

- ❑ *Аргументы запроса* — поставляются документом запроса.
- ❑ *Контекст* — глобальное состояние выполнения запроса, доступное всем сопоставителям. Оно используется, к примеру, для передачи сопоставителям информации и зависимостей.

Функция сопоставления может вызывать один сервис или реализовать шаблон объединения API, чтобы извлекать данные из нескольких сервисов. Функции сопоставления в сервере Apollo GraphQL возвращают JavaScript-объект `Promise`, который является аналогом `CompletableFuture` в Java. Он содержит объект (или список объектов), извлеченный функцией сопоставления из хранилища данных. Ядро GraphQL включает возвращаемое значение в итоговый объект.

Рассмотрим несколько примеров. Далее показана функция `resolveOrders()`, которая выступает сопоставителем для запроса `orders`:

```
function resolveOrders(_, { consumerId }, context) {
  return context.orderServiceProxy.findOrders(consumerId);
}
```

Эта функция достает из контекста объект `OrderServiceProxy` и использует его для извлечения заказов клиента. Она игнорирует свой первый параметр. Аргумент `consumerId`, предоставленный документом запроса, передается в метод `OrderServiceProxy.findOrders()`, извлекающий заказы клиента из `OrderHistoryService`.

Далее представлена функция `resolveOrderRestaurant()`, которая выполняет сопоставление для поля `Order.restaurant`, извлекая ресторан заказа:

```
function resolveOrderRestaurant({restaurantId}, args, context) {
  return context.restaurantServiceProxy.findRestaurant(restaurantId);
}
```

Ее первый параметр — `Order`. Она вызывает `RestaurantServiceProxy.findRestaurant()` с полем `restaurantId` заказа, который предоставил метод `resolveOrders()`.

Для вызова функций сопоставления GraphQL применяет рекурсивный алгоритм. Сначала выполняются функции для запроса верхнего уровня, указанного в документе Query. Затем перебираются поля каждого возвращенного объекта, перечисленные в документе. Если у поля есть сопоставитель, он вызывается с объектом и аргументами, взятыми из того же документа. После этого алгоритм рекурсивно перебирает объект или объекты, возвращенные сопоставителем.

На рис. 8.11 показано, как этот алгоритм выполняет запрос, который извлекает заказы клиента, а также информацию о доставке и ресторанах для каждого из них. Вначале ядро GraphQL вызывает функцию `resolveConsumer()`, извлекающую объект `Consumer`. Далее она выполняет сопоставитель для поля `Consumer.orders`, `resolveConsumerOrders()`, который возвращает заказы клиента. Затем ядро GraphQL перебирает объекты `Order`, вызывая функции сопоставления для полей `Order.restaurant` и `Order.deliveryInfo`.

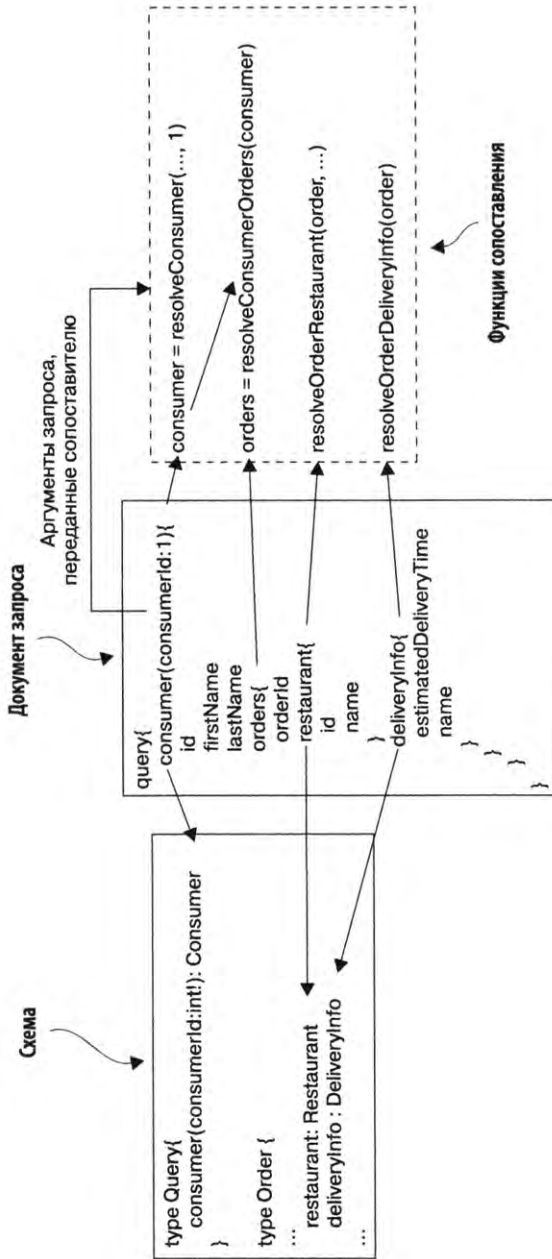


Рис. 8.11. Чтобы выполнить запрос, GraphQL рекурсивно вызывает функции сопоставления для полей, указанных в документе Query. Сначала выполняется сопоставитель для запроса, а затем — сопоставители для полей в иерархии итогового объекта

Результат выполнения сопоставителей — объект `Consumer`, который содержит данные, извлеченные из нескольких сервисов.

Теперь посмотрим, как оптимизировать работу сопоставителей с помощью пакетного выполнения и кэширования.

Оптимизация загрузки с помощью пакетного выполнения и кэширования

При выполнении запроса GraphQL может вызвать много сопоставителей. Поскольку каждый сопоставитель выполняется сервером GraphQL по отдельности, существует риск снижения производительности из-за лишних обращений к сервисам. Возьмем, к примеру, запрос, который извлекает информацию о клиенте, его заказах и ресторанах, которые эти заказы выполнили. Если есть N заказов, в простейшей реализации мы сделаем один вызов сервиса `Consumer`, один вызов сервиса `Order History` и затем N вызовов сервиса `Restaurant`. Несмотря на то что в обычных условиях ядро GraphQL выполнило бы вызовы сервиса `Restaurant` параллельно, мы все равно рискуем получить низкую производительность. К счастью, существует несколько методик, которые способны помочь в этой ситуации.

Одна из важных оптимизаций — совместное пакетное выполнение и кэширование на стороне сервера. *Пакетное выполнение* превращает N обращений к сервису, такому как `Restaurant`, в один-единственный вызов, который извлекает набор из N объектов. *Кэширование* позволяет повторно использовать результаты предыдущего извлечения того же объекта и тем самым избежать необязательного дублирования вызовов. Сочетание этих двух подходов значительно снижает количество обращений к внутренним сервисам.

Сервер GraphQL, базирующийся на NodeJS, может задействовать модуль `DataLoader` для реализации пакетного выполнения и кэширования (github.com/facebook/dataloader). Он объединяет загрузки, которые происходят в рамках одной итерации рабочего цикла, и вызывает предоставленную вами пакетную функцию. Кроме того, он кэширует вызовы, чтобы избежать дублирования загрузок. В листинге 8.9 показано, как `RestaurantServiceProxy` может использовать `DataLoader`. Метод `findRestaurant()` применяет этот модуль для загрузки объектов `Restaurant`.

Листинг 8.9. Использование `DataLoader` для оптимизации обращения к вызову `Restaurant`

```
const DataLoader = require('dataloader');

class RestaurantServiceProxy {
  constructor() {
    this.dataLoader =
      new DataLoader(restaurantIds =>
        this.batchFindRestaurants(restaurantIds));
  }

  findRestaurant(restaurantId) {
```

Создаем объект `DataLoader`, который использует `batchFindRestaurants()` в качестве пакетной функции

Загружаем заданный экземпляр `Restaurant` через `DataLoader`

```

    return this.dataLoader.load(restaurantId);
  }

  batchFindRestaurants(restaurantIds) {
    ...
  }
}

```

Загружаем набор объектов Restaurant

`RestaurantServiceProxy` и, следовательно, `DataLoader` создаются для каждого запроса, поэтому `DataLoader` в принципе не может смешать данные разных пользователей.

Теперь посмотрим, как интегрировать ядро GraphQL с веб-фреймворком, чтобы к нему могли обращаться клиенты.

Интеграция сервера Apollo GraphQL с Express

Сервер Apollo GraphQL выполняет запросы формата GraphQL. Чтобы клиенты могли к нему обращаться, вы должны интегрировать его с веб-фреймворком. Server Apollo GraphQL поддерживает несколько веб-фреймворков, включая Express — популярное решение для NodeJS.

В листинге 8.10 показано, как использовать сервер Apollo GraphQL в приложении на основе Express. Ключевой функцией здесь является `graphqlExpress`, которая предоставляется модулем `apollo-server-express`. Она формирует метод-обработчик для Express, который выполняет запросы к схеме в формате GraphQL. В этом примере приложение Express сконфигурировано для перенаправления запросов к конечным точкам `GET /graphql` и `POST /graphql`, принадлежащим обработчику запросов GraphQL. Кроме того, оно создает контекст GraphQL и помещает в него прокси-объекты, благодаря чему они становятся доступными для сопоставителей.

Листинг 8.10. Интеграция сервера GraphQL с веб-фреймворком Express

```

const {graphqlExpress} = require("apollo-server-express");

const typeDefs = gql`
  type Query {
    orders: resolveOrders,
    ...
  }

  type Consumer {
    ...
  }
`

const resolvers = {
  Query: {
    ...
  }
}

const schema = makeExecutableSchema({ typeDefs, resolvers });

```

Определяем схему GraphQL

Определяем сопоставители

Связываем схему с сопоставителями, чтобы создать исполняемую схему

```

const app = express();

function makeContextWithDependencies(req) {
  const orderServiceProxy = new OrderServiceProxy();
  const consumerServiceProxy = new ConsumerServiceProxy();
  const restaurantServiceProxy = new RestaurantServiceProxy();
  ...
  return {orderServiceProxy, consumerServiceProxy,
          restaurantServiceProxy, ...};
}

function makeGraphQLHandler() {
  return graphqlExpress(req => {
    return {schema: schema, context: makeContextWithDependencies(req)}
  });
}

app.post('/graphql', bodyParser.json(), makeGraphQLHandler());
app.get('/graphql', makeGraphQLHandler());
app.listen(PORT);

```

Внедряем репозитории в контекст, чтобы сделать их доступными для сопоставителей

Обработчик Express, который шлет GraphQL-запросы к исполняемой схеме

Направляем конечные точки POST /graphql и GET /graphql к серверу GraphQL

В этом примере не учитываются такие аспекты, как безопасность, но их было бы несложно реализовать. API-шлюз мог бы, к примеру, аутентифицировать пользователей с помощью Passport — фреймворка для обеспечения безопасности в NodeJS (см. главу 11). Функция `makeContextWithDependencies()` передавала бы информацию о пользователе каждому конструктору, чтобы тот переслал ее сервисам.

Теперь поговорим о том, как клиент может обратиться к серверу, чтобы выполнить GraphQL-запросы.

Написание клиента GraphQL

Клиентское приложение может обращаться к серверу GraphQL несколькими способами. Поскольку сервер GraphQL имеет API на основе HTTP, клиент способен выполнять запросы с помощью HTTP-библиотеки — например, `GET http://localhost:3000/graphql?query={orders(consumerId:1){orderId,restaurant{id}}}`. Более простое решение — использование клиентской библиотеки GraphQL, которая берет на себя корректное форматирование запросов и обычно предоставляет такие возможности, как кэширование на клиентской стороне.

В листинге 8.11 показан класс `FtgoGraphQLClient`, который представляет собой простой клиент для приложения FTGO на основе GraphQL. Его конструктор создает объект `ApolloClient`, который входит в состав клиентской библиотеки `Apollo GraphQL`. Класс `FtgoGraphQLClient` определяет метод `findConsumer()`, который действует этот клиент для извлечения имени заказчика.

Класс `FtgoGraphQLClient` может содержать различные методы запросов, такие как `findConsumer()`. Каждый из них выполняет запрос, который извлекает именно те данные, которые нужны клиенту.

Листинг 8.11. Использование клиента Apollo GraphQL для выполнения запросов

```

class FtgoGraphQLClient {

    constructor(...) {
        this.client = new ApolloClient({ ... });
    }

    findConsumer(consumerId) {
        return this.client.query({
            variables: { cid: consumerId},
            query: gql`
                query foo($cid : Int!) {
                    consumer(consumerId: $cid) {
                        id
                        firstName
                        lastName
                    }
                }
            `
        })
    }
}

```

Предоставляем значение для \$cid

Определяем \$cid как переменную типа Int

Присваиваем \$cid параметру запроса consumerid

В этом разделе мы едва затронули возможности GraphQL. Надеюсь, мне удалось продемонстрировать, что эта технология является привлекательной альтернативой более традиционному API-шлюзу, основанному на REST. Таким образом, при разработке своих API-шлюзов вы должны рассматривать GraphQL в качестве одного из вариантов.

Резюме

- ❑ Обычно внешние клиенты приложения обращаются к его сервисам через API-шлюз. API-шлюз предоставляет каждому клиенту отдельный API. Он отвечает за маршрутизацию запросов, объединение API, преобразование протоколов и реализацию таких граничных функций, как аутентификация.
- ❑ У вашего приложения может быть один или несколько API-шлюзов, по одному для каждого типа клиентов. В последнем случае применяется шаблон BFF. Основное его преимущество в том, что он делает команды клиентской разработки более автономными, поскольку каждая из них пишет, развертывает и администрирует собственный API-шлюз.
- ❑ Существует целый ряд технологий, используемых для реализации API-шлюза, включая готовые решения. Но вы можете разработать собственный API-шлюз с помощью фреймворка.
- ❑ Spring Cloud Gateway — это простой в применении фреймворк, который хорошо подходит для разработки API-шлюзов. Для маршрутизации запросов он может задействовать любые их атрибуты, включая метод и путь. Он может направлять запросы напрямую к внутренним сервисам или к пользовательскому

методу-обработчику. Этот проект основан на масштабируемых реактивных фреймворках Spring Framework 5 и Project Reactor. Вы можете писать пользовательские обработчики запросов в реактивном стиле на основе таких абстракций, как `Mono` из состава Project Reactor.

- Еще одна отличная основа для разработки API-шлюзов — фреймворк GraphQL, предоставляющий графовый язык запросов. Для описания модели серверных данных и запросов, которые она поддерживает, используется схема в виде графа. Она накладывается на ваши сервисы путем написания функций сопоставления, которые извлекают данные. Клиенты, основанные на GraphQL, обращаются к схеме, указывая серверу, какие именно данные он должен вернуть. В итоге API-шлюз, построенный по этой технологии, поддерживает разные виды клиентов.

Тестирование микросервисов, часть 1

В этой главе

- Эффективные стратегии тестирования микросервисов.
- Применение макетов и заглушек для изолированного тестирования программных элементов.
- Использование пирамиды тестов для расстановки приоритетов при тестировании.
- Модульное тестирование классов внутри сервиса.

В FTGO, как и во многих других организациях, подход к тестированию традиционный. *Тестирование* — это процесс, проводимый в основном после разработки. Разработчики передают написанный код коллегам из отдела обеспечения качества, которые проверяют, работает ли программный продукт так, как задумано. Большая часть тестирования выполняется вручную. К сожалению, такой подход неприемлем по двум причинам.

- ❑ *Ручное тестирование чрезвычайно неэффективно.* Никогда не просите человека сделать то, что у компьютера получается намного лучше. По сравнению с компьютерами люди работают с низкой производительностью и не способны делать это круглосуточно. Если вы полагаетесь на ручное тестирование, забудьте о быстрой и безопасной доставке программного обеспечения. Написание автоматических тестов — это очень важно.
- ❑ *Тестирование производится на слишком позднем этапе процесса доставки.* Безусловно, тестирование уже написанного приложения имеет право на существование, но, как показывает опыт, этого недостаточно. Написание автоматических тестов

намного лучше сделать частью разработки. Это повысит продуктивность, поскольку разработчики смогут видеть результаты тестирования во время редактирования кода.

В этом отношении FTGO — типичная организация. Отчет Sauce Labs Testing Trends за 2018 год рисует довольно мрачную картину состояния автоматизации тестов (saucelabs.com/resources/white-papers/testing-trends-for-2018). В нем говорится, что в основном автоматизированными являются лишь 26 % организаций, а полностью автоматизированными — жалкие 3 %!

Зависимость от ручных тестов не связана с нехваткой инструментария и фреймворков. Например, JUnit — популярный фреймворк тестирования для Java — был выпущен еще в 1998 году. Плачевное состояние автоматических тестов вызвано культурными аспектами: «тестированием должен заниматься отдел обеспечения качества», «у разработчиков есть более важные задачи» и т. д. Ситуацию не улучшает и то, что создание набора быстрых, эффективных и легкоподдерживаемых тестов требует больших усилий. К тому же крупное монолитное приложение, как правило, очень сложно протестировать.

Как упоминалось в главе 2, ключевым фактором при выборе микросервисной архитектуры является улучшение тестируемости. В то же время из-за своей сложности микросервисный подход *требует* написания автоматических тестов. Более того, некоторые аспекты тестирования микросервисов трудно реализовать, ведь необходимо убедиться в том, что они могут корректно взаимодействовать между собой, но при этом минимизировать количество медленных, сложных и ненадежных сквозных тестов, которые требуют запуска множества сервисов.

Это первая из двух глав, посвященных тестированию. Считайте ее введением. В главе 10 рассматриваются более продвинутые концепции. Обе они довольно длинные, но в них вы сможете найти идеи и методики тестирования, необходимые для разработки современного программного обеспечения в целом и микросервисной архитектуры в частности.

Я начну главу с описания эффективных стратегий тестирования микросервисных приложений. Эти стратегии дадут вам уверенность в работоспособности вашего кода, минимизируя при этом сложность тестов и время их выполнения. После этого я продемонстрирую написание определенного вида тестов для сервисов, называемых модульными. Другие типы тестов — интеграционные, компонентные и сквозные — будут описаны в главе 10.

Поговорим о стратегиях тестирования микросервисов.

Почему мы начинаем с введения в тестирование

Вам, наверное, интересно, почему эта глава включает в себя знакомство с базовыми принципами тестирования. Если вы уже имели дело с такими понятиями, как пирамида тестов и тесты разных типов, можете быстро пролистать ее и перейти к следующей, которая посвящена аспектам тестирования, связанным с микросервисами. Но мой опыт консультации и обучения клиентов по всему миру говорит о том, что фундаментальной слабостью многих организаций, занимающихся разработкой программного обеспечения,

является отсутствие автоматических тестов. Автоматическое тестирование *необходимо*, если вам нужна быстрая и надежная доставка обновлений. Это единственный способ сокращения времени, которое уходит на развертывание зафиксированного кода в промышленной среде. Возможно, еще более важный момент — то, что автоматические тесты заставляют вас разрабатывать приложения, которые в принципе можно протестировать. Обычно автоматическое тестирование очень непросто внедрить в большой и сложный проект. Иными словами, если вы хотите максимально быстро очутиться в монолитном аду, не пишите автоматические тесты.

9.1. Стратегии тестирования микросервисных архитектур

Представьте, что вы вносите изменение в сервис `Order` приложения FTGO. Вслед за этим будет естественно запустить измененный код и убедиться в его корректной работе. Вы можете протестировать изменение вручную. Сначала нужно запустить сервис `Order` и все его зависимости, включая инфраструктурные компоненты наподобие БД и другие сервисы приложения. Затем, чтобы проверить сервис, вы должны обратиться к нему либо через его API, либо через пользовательский интерфейс приложения. Это медленный ручной способ тестирования кода.

Куда более подходящим выбором будет написание автоматических тестов, которые можно запускать во время разработки. Процесс написания приложения должен выглядеть так: отредактировать код, запустить тесты (в идеале одним нажатием клавиши), повторить. Если тесты выполняются быстро, через несколько секунд станет ясно, работает ли измененный код. Но как сделать тесты быстрыми? Как узнать, требуется ли более комплексное тестирование? Ответы на эти вопросы я даю в этом и следующем разделах.

Данный раздел начинается с обзора важных концепций автоматического тестирования. Вы увидите, какие задачи оно решает, и познакомитесь со структурой типичного теста. Я опишу различные типы тестов, которые вам нужно будет создавать. Также будет представлена пирамида тестов, которая служит полезным руководством относительно того, какие участки кода больше всего нуждаются в тестировании. После знакомства с основными концепциями мы поговорим о стратегиях тестирования микросервисов и присущих им конкретных трудностях. Вы познакомитесь с методиками, позволяющими писать более простые, быстрые, но в то же время эффективные тесты для микросервисной архитектуры.

Итак, начнем с концепций тестирования.

9.1.1. Обзор методик тестирования

В этой главе основное внимание уделяется автоматическим тестам. Упомянув здесь какие-либо *тесты*, я подразумеваю, что они *автоматические*. «Википедия» дает следующее определение *тестового случая*: «Тестовый случай — это формально

описанный алгоритм тестирования программы, специально созданный для определения возникновения в программе определенной ситуации, определенных выходных данных»¹.

Иными словами, целью теста (рис. 9.1) является проверка поведения тестируемой системы. Под *системой* здесь подразумевается элемент программного обеспечения, к которому применяется тест. Это может быть всего лишь класс, или целое приложение, или нечто среднее, например набор классов или отдельный сервис. Коллекция взаимосвязанных тестов формирует *тестовый набор*.

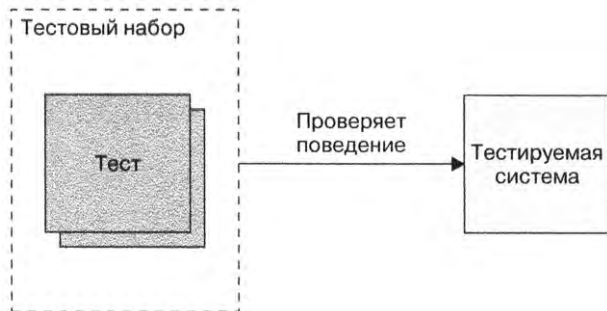


Рис. 9.1. Цель теста состоит в проверке поведения тестируемой системы. Система может быть всего лишь классом или целым приложением

Сначала мы рассмотрим концепцию автоматических тестов. Затем обсудим тесты разных типов, которые вам придется писать. После этого будет представлена пирамида тестов, описывающая относительные пропорции тестов, необходимых в том или ином случае.

Написание автоматических тестов

Автоматические тесты обычно пишут, используя специальный фреймворк. Например, JUnit — популярный фреймворк тестирования для Java. Структура автоматического теста показана на рис. 9.2. Каждый тест реализуется в виде метода, который принадлежит тестовому классу.

Типичный автоматический тест состоит из четырех этапов (xunitpatterns.com/Four%20Phase%20Test.html).

1. *Подготовка* — инициализирует среду тестирования, состоящую из самой системы и ее зависимостей, приводя ее в нужное состояние. Например, создает тестируемый класс и приводит его в состояние, необходимое для демонстрации желаемого поведения.
2. *Выполнение* — запускает тестируемую систему, например вызов метода из тестируемого класса.

¹ ru.wikipedia.org/wiki/Вариант_тестирования.

3. *Проверка* — делает выводы о результате выполнения и состоянии тестируемой системы. Например, проверяет значение, возвращаемое методом, и новое состояние тестируемого класса.
4. *Очистка* — удаляет среду тестирования, если это необходимо. Многие тесты пропускают этот этап, но, например, при тестировании БД иногда нужно откатить транзакции, инициированные на этапе подготовки.

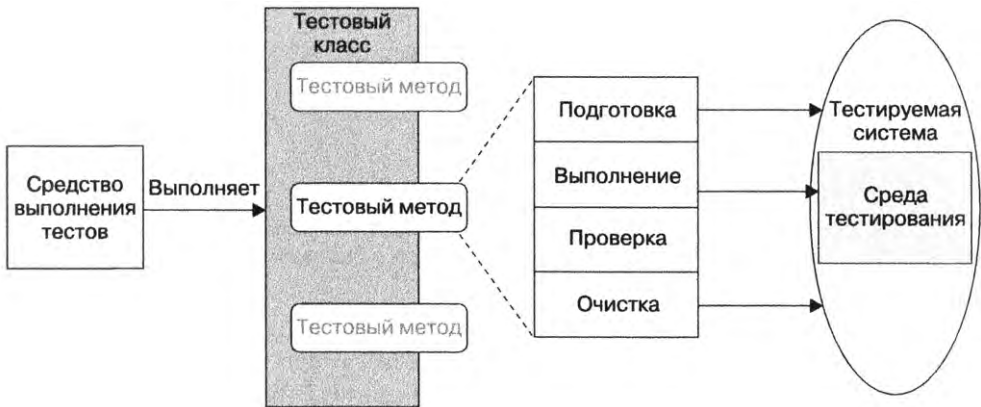


Рис. 9.2. Каждый автоматический тест реализуется тестовым методом, который принадлежит тестовому классу. Тест состоит из четырех этапов: подготовки — подготовки среды тестирования (то есть всего, что необходимо для выполнения теста); выполнения — запуска тестируемой системы; проверки — оценки результатов теста; очистки — удаления среды тестирования

Чтобы уменьшить дублирование кода и упростить тесты, в тестовый класс иногда добавляют подготовительные методы, которые выполняются перед вызовом самого теста, и методы очистки, реализуемые в самом конце. Тестовый *набор* — это перечень тестовых классов. Для их запуска используется *средство выполнения тестов*.

Тестирование с помощью макетов и заглушек

Тестируемая система часто имеет зависимости, которые могут осложнить и замедлить ваши тесты. Например, класс `OrderController` обращается к сервису `Order`, который так или иначе зависит от многих других прикладных и инфраструктурных сервисов. Тестирование класса `OrderController` путем запуска значительной части приложения было бы непрактичным. Нам нужно как-то изолировать свои тесты.

Решение, показанное на рис. 9.3, состоит в замене зависимостей тестируемой системы дублерами. *Дублер* — это объект, который симулирует поведение зависимости.

Существует два вида дублеров: заглушки (stubs) и макеты (mocks). Эти термины часто считают взаимозаменяемыми, хотя они немного различаются. *Заглушка* — это дублер, который возвращает значения тестируемой системе. *Макет* — это дублер, используемый тестом для проверки того, что тестируемая система корректно вызывает свои зависимости. Во многих случаях макет является заглушкой.



Рис. 9.3. Замена зависимости дублером, который позволяет тестировать систему в изоляции. Тест получается более простым и быстрым

Позже в этой главе вы увидите примеры применения дублеров. Так, в подразделе 9.2.5 будет показано, как протестировать класс `OrderController` в изоляции, задействуя дублер класса `OrderService`. В этом примере дублер `OrderService` реализуется с помощью Mockito — популярного фреймворка для создания макетов объектов в Java. В главе 10 вы увидите, как протестировать сервис `Order`, используя дублеры тех сервисов, к которым он обращается. Эти дублеры будут отвечать на командные сообщения, отправляемые сервисом `Order`.

Теперь рассмотрим разные типы тестов.

Типы тестов

Существует множество типов тестов. Некоторые из них, такие как тесты производительности и удобства использования, позволяют убедиться в том, что приложение отвечает требованиям к качеству обслуживания. В этой главе основное внимание уделяется автоматическим тестам, которые проверяют функциональные аспекты приложения или сервисов. Вы узнаете, как пишутся тесты четырех разных типов:

- ❑ *модульные тесты*, которые тестируют небольшую часть сервиса, такую как класс;
- ❑ *интеграционные тесты*, которые проверяют, может ли сервис взаимодействовать с инфраструктурными компонентами, такими как базы данных и другие сервисы приложения;
- ❑ *компонентные тесты* — приемочные тесты для отдельного сервиса;
- ❑ *сквозные тесты* — приемочные тесты для целого приложения.

Они различаются в основном охватом. На одном конце спектра находятся модульные тесты, которые проверяют поведение наименьшего значимого элемента

программы. В объектно-ориентированных языках, таких как Java, это класс. Их противоположность — сквозные тесты, проверяющие поведение целого приложения. Посередине находятся компонентные тесты, относящиеся к отдельным сервисам. Интеграционные тесты, как вы увидите в следующей главе, имеют относительно небольшой охват, но они сложнее, чем обычные модульные тесты. Охват — это лишь один из способов охарактеризовать тест. Еще одна характеристика — тестовый квадрант.

Модульные тесты на этапе компиляции

Тестирование — это неотъемлемая часть разработки. Современный процесс разработки состоит из редактирования кода с последующим запуском тестов. Более того, если вы практикуете TDD (Test-Driven Development — разработка через тестирование), создание новой функции или исправление ошибки подразумевает предварительное написание неисправного теста, который ваш код должен успешно пройти. Но даже если вы не поклонник TDD, написание теста, воспроизводящего ошибку, и кода, который ее исправляет, — превосходный подход.

Тесты, которые запускаются в ходе этого процесса, называются тестами *этапа компиляции*. В современных IDE, таких как IntelliJ IDEA и Eclipse, обычно не предусматривается отдельного шага для компиляции приложения. Вместо этого обеспечивается комбинация клавиш, которая компилирует код и сразу запускает тесты. Чтобы оставаться в этом потоке, тесты должны выполняться как можно быстрее — желательно не дольше нескольких секунд.

Использование тестового квадранта для классифицирования тестов

Хороший способ классифицировать тесты — тестовый квадрант, предложенный Брайаном Мариком (Brian Marick) (www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1). Он группирует тесты по двум основаниям (рис. 9.4).

- ❑ *К чему относится тест — к бизнесу или технологиям.* Бизнес-тест описывается в терминологии специалиста проблемной области, тогда как для описания технологического теста используется терминология разработчиков и реализации.
- ❑ *Какова цель теста — помочь с написанием кода или дать оценку приложению.* Разработчики применяют тесты, которые помогают им в ежедневной работе. Тесты, оценивающие приложение, нужны для определения проблемных участков.

Тестовый квадрант определяет четыре категории тестов:

- ❑ *Q1* — помощь в программировании с ориентацией на технологии — модульные и интеграционные тесты;
- ❑ *Q2* — помощь в программировании с ориентацией на бизнес — компонентные и сквозные тесты;

- ❑ Q3 — оценка приложения с точки зрения бизнеса — проверка удобства использования и исследовательское тестирование;
- ❑ Q4 — оценка приложения с точки зрения технологий — нефункциональное приемочное тестирование, такое как проверка производительности.

Помимо тестового квадранта, существуют и другие способы организации тестов. Например, пирамида тестов помогает определить, сколько тестов каждого типа следует написать.



Рис. 9.4. Тестовый квадрант классифицирует тесты по двум основаниям. Первое — это ориентация теста на бизнес или технологии. Второе — назначение теста: помощь в программировании или оценка приложения

Применение пирамиды тестов как средства приоритизации тестирования

Чтобы удостовериться, что наше приложение работает, мы должны написать разного рода тесты. Но проблема в том, что с увеличением охвата теста растут его сложность и время выполнения. Кроме того, чем шире охват теста и чем больше составных элементов он в себя включает, тем менее надежным становится. ненадежные тесты не намного лучше, чем их отсутствие, ведь, если тесту нельзя доверять, его сбои, скорее всего, будут игнорироваться.

На одном конце спектра располагаются модульные тесты для отдельных классов. Они надежны, просты в написании и быстро выполняются. На противоположном конце находятся сквозные тесты для всего приложения. Они медленные, их сложно

писать, а из-за сложности они часто оказываются ненадежными. Поскольку наш бюджет на разработку и тестирование ограничен, мы хотим сосредоточиться на написании тестов с небольшим охватом, не ставя при этом под угрозу эффективность тестового набора.

Хорошим подспорьем в этом может стать пирамида тестов (рис. 9.5) (martinfowler.com/bliki/TestPyramid.html). В ее основании лежат быстрые, простые и надежные тесты. Сквозные тесты, отличающиеся низкой скоростью, высокой сложностью и ненадежностью, расположены на вершине. Пирамида тестов описывает относительные пропорции каждого типа тестирования. Она похожа на пирамиду питания, которую публикует Министерство сельского хозяйства США (ru.wikipedia.org/wiki/Пирамида_питания), но, честно говоря, приносит больше пользы и вызывает меньше споров.



Рис. 9.5. Пирамида тестов описывает относительные пропорции типов тестов, которые нужно написать. Продвигаясь вверх, вы должны писать все меньше и меньше тестов

Суть этого подхода заключается в том, что с продвижением вверх по пирамиде мы должны писать все меньше и меньше тестов. Модульных тестов должно быть много, а сквозных — мало.

В этой главе я рассмотрю стратегию, которая делает акцент на тестировании элементов сервиса. Она минимизирует даже количество компонентных тестов, которые проверяют сервис целиком.

Тестирование отдельных микросервисов наподобие **Consumer**, не зависящих от других сервисов, не составляет труда. Но как насчет таких сервисов, как **Order**, у которых есть многочисленные зависимости? Как можно быть уверенным в том, что приложение в целом работоспособно? Это ключевые вопросы, которые относятся к тестированию приложений с микросервисной архитектурой. Сложность тестирования — характеристика не столько отдельных сервисов, сколько взаимодействия между ними. Давайте посмотрим, как подойти к этой проблеме.

9.1.2. Трудности тестирования микросервисов

Межпроцессное взаимодействие играет намного более важную роль в микросервисной архитектуре, чем в монолитном приложении. Монолитный код может общаться с несколькими внешними клиентами и сервисами. Например, монолитная версия FTGO использует несколько сторонних веб-сервисов со стабильными API: Stripe для платежей, Twilio для обмена сообщениями и Amazon SES — для почты. Любое взаимодействие между внутренними модулями происходит на уровне языка программирования. Фактически IPC находится на границе приложения.

Для сравнения: в микросервисной архитектуре межпроцессное взаимодействие играет одну из ключевых ролей. Микросервисное приложение — распределенная система. Разные команды заняты разработкой своих сервисов и развитием их API. Очень важно, чтобы разработчики сервиса писали тесты, которые проверяют, как он взаимодействует со своими зависимостями и клиентами.

Как говорилось в главе 3, сервисы могут общаться между собой, применяя разнообразные стили и механизмы IPC. В некоторых случаях используется стиль «запрос/ответ», который реализуется с помощью таких синхронных протоколов, как REST или gRPC. Сервисы могут общаться также в стиле «запрос/асинхронный ответ» или «издатель/подписчик», обмениваясь асинхронными сообщениями. Например, на рис. 9.6 показано, как взаимодействуют некоторые сервисы приложения FTGO. Каждая стрелка ведет от потребителя к отправителю.

Стрелка указывает в направлении зависимости — от потребителя API к сервису, который его предоставляет. Ожидания потребителя относительно API зависят от природы взаимодействия:

- ❑ *REST-клиент → сервис.* API-шлюз направляет запросы к сервисам и занимается объединением API.
- ❑ *Потребитель доменных событий → издатель.* Сервис Order History потребляет события, публикуемые сервисом Order.
- ❑ *Сторона, запрашивающая командные сообщения → отвечающая сторона.* Сервис Order шлет командные сообщения различным сервисам и потребляет их ответы.

Каждое взаимодействие между двумя сервисами может быть представлено в виде соглашения или контракта. Например, сервисы Order History и Order должны согласовать структуру сообщений с событиями и канал, в который те будут публиковаться. Точно так же API-шлюз и сервисы должны согласовать конечные точки REST API. К тому же сервис Order и все другие сервисы, с которыми он общается с помощью асинхронных запросов и ответов, должны выбрать командный канал, а также формат команд и ответов.

Вы, как разработчик сервиса, должны быть уверены в стабильности API, которые потребляете. По этой же причине не следует вносить ломающие изменения в API собственного сервиса. Например, если вы работаете над сервисом Order, то должны быть уверены в том, что разработчики его зависимостей, таких как сервисы Consumer и Kitchen, не нарушат совместимость их API с вашим кодом. Точно так же вы должны следить за тем, чтобы изменения в API сервиса Order не повлияли на совместимость с API-шлюзом или сервисом Order History.

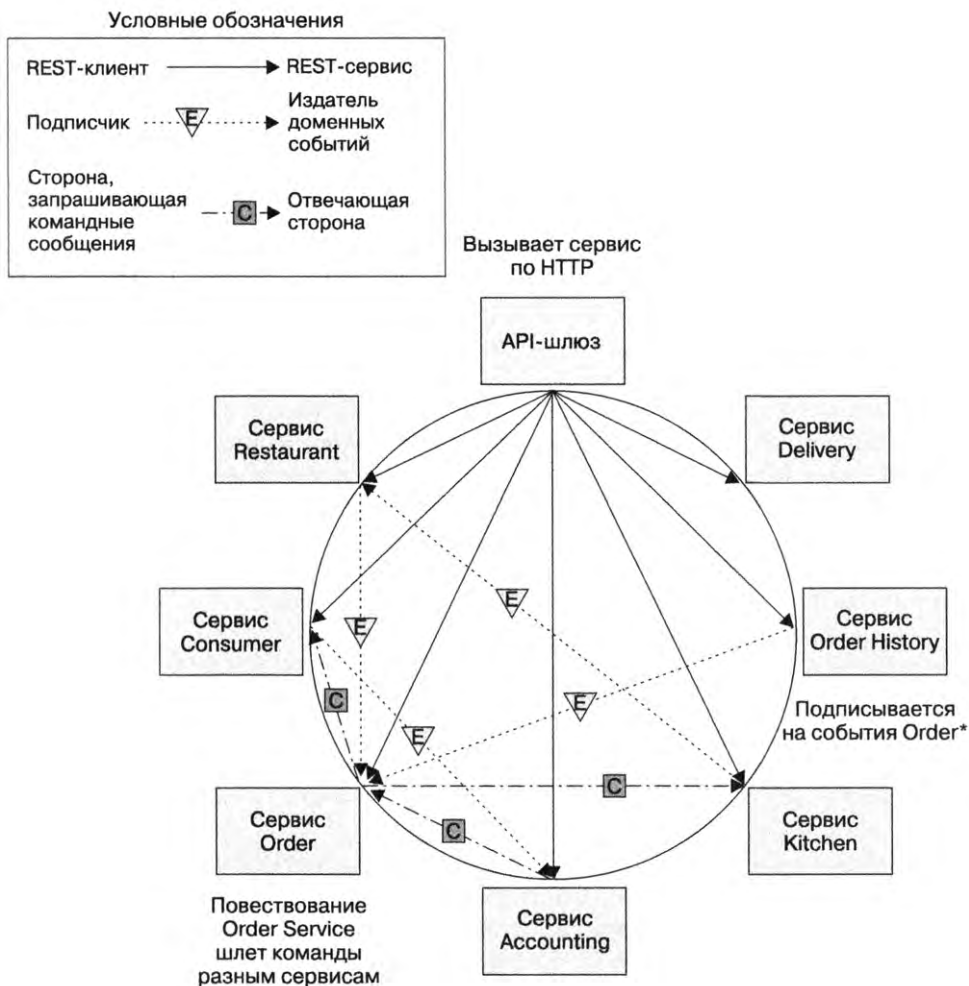


Рис. 9.6. Часть межсервисного взаимодействия в приложении FTGO

Чтобы проверить, способны ли два сервиса общаться между собой, можно обратиться к API, который инициирует взаимодействие, и убедиться в том, что он возвращает ожидаемый результат. Это точно поможет выявить проблемы с интеграцией, но это, в сущности, сквозной тест. Скорее всего, ему придется запускать множество других переходных зависимостей этих сервисов. Возможно, он также должен будет вызывать сложные высокоуровневые функции, такие как бизнес-логика, хотя его цель — проверка относительно низкоуровневого механизма IPC. Лучше всего избегать написания подобных сквозных тестов. Нам нужны более быстрые, простые и надежные тесты, которые в идеале проверяют работу сервисов в изоляции. В качестве решения можно воспользоваться так называемым *тестированием контрактов с расчетом на потребителя*.

Тестирование контрактов с расчетом на потребителя

Представьте, что вы являетесь членом команды, которая занимается разработкой API-шлюза, описанного в главе 8. Объект шлюза `OrderServiceProxy` обращается к различным конечным точкам REST, включая `GET /orders/{orderId}`. В этом случае крайне важно иметь тесты, которые проверяют согласованность API между API-шлюзом и сервисом `Order`. В терминологии тестирования потребительских контрактов между этими двумя сервисами имеется связь *«потребитель — провайдер»*. Потребителем выступает API-шлюз, а провайдером — сервис `Order`. Проверка потребительского контракта — это интеграционный тест для провайдера, такого как сервис `Order`, он позволяет убедиться в том, что API провайдера отвечает ожиданиям потребителя, такого как API-шлюз.

Тестирование потребительского контракта сосредоточено на проверке того, что API провайдера по своей форме отвечает ожиданиям потребителя. В данном случае оно позволяет убедиться в том, что провайдер реализует конечную точку REST, которая:

- ☐ имеет нужные HTTP-метод и путь;
- ☐ принимает нужные заголовки, если таковые имеются;
- ☐ принимает тело запроса, если оно имеется;
- ☐ возвращает ответ с ожидаемыми кодом состояния, заголовками и телом.

Следует помнить, что тесты контрактов не занимаются тщательной проверкой бизнес-логики провайдера. За это отвечают модульные тесты. Позже вы увидите, что в контексте REST API тесты потребительских контрактов на самом деле являются тестами макетов контроллеров.

Команда, разрабатывающая потребительский код, пишет набор тестов для контрактов и делает его частью тестового набора провайдера, например, через запрос на принятие изменений. Разработчики других сервисов, которые обращаются к сервису `Order`, тоже вносят свой вклад в этот набор (рис. 9.7). Каждый набор тестов будет проверять те аспекты API `Order`, которые относятся к тому или иному потребителю. Например, тестовый набор для сервиса `Order History` проверяет, публикует ли сервис `Order` ожидаемые события.

Эти тестовые наборы выполняются в процессе развертывания сервиса `Order`. Если проверка потребительского контракта завершается неудачно, разработчики провайдера делают вывод о том, что они внесли ломающее изменение в API. Им следует либо исправить API, либо связаться с командой потребительской стороны.

Шаблон «Тестирование контрактов с расчетом на потребителя»

Проверяет, соответствует ли сервис ожиданиям своих клиентов. См. microservices.io/patterns/testing/service-integration-contract-test.html.

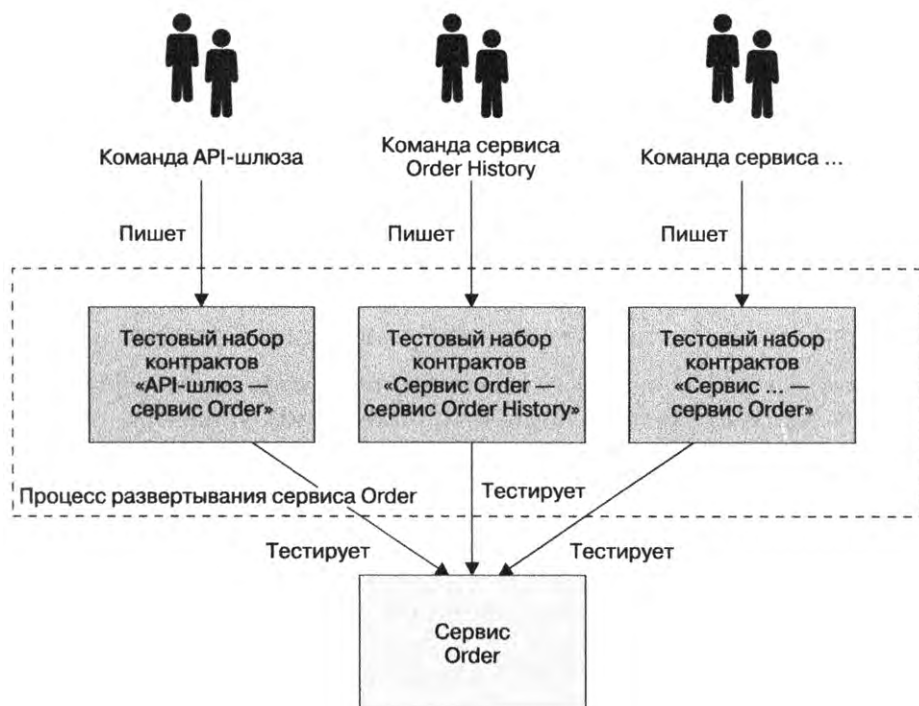


Рис. 9.7. Команда каждого сервиса, который потребляет API сервиса Order, предоставляет тестовый набор контрактов. Этот набор проверяет, соответствует ли API ожиданиям потребителей. Данные тесты в сочетании с тестовыми наборами других команд выполняются в рамках процесса развертывания сервиса Order

Тесты контрактов с расчетом на потребителя обычно применяют тестирование по примеру. Взаимодействие между потребителем и провайдером определяется набором примеров, которые называются контрактами. Каждый *контракт* состоит из примеров сообщений, обмен которыми происходит во время взаимодействия. Скажем, контракт для REST API состоит из примеров HTTP-запроса и ответа. На первый взгляд может показаться, что взаимодействие лучше описывать в виде схем в формате OpenAPI или JSON. Но, как оказалось, схемы не очень подходят для написания тестов. Они могут помочь с проверкой ответа, но тест все равно должен обратиться к провайдеру с примером запроса.

Более того, потребительским тестам нужны еще и примеры ответов. Несмотря на то что основной задачей данного подхода является проверка провайдера, контракты также проверяют, соответствует ли им потребитель. Например, потребительский контракт для REST-клиента конфигурирует заглушку сервиса, которая проверяет, совпадает ли HTTP-запрос с запросом контракта, и возвращает обратно его HTTP-ответ. Тестирование обеих сторон взаимодействия позволяет убедиться в том, что потребитель и провайдер согласовали API. Позже вы увидите примеры написания

подобного рода тестов, но сначала посмотрим, как выполнять тестирование потребительских контрактов с помощью Spring Cloud Contract.

Шаблон «Тестирование контрактов на стороне потребителя»

Проверяет, может ли клиент взаимодействовать с сервисом. См. microservices.io/patterns/testing/consumer-side-contract-test.html.

Тестирование сервисов с помощью Spring Cloud Contract

Существует два популярных фреймворка для тестирования контрактов: Spring Cloud Contract (<https://spring.io/projects/spring-cloud-contract>), который позволяет тестировать потребительские контракты в приложениях, основанных на Spring, и семейство фреймворков Pact (github.com/pact-foundation) с поддержкой разных языков. Приложение FTGO использует фреймворк Spring, поэтому в данной главе я покажу, как работать со Spring Cloud Contract. Для написания контрактов эта технология предоставляет проблемно-ориентированный язык (DSL) в стиле Groovy. Каждый контракт — это конкретный пример взаимодействия между потребителем и провайдером, такой как HTTP-запрос и ответ. Код Spring Cloud Contract генерирует тесты контрактов для провайдера. Он также настраивает макеты (например, макет HTTP-сервера) для потребительских интеграционных тестов.

Представьте, к примеру, что вы работаете над API-шлюзом и хотите написать тест потребительского контракта для сервиса `Order`. Этот процесс (рис. 9.8) подразумевает взаимодействие с командами, отвечающими за этот сервис. Вы пишете контракты, которые определяют, как API-шлюз общается с сервисом `Order`. Команда сервиса `Order` использует эти контракты для тестирования своего сервиса, а вы с их помощью проверяете свой API-шлюз. Последовательность шагов приведена далее.

1. Вы пишете один или несколько контрактов (пример приведен в листинге 9.1). Каждый контракт состоит из HTTP-запроса, который API-шлюз может послать сервису `Order`, и ожидаемого HTTP-ответа. Эти контракты вы передаете команде сервиса `Order` (возможно, посредством запроса на принятие изменений в Git).
2. Команда сервиса `Order` тестирует его с помощью тестов, код которых сгенерирован из потребительских контрактов с помощью Spring Cloud Contract.
3. Команда сервиса `Order` публикует полученные контракты в репозиторий Maven.
4. Вы используете опубликованные контракты, чтобы написать тесты для API-шлюза.

Поскольку вы тестируете API-шлюз с помощью опубликованных контрактов, то можете быть уверены в его совместимости с развернутым сервисом `Order`.

Контракты — это ключевая часть стратегии тестирования. В листинге 9.1 показан пример контракта для Spring Cloud Contract. Он состоит из HTTP-запроса и HTTP-ответа.

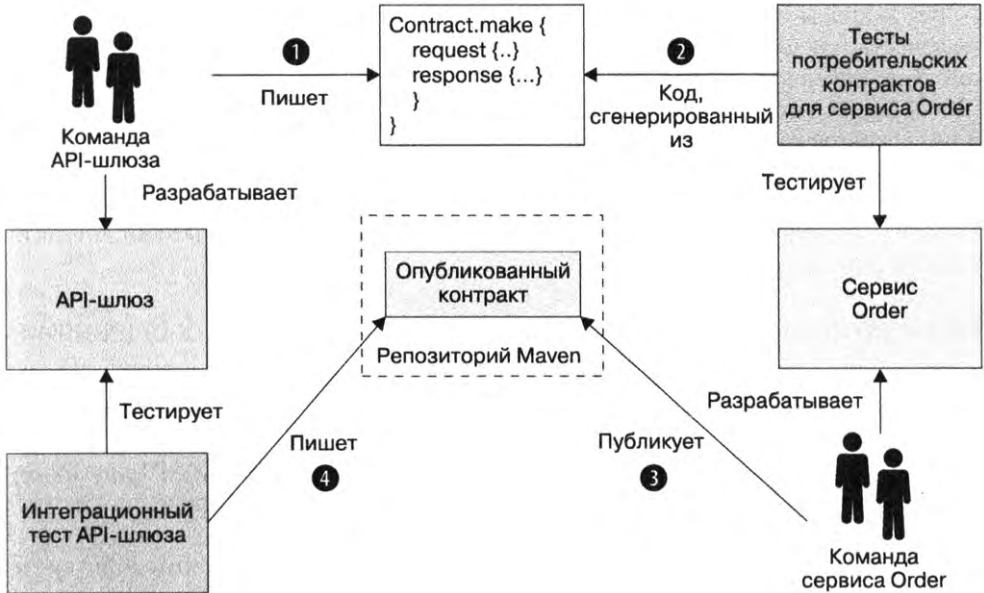


Рис. 9.8. Команда API-шлюза пишет контракты. Команда сервиса Order тестирует его с помощью этих контрактов и публикует их в репозиторий. Команда API-шлюза применяет опубликованные контракты для тестирования своего кода

Листинг 9.1. Контракт, описывающий то, как API-шлюз обращается к сервису Order

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'           ← Метод и путь HTTP-запроса
        url '/orders/1223232'
    }
    response {
        status 200             ← Код состояния, заголовки
        headers {               и тело HTTP-ответа
            header('Content-Type': 'application/json;charset=UTF-8')
        }
        body("{ ... }")
    }
}
```

Роль запрашивающего элемента играет HTTP-запрос для конечной точки REST `GET /orders/{orderId}`. Ответный элемент представляет собой HTTP-ответ, описывающий заказ, ожидаемый API-шлюзом. Контракты на языке Groovy являются частью кодовой базы провайдера. Каждая потребительская команда создает контракты, которые описывают взаимодействие ее сервиса с провайдером, и передает их команде провайдера (возможно, через запрос принятия изменений в Git). Команда провайдера упаковывает контракты в архив JAR и публикует их в репозитории Maven. Тесты на стороне потребителя загружают этот архив из репозитория.

Все запросы и ответы контракта используются не только для тестирования, но и в качестве спецификации ожидаемого поведения. В тестах на стороне потребителя контракт применяется для конфигурации заглушки, аналогичной объекту-макету в Mockito, и симулирует поведение сервиса `Order`. Это позволяет тестировать API-шлюз без запуска этого сервиса. На стороне провайдера сгенерированный тестовый класс шлет провайдеру запрос контракта и проверяет, совпадает ли его ответ с ответом контракта. Подробно о Spring Cloud Contract мы поговорим в следующей главе, а пока что посмотрим, как использовать тестирование потребительских контрактов для API обмена сообщениями.

Тесты потребительских контрактов для API обмена сообщениями

REST-клиент — это не единственный вид потребителей с определенными ожиданиями относительно API провайдера. Потребителями могут выступать также сервисы, которые подписываются на доменные события и взаимодействуют с помощью асинхронных запросов/ответов. Они обращаются к асинхронным API других сервисов, делая предположения о природе этих API. Для них тоже нужно писать тесты потребительских контрактов.

Spring Cloud Contract также поддерживает тестирование взаимодействия на основе обмена сообщениями. Структура контракта и то, как он применяется в тестах, зависит от типа взаимодействия. Контракт для публикации доменных событий состоит из примера доменного события. В ходе тестирования провайдер генерирует событие и проверяет, совпадает ли оно с событием контракта. Потребительский тест проверяет, может ли потребитель обработать событие. Пример такого теста будет представлен в следующей главе.

Контракт для асинхронного взаимодействия в стиле «запрос/ответ» похож на HTTP-контракты. Он состоит из двух сообщений: с запросом и ответом. Тест провайдера шлет запрос контракта интерфейсу (API) и проверяет, совпадает ли полученный ответ с ответом контракта. Потребительский тест использует контракт, чтобы сконфигурировать заглушку для подписчика, которая перехватывает запрос контракта и возвращает указанный ответ. Пример такого теста описывается в следующей главе. Здесь же мы рассмотрим процесс развертывания, в рамках которого выполняются эти и другие тесты.

9.1.3. Процесс развертывания

У каждого сервиса есть свой процесс развертывания. В книге Джеза Хамбла (Jez Humble) *Continuous Delivery* (Addison-Wesley, 2010)¹ процесс развертывания описывается как автоматическая доставка кода из компьютера разработчика в промышленную среду. Он состоит из поэтапного выполнения тестов, вслед за которым происходит выпуск или развертывание сервиса (рис. 9.9). В идеале этот процесс должен быть полностью автоматизированным, но в реальности он может требовать ручного вмешательства. Процесс развертывания часто реализуется с помощью CI-сервера (Continuous Integration — непрерывное развертывание), такого как Jenkins.

¹ Хамбл Д. Непрерывное развертывание. — М.: Вильямс, 2011.

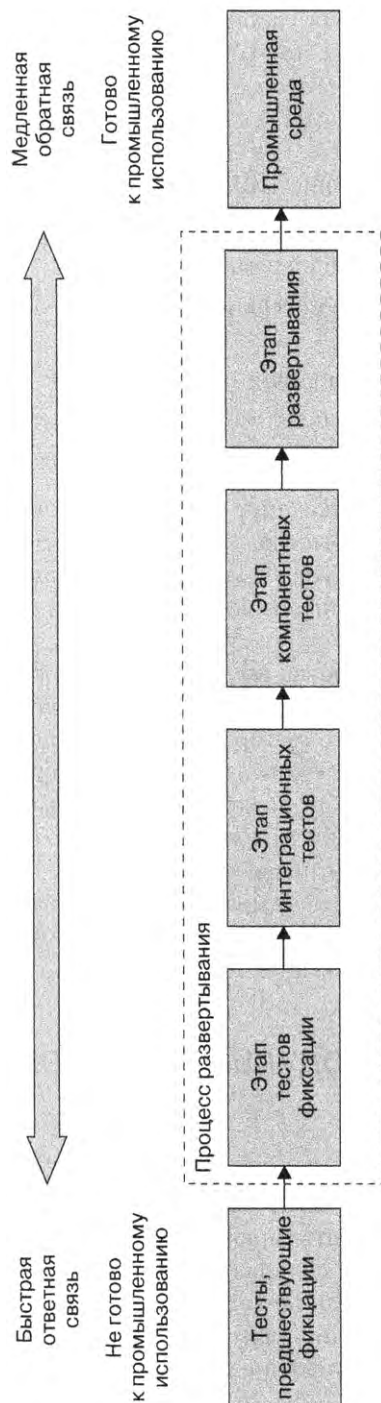


Рис. 9.9. Пример процесса развертывания для сервиса Order. Он состоит из последовательных этапов. Тесты, предшествующие фиксации, разработчик выполняет перед фиксацией кода. Остальные этапы реализует автоматизированная система, такая как CI-сервер Jenkins

По мере прохождения кода через процесс развертывания наборы тестов все более тщательно тестируют его в среде, приближенной к промышленной. Одновременно время выполнения каждого тестового набора обычно увеличивается. Основной смысл процедуры состоит в том, чтобы как можно скорее сообщить о непройденных тестах.

Процесс развертывания (см. рис. 9.9) состоит из следующих этапов.

- ❑ *Этап, предшествующий фиксации*, — выполняет модульные тесты. Запускается разработчиком перед фиксацией изменений.
- ❑ *Этап фиксации* — компилирует сервис, выполняет модульные тесты и производит статический анализ кода.
- ❑ *Интеграционный этап* — выполняет интеграционные тесты.
- ❑ *Компонентный этап* — выполняет компонентные тесты для сервиса.
- ❑ *Этап развертывания* — развертывает сервис в промышленной среде.

CI-сервер запускает этап фиксации, когда разработчик фиксирует изменение. Он выполняется чрезвычайно быстро, чтобы сразу предоставить сведения о фиксации. Дальнейшие этапы протекают дольше и предоставляют информацию не так быстро. Если все тесты пройдены, на заключительном этапе код развертывается в промышленную среду.

В этом примере автоматизирован весь процесс, от фиксации до развертывания. Однако в некоторых ситуациях требуется вмешательство человека. Например, вам может понадобиться этап ручного тестирования в предпроектной среде. В этом сценарии код переходит на следующий этап, когда тестировщик отмечает успешное тестирование нажатием кнопки. Это может быть также выпуск новой версии сервиса в рамках процесса развертывания. Позже выпущенные сервисы будут упакованы и в качестве готового продукта отправлены заказчиком.

Теперь вы знаете, как организован процесс развертывания и на каких этапах выполняются разные типы тестов. Переместимся в самый низ пирамиды тестирования и посмотрим, как пишут модульные тесты для сервиса.

9.2. Написание модульных тестов для сервиса

Представьте, что вам нужно написать тест, который проверяет, вычисляет ли сервис `Order` приложения FTGO корректную промежуточную стоимость заказа. Код вашего теста может запустить сервис `Order`, обратиться к его REST API, чтобы создать заказ, и проверить, содержит ли HTTP-ответ ожидаемые значения. Однако при этом тест получится не только сложным, но и медленным. Если он выполняется на этапе компиляции класса `Order`, вы будете тратить много времени в ожидании его завершения. Написание модульных тестов для класса `Order` — куда более продуктивный подход.

Как видно на рис. 9.10, модульные тесты находятся на самом нижнем уровне пирамиды тестирования. Они ориентированы на технологии и могут использоваться в разработке. Модульный тест позволяет убедиться в корректной работе *модуля*, который представляет собой очень маленькую часть сервиса. Обычно в качестве модуля выступает класс, поэтому модульное тестирование проверяет, ведет ли он себя так, как от него ожидается.

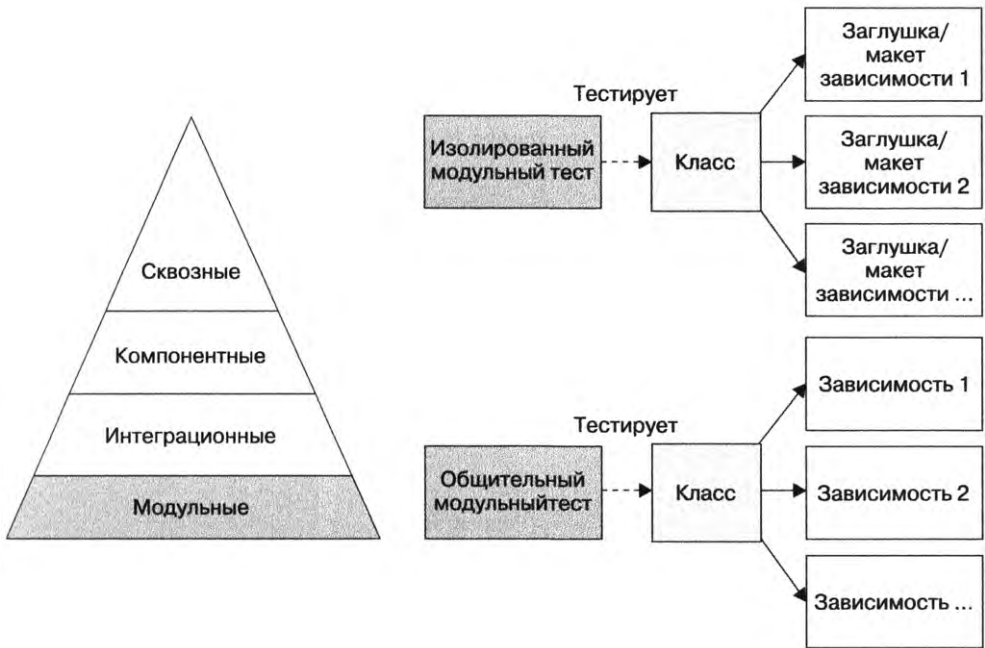


Рис. 9.10. Модульные тесты лежат в основании пирамиды. Они быстрые, простые в написании и надежные. Изолированный модульный тест тестирует отдельно взятый класс, задействуя макеты и заглушки вместо его зависимостей. Общительный модульный тест тестирует класс вместе с его зависимостями

Существует два типа модульных тестов (martinfowler.com/bliki/UnitTest.html):

- ❑ *изолированный* — тестирует отдельно взятый класс, заменяя его зависимости объектами-макетами;
- ❑ *общительный* — тестирует класс и его зависимости.

Тип теста, который следует использовать, зависит от назначения класса и его роли в архитектуре. Шестигранная архитектура типичного сервиса и типы модульных тестов, применяемые для определенного вида классов, показаны на рис. 9.11. Классы контроллеров и сервиса обычно тестируются изолированно. Доменные объекты, такие как сущности и объекты значений, чаще всего тестируются с помощью общительных модульных тестов.

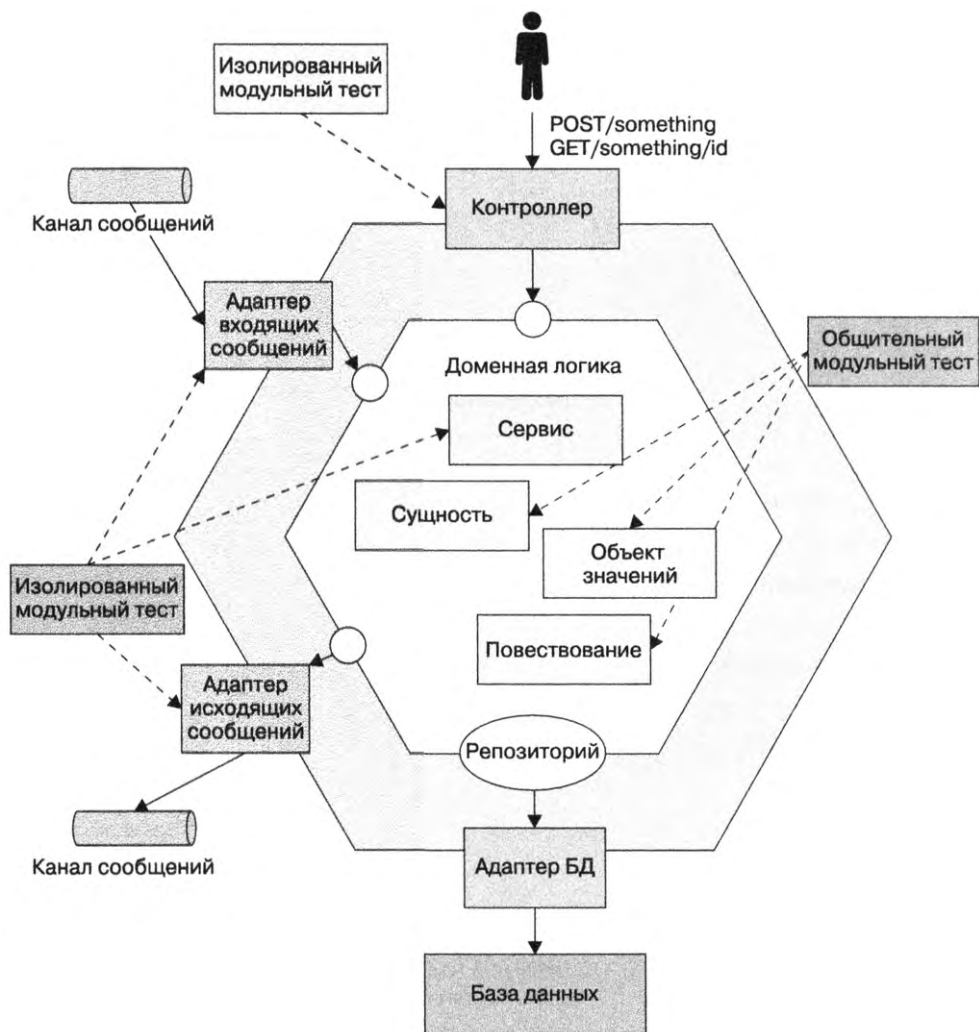


Рис. 9.11. Назначение класса определяет, какие модульные тесты нужно использовать — изолированные или общительные

Типичная стратегия тестирования класса выглядит так.

- ❑ Такие сущности, как **Order**, которые обладают постоянными идентификаторами (см. главу 5), тестируются с помощью общительных модульных тестов.
- ❑ Такие объекты, как **Money**, представляющие собой набор значений (см главу 5), тестируются с применением общительных модульных тестов.
- ❑ Повествования наподобие **CreateOrderSaga**, которые обеспечивают согласованность данных между сервисами (см. главу 4), тестируются общительными модульными тестами.

- ❑ Доменные сервисы, такие как `OrderService`, реализующие бизнес-логику, которая не подходит для сущностей или объектов значений (см. главу 5), тестируются с помощью изолированных модульных тестов.
- ❑ Контроллеры, обрабатывающие HTTP-запросы, такие как `OrderController`, тестируются с использованием изолированных модульных тестов.
- ❑ Шлюзы для входящих и исходящих сообщений тестируются с помощью изолированных модульных тестов.

Для начала посмотрим, как тестируются доменные сущности.

9.2.1. Разработка модульных тестов для доменных сущностей

В листинге 9.2 показан фрагмент класса `OrderTest`, реализующий модульные тесты для сущности `Order`. Этот класс содержит метод `@Before setUp()`, который создает объект `Order` перед запуском каждого теста. Методы, помеченные как `@Test`, могут выполнить инициализацию объекта `Order`, вызвать один из его методов и затем сделать утверждение о его возвращаемом значении и состоянии.

Листинг 9.2. Простой и быстрый модульный тест для сущности `Order`

```
public class OrderTest {

    private ResultWithEvents<Order> createResult;
    private Order order;

    @Before
    public void setUp() throws Exception {
        createResult = Order.createOrder(CONSUMER_ID, AJANTA_ID, CHICKEN_VINDALOO
            _LINE_ITEMS);
        order = createResult.result;
    }

    @Test
    public void shouldCalculateTotal() {
        assertEquals(CHICKEN_VINDALOO_PRICE.multiply(CHICKEN_VINDALOO_QUANTITY),
            order.getOrderTotal());
    }

    ...
}
```

Метод `@Test shouldCalculateTotal()` проверяет, возвращает ли `Order.getOrderTotal()` ожидаемое значение. Модульные тесты тщательно тестируют бизнес-логику. Они являются общительными и распространяются не только на класс `Order`, но и на его зависимости. Вы можете использовать их на этапе компиляции, так как выполняются они чрезвычайно быстро. Также важно протестировать объект значений `Money`, от которого зависит класс `Order`. Посмотрим, как это делается.

9.2.2. Написание модульных тестов для объектов значений

Объекты значений не изменяются, поэтому их обычно легко тестировать. Вам не нужно беспокоиться о побочных эффектах. Как правило, тест должен создать объект значений в определенном состоянии, вызвать один из его методов и сделать вывод о возвращаемом значении. В листинге 9.3 приводятся тесты для объекта `Money` — простого класса, который представляет денежное значение. Эти тесты проверяют поведение методов класса `Money`, включая `add()`, который складывает два экземпляра `Money`, и `multiply()`, который умножает объект `Money` на целое число. Эти тесты изолированы, поскольку класс `Money` не зависит ни от каких других классов приложения.

Листинг 9.3. Простой и быстрый тест для объекта значений `Money`

```
public class MoneyTest {

    private final int M1_AMOUNT = 10;
    private final int M2_AMOUNT = 15;

    private Money m1 = new Money(M1_AMOUNT);
    private Money m2 = new Money(M2_AMOUNT);

    @Test
    public void shouldAdd() {
        assertEquals(new Money(M1_AMOUNT + M2_AMOUNT), m1.add(m2));
    }

    @Test
    public void shouldMultiply() {
        int multiplier = 12;
        assertEquals(new Money(M2_AMOUNT * multiplier), m2.multiply(multiplier));
    }

    ...
}
```

Проверяем, можно ли сложить вместе два объекта `Money`

Проверяем, можно ли умножить объект `Money` на целое число

Доменные сущности и объекты значений — это кирпичики, из которых состоит бизнес-логика сервиса. Но иногда она может находиться также в повествованиях и других сервисах приложения. Посмотрим, как их тестировать.

9.2.3. Разработка модульных тестов для повествований

Повествования, такие как класс `CreateOrderSaga`, реализуют важную бизнес-логику, поэтому их тоже нужно тестировать. В данном случае мы имеем дело с сохраняемым объектом, которые рассылает командные сообщения участникам повествования и обрабатывает их ответы. Как говорилось в главе 4, класс `CreateOrderSaga` обменивается командными/ответными сообщениями с несколькими сервисами, включая `Consumer`

и `Kitchen`. Тест для этого класса создает повествование и проверяет, шлет ли тот ожидаемую последовательность сообщений своим участникам. Один из тестов должен быть написан для оптимистичного сценария. Но вы должны предусмотреть и тесты для различных случаев, когда повествование откатывается, получив сообщение об отказе от одного из своих участников.

Вы можете написать тесты, которые используют настоящие базу данных и брокер сообщений, но заменяют участников повествования заглушками. Например, заглушка для сервиса `Consumer` будет подписываться на командный канал `consumerService` и отправлять обратно сообщение с нужным ответом. Однако тесты, написанные таким образом, будут довольно медленными. Куда более эффективный подход заключается в применении макетов место брокера сообщений и классов для взаимодействия с базой данных. Это позволит вам сосредоточиться на тестировании основных функций повествования.

В листинге 9.4 показан тест для `CreateOrderSaga`. Это общительный модульный тест, который проверяет класс повествования и его зависимости. Он написан с использованием фреймворка тестирования `Eventuate Tram Saga` (github.com/eventuate-tram/eventuate-tram-sagas), который предоставляет простой в применении язык DSL, абстрагирующий подробности взаимодействия с повествованиями. С помощью этого языка вы можете создать повествование и убедиться в том, что оно отправляет корректные командные сообщения. При этом фреймворк тестирования подготавливает макеты для базы данных и инфраструктуры обмена сообщениями.

Листинг 9.4. Простой и быстрый модульный тест для `CreateOrderSaga`

```
public class CreateOrderSagaTest {

    @Test
    public void shouldCreateOrder() {
        given()
            .saga(new CreateOrderSaga(kitchenServiceProxy),
                new CreateOrderSagaState(ORDER_ID,
                    CHICKEN_VINDALOO_ORDER_DETAILS)).
        expect().
            command(new ValidateOrderByConsumer(CONSUMER_ID, ORDER_ID,
                CHICKEN_VINDALOO_ORDER_TOTAL)).
            to(ConsumerServiceChannels.consumerServiceChannel).
        andGiven().
            successReply().
        expect().
            command(new CreateTicket(AJANTA_ID, ORDER_ID, null)).
            to(KitchenServiceChannels.kitchenServiceChannel);
    }

    @Test
    public void shouldRejectOrderDueToConsumerVerificationFailed() {
        given()
            .saga(new CreateOrderSaga(kitchenServiceProxy),
                new CreateOrderSagaState(ORDER_ID,
                    CHICKEN_VINDALOO_ORDER_DETAILS)).
```

Создаем повествование

Проверяем, шлет ли оно сообщение `ValidateOrderByConsumer` сервису `Consumer`

Успешно отвечаем на это сообщение

Проверяем, шлет ли оно сообщение `CreateTicket` сервису `Kitchen`

```

expect().
    command(new ValidateOrderByConsumer(CONSUMER_ID, ORDER_ID,
        CHICKEN_VINDALOO_ORDER_TOTAL)).
    to(ConsumerServiceChannels.consumerServiceChannel).
andGiven().
    failureReply().
expect().
    command(new RejectOrderCommand(ORDER_ID)).
    to(OrderServiceChannels.orderServiceChannel);
}
}

```

Возвращаем отрицательный ответ:
сервис Consumer отклонил заказ

Проверяем, шлет ли повествование
сообщение RejectOrderCommand сервису Order

Метод `@Test shouldCreateOrder()` тестирует оптимистичный сценарий. Метод `@Test shouldRejectOrderDueToConsumerVerificationFailed()` тестирует случай, когда сервис Consumer отклоняет заказ. Он проверяет, шлет ли `CreateOrderSaga` команду `RejectOrderCommand`, чтобы компенсировать отклонение действий клиента. Класс `CreateOrderSagaTest` содержит методы для тестирования и других отрицательных сценариев.

Теперь посмотрим, как тестировать доменные сервисы.

9.2.4. Написание модульных тестов для доменных сервисов

Большая часть бизнес-логики сервиса реализуется его доменными сущностями, объектами значений и повествованиями. Остальное содержится в таких классах, как `OrderService`. Это типичный класс доменного сервиса. Его методы вызывают сущности и репозитории и публикуют доменные события. Чтобы его эффективно протестировать, нужно использовать в основном изолированные модульные тесты, которые предоставляют макеты для таких зависимостей, как репозитории и классы обмена сообщениями.

В листинге 9.5 представлен класс `OrderServiceTest`, который тестирует `OrderService`. Он определяет изолированные модульные тесты, заменяющие зависимости сервиса макетами из состава Mockito. Каждый тест состоит из следующих этапов.

1. *Подготовка* — конфигурирует объекты-макеты для зависимостей сервиса.
2. *Выполнение* — вызывает метод сервиса.
3. *Проверка* — проверяет корректность значения, возвращенного методом сервиса, и убеждается в том, что зависимости были вызваны правильно.

Метод `setUp()` создает экземпляр `OrderService` с внедренными макетами зависимостей. Метод `@Test shouldCreateOrder()` проверяет, обратился ли вызов `OrderService.createOrder()` к `OrderRepository`, чтобы сохранить только что созданный заказ, опубликовал ли он событие `OrderCreatedEvent` и создал ли `CreateOrderSaga`.

Листинг 9.5. Простой и быстрый модульный тест для класса `OrderService`

```

public class OrderServiceTest {

    private OrderService orderService;
    private OrderRepository orderRepository;
    private DomainEventPublisher eventPublisher;
    private RestaurantRepository restaurantRepository;
    private SagaManager<CreateOrderSagaState> createOrderSagaManager;
    private SagaManager<CancelOrderSagaData> cancelOrderSagaManager;
    private SagaManager<ReviseOrderSagaData> reviseOrderSagaManager;

    @Before
    public void setup() {
        orderRepository = mock(OrderRepository.class);
        eventPublisher = mock(DomainEventPublisher.class);
        restaurantRepository = mock(RestaurantRepository.class);
        createOrderSagaManager = mock(SagaManager.class);
        cancelOrderSagaManager = mock(SagaManager.class);
        reviseOrderSagaManager = mock(SagaManager.class);
        orderService = new OrderService(orderRepository, eventPublisher,
            restaurantRepository, createOrderSagaManager,
            cancelOrderSagaManager, reviseOrderSagaManager);
    }

    @Test
    public void shouldCreateOrder() {
        when(restaurantRepository
            .findById(AJANTA_ID)).thenReturn(Optional.of(AJANTA_RESTAURANT_));
        when(orderRepository.save(any(Order.class))).then(invocation -> {
            Order order = (Order) invocation.getArguments()[0];
            order.setId(ORDER_ID);
            return order;
        });

        Order order = orderService.createOrder(CONSUMER_ID,
            AJANTA_ID, CHICKEN_VINDALOO_MENU_ITEMS_AND_QUANTITIES);

        verify(orderRepository).save(same(order));

        verify(eventPublisher).publish(Order.class, ORDER_ID,
            singletonList(
                new OrderCreatedEvent(CHICKEN_VINDALOO_ORDER_DETAILS)));

        verify(createOrderSagaManager)
            .create(new CreateOrderSagaState(ORDER_ID,
                CHICKEN_VINDALOO_ORDER_DETAILS),
                Order.class, ORDER_ID);
    }
}

```

Создаем макеты Mockito для зависимостей класса `OrderService`

Создаем экземпляр `OrderService` с внедренными макетами зависимостей

Делаем так, чтобы метод `RestaurantRepository.findById()` вернул ресторан Ajanta

Делаем так, чтобы метод `OrderRepository.save()` сохранил ID заказа

Вызываем `OrderService.create()`

Проверяем, сохранил ли класс `OrderService` только что созданный заказ в БД

Проверяем, опубликовал ли класс `OrderService` событие `OrderCreatedEvent`

Проверяем, создал ли класс `OrderCreatedEvent` повествование `CreateOrderSaga`

Итак, мы обсудили то, как выполнить модульное тестирование классов бизнес-логики. Теперь поговорим о том, как сделать то же самое с адаптерами, которые взаимодействуют с внешними системами.

9.2.5. Разработка модульных тестов для контроллеров

Сервисы, такие как `Order`, обычно содержат один или несколько контроллеров для обработки HTTP-запросов от других сервисов и API-шлюза. Класс контроллера состоит из набора методов, обрабатывающих запросы. Каждый такой метод реализует конечную точку REST API, а его параметры представляют значения HTTP-запроса, такие как переменные пути. Обычно он обращается к доменному сервису или репозиторию и возвращает объект с ответом. `OrderController`, к примеру, обращается к `OrderService` и `OrderRepository`. Эффективная стратегия тестирования контроллеров предполагает использование изолированных модульных тестов, которые заменяют макетами сервисы и репозитории.

Вы могли бы написать класс вроде `OrderServiceTest`, который создает экземпляры контроллера и вызывает его методы. Однако такой подход не позволяет проверить некоторые важные возможности, такие как маршрутизация запросов. Куда более эффективным будет применение фреймворков для тестирования в стиле MVC, например `Spring MockMvc`, который входит в состав `Spring Framework`, или `Rest Assured Mock MVC`, основанный на `Spring MockMvc`. Тесты, написанные с помощью одной из этих технологий, выполняют нечто похожее на HTTP-запрос и делают заключение об HTTP-ответах. Эти фреймворки позволяют тестировать маршрутизацию HTTP-запросов и преобразование объектов Java в формат JSON и обратно, избегая при этом реальных сетевых вызовов. `Spring MockMvc` автоматически создает экземпляры ровно того количества классов `Spring MVC`, которого должно хватить для тестирования.

Действительно ли это модульные тесты?

Эти тесты используют `Spring Framework`, поэтому можно предположить, что они не модульные. Действительно, они тяжеловесней тех, что я описывал ранее. В документации `Spring MockMvc` они называются внеконтейнерными интеграционными тестами (docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#spring-mvc-test-vs-end-to-end-integration-tests). В то же время в `Rest Assured Mock MVC` они считаются модульными (github.com/rest-assured/rest-assured/wiki/Usage#spring-mock-mvc-module). Несмотря на такое различие в терминологии, написание этих тестов очень важно.

В листинге 9.6 показан класс `OrderControllerTest`, тестирующий `OrderController` сервиса `Order`. Он определяет изолированные модульные тесты, использующие

макеты вместо зависимостей `OrderController`. Этот класс написан с помощью фреймворка `Rest Assured Mock MVC`, который предоставляет простой язык DSL, абстрагирующий подробности взаимодействия с контроллерами. `Rest Assured` упрощает отправку контроллеру поддельных HTTP-запросов и проверку ответов. `OrderControllerTest` создает контроллер с внедренными макетами `Mockito` для `OrderService` и `OrderRepository`. Каждый тест конфигурирует макеты, выполняет HTTP-запрос, проверяет корректность ответа и, возможно, следит за тем, чтобы контроллер обратился к этим макетам.

Листинг 9.6. Простой и быстрый модульный тест для класса `OrderController`

```
public class OrderControllerTest {

    private OrderService orderService;
    private OrderRepository orderRepository;

    @Before
    public void setUp() throws Exception {
        orderService = mock(OrderService.class);
        orderRepository = mock(OrderRepository.class);
        orderController = new OrderController(orderService, orderRepository);
    }

    @Test
    public void shouldFindOrder() {
        when(orderRepository.findById(1L))
            .thenReturn(Optional.of(CHICKEN_VINDALOO_ORDER_));

        given().
            standaloneSetup(configureControllers(
                new OrderController(orderService, orderRepository))).
        when().
            get("/orders/1").
        then().
            statusCode(200).
            body("orderId",
                equalTo(new Long(OrderDetailsMother.ORDER_ID).intValue())).
            body("state",
                equalTo(OrderDetailsMother.CHICKEN_VINDALOO_ORDER_STATE.name())).
            body("orderTotal",
                equalTo(CHICKEN_VINDALOO_ORDER_TOTAL.asString()))
        ;
    }

    @Test
    public void shouldFindNotOrder() { ... }

    private StandaloneMockMvcBuilder controllers(Object... controllers) { ... }
}
```

Создаем макеты зависимостей класса `OrderController`

Делаем так, чтобы макет `OrderRepository` возвращал заказ

Конфигурируем `OrderController`

Выполняем HTTP-запрос

Проверяем код состояния ответа

Проверяем элементы тела ответа в формате JSON

Первым делом тестовый метод `shouldFindOrder()` конфигурирует макет `OrderRepository` так, чтобы тот возвращал `Order`. Затем он выполняет HTTP-запрос, чтобы извлечь заказ. В конце проверяет успешность запроса и то, содержит ли тело ответа ожидаемые данные.

Контроллеры — это не единственный вид адаптеров, обрабатывающих запросы из внешних систем. Есть также обработчики событий/сообщений. Посмотрим, как выполняется модульное тестирование для них.

9.2.6. Написание модульных тестов для обработчиков событий и сообщений

Сервисы часто обрабатывают сообщения, переданные внешними системами. К примеру, сервис `Order` содержит адаптер сообщений `OrderEventConsumer`, обрабатывающий доменные события, публикуемые другими сервисами. Как и контроллеры, адаптеры сообщений обычно представляют собой простые классы, которые обращаются к доменным сервисам. Каждый метод адаптера, как правило, вызывает метод сервиса, передавая ему данные из сообщения или события.

Для модульного тестирования адаптеров сообщений можно применить подход, аналогичный используемому для контроллеров. Каждый тест создает экземпляр адаптера, отправляет сообщение в канал и проверяет корректность обращения к макету сервиса. Одновременно автоматически создаются заглушки для инфраструктуры обмена сообщениями, поэтому брокер не участвует в этом процессе. Посмотрим, как протестировать класс `OrderEventConsumer`.

В листинге 9.7 показан фрагмент класса `OrderEventConsumerTest`, который тестирует адаптер `OrderEventConsumer`. Он проверяет, направляет ли тот каждое событие подходящему методу-обработчику, и следит за корректностью обращения к `OrderService`. Здесь применяется фреймворк `Eventuate Tram Mock Messaging`, который предоставляет простой в использовании язык DSL для создания макетов инфраструктуры обмена сообщениями. Он имеет тот же формат тестов «дано — когда — тогда», что и `Rest Assured`. Каждый тест создает экземпляр `OrderEventConsumer` с внедренным макетом `OrderService`, публикует доменное событие и проверяет, корректно ли `OrderEventConsumer` обращается к макету сервиса.

Метод `setUp()` создает экземпляр `OrderEventConsumer` с внедренным макетом `OrderService`. Метод `shouldCreateMenu()` публикует событие `RestaurantCreated` и проверяет, вызвал ли объект `OrderEventConsumer` метод `OrderService.createMenu()`. `OrderEventConsumerTest`, как и другие классы для модульного тестирования, выполняется очень быстро. Модульные тесты работают всего несколько секунд.

Однако эти тесты не проверяют, насколько корректно сервис наподобие `Order` взаимодействует с другими сервисами. Например, они не позволяют убедиться в том, что заказ можно сохранить в `MySQL` или что `CreateOrderSaga` шлет командные сообщения в правильном формате и в нужный канал. И с их помощью нельзя узнать, имеет ли событие `RestaurantCreated`, обработанное адаптером `OrderEventConsumer`, ту же структуру, что и события, публикуемые сервисом `Restaurant`. Для тестирования корректности взаимодействия между сервисами необходимо писать интеграци-

онные тесты. Также понадобятся компонентные тесты, которые тестируют отдельно взятый сервис целиком. Все это, а также выполнение сквозного тестирования, мы обсудим в следующей главе.

Листинг 9.7. Быстрый модульный тест для класса `OrderEventConsumer`

```
public class OrderEventConsumerTest {

    private OrderService orderService;
    private OrderEventConsumer orderEventConsumer;

    @Before
    public void setUp() throws Exception {
        orderService = mock(OrderService.class);
        orderEventConsumer = new OrderEventConsumer(orderService);
    }

    @Test
    public void shouldCreateMenu() {
        given().
            eventHandlers(orderEventConsumer.domainEventHandlers()).
        when().
            aggregate("net.chrisrichardson.ftgo.restaurant.service.domain.Restaurant",
                AJANTA_ID).
            publishes(new RestaurantCreated(AJANTA_RESTAURANT_NAME,
                RestaurantMother.AJANTA_RESTAURANT_MENU))
        then().
            verify(() -> {
                verify(orderService)
                    .createMenu(AJANTA_ID,
                        new RestaurantMenu(RestaurantMother.AJANTA_RESTAURANT_MENU_ITEMS));
            })
        ;
    }
}
```

Создаем экземпляр `OrderEventConsumer` с поддельными зависимостями

Конфигурируем доменные обработчики `OrderEventConsumer`

Публикуем событие `RestaurantCreated`

Проверяем, вызвал ли экземпляр `OrderEventConsumer` метод `OrderService.createMenu()`

Резюме

- ❑ Автоматическое тестирование лежит в основе быстрой и безопасной доставки программного обеспечения. К тому же микросервисная архитектура сложна по своей природе, поэтому, чтобы воспользоваться всеми ее преимуществами, вы *должны* автоматизировать свои тесты.
- ❑ Тест нужен для того, чтобы проверить поведение тестируемой системы. В данном случае под *системой* понимается тестируемый элемент программного обеспечения. Это может быть как отдельный класс, так и приложение целиком. Или же что-то среднее, например коллекция классов или отдельный сервис. Связанные между собой тесты объединяются в тестовый набор.

- ❑ Хороший способ упрощения и ускорения тестов — задействование дублеров. Дублер — это объект, который симулирует поведение зависимости тестируемой системы. Существует два типа дублеров: заглушки и макеты. Заглушка возвращает значение тестируемой системе. Макет используется тестом для проверки, корректно ли система вызывает свои зависимости.
- ❑ Применяйте пирамиду тестов, чтобы определить, где следует приложить усилия при тестировании сервисов. Большинство ваших тестов должны быть модульными, то есть быстрыми, надежными и простыми в написании. Вы должны минимизировать количество сквозных тестов, поскольку они медленные и нестабильные, а их написание занимает много времени.

Тестирование микросервисов, часть 2

В этой главе

- Методики изолированного тестирования сервисов.
- Использование тестирования контрактов с расчетом на потребителя для написания тестов, которые быстро, но в то же время надежно проверяют межсервисное взаимодействие.
- Когда и как выполнять сквозное тестирование приложений.

Здесь мы будем отталкиваться от предыдущей главы, в которой были представлены концепции тестирования, включая пирамиду тестов. *Пирамида тестов* описывает относительные пропорции разных типов тестирования, которые вы должны применять. Из предыдущей главы вы узнали, как писать модульные тесты, лежащие в основе этой пирамиды. Продолжим восхождение к ее вершине.

Эта глава начинается с того, как писать интеграционные тесты, которые находятся на уровень выше модульных. *Интеграционные тесты* проверяют корректность взаимодействия сервиса с инфраструктурой, включая базы данных и другие сервисы приложения. Вслед за этим мы обсудим *компонентные тесты*, которые являются формой приемочного тестирования сервисов. Компонентный тест проверяет работу сервиса в изоляции, используя заглушки вместо его зависимостей. Дальше вы узнаете, как писать сквозные тесты, которые тестируют группу сервисов или целое приложение. Сквозные тесты находятся на вершине пирамиды, поэтому применять их следует как можно реже.

Начнем с рассмотрения интеграционных тестов.

10.1. Написание интеграционных тестов

Обычно сервисы взаимодействуют друг с другом. Например, сервис `Order` общается с несколькими другими сервисами (рис. 10.1). Его REST API потребляется API-шлюзом, а доменные события обрабатываются такими сервисами, как `Order History`. Последний, в свою очередь, использует еще несколько компонентов приложения: сохраняет заказы в `MySQL` и обменивается командами/ответами с несколькими другими сервисами, такими как `Kitchen`.

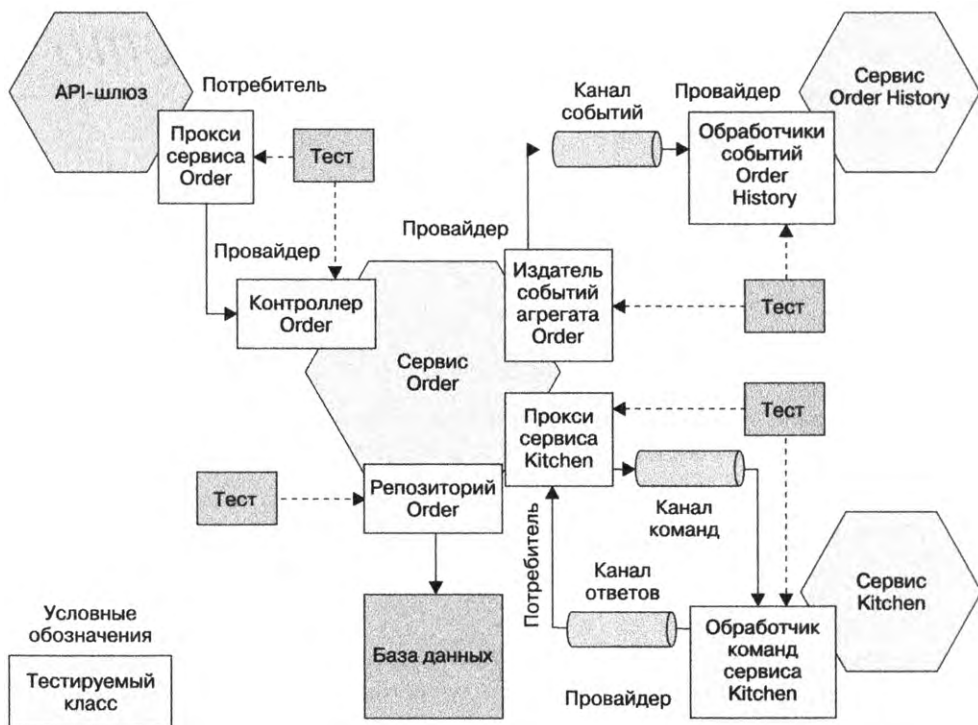


Рис. 10.1. Интеграционные тесты должны проверить, может ли сервис взаимодействовать со своими клиентами и зависимостями. Но вместо тестирования целых сервисов проверяются классы отдельных адаптеров, которые реализуют взаимодействие

Чтобы удостовериться в том, что сервис `Order` ведет себя как положено, мы должны написать тесты, проверяющие его способность корректно взаимодействовать с инфраструктурой и другими сервисами приложения. Для этого можно запустить все сервисы и протестировать их API. Подобного рода тесты называют сквозными, они работают медленно, склонны к ошибкам и требуют больших затрат. Как вы увидите в разделе 10.3, такой вид тестирования тоже имеет право на жизнь, но он находится на самой вершине пирамиды тестов, поэтому его использование следует минимизировать.

Гораздо более эффективная стратегия — написание так называемых интеграционных тестов. В пирамиде тестирования интеграционные тесты находятся на уровень выше модульных (рис. 10.2). Они помогают убедиться в том, что сервис как следует взаимодействует с инфраструктурой и другими сервисами. Но, в отличие от сквозных, они эти сервисы не запускают. Вместо этого задействуются две стратегии, которые существенно упрощают код тестов, не влияя на их эффективность.



Рис. 10.2. Интеграционные тесты находятся на уровень выше модульных. Они проверяют, может ли сервис взаимодействовать со своими зависимостями, включая инфраструктурные компоненты наподобие базы данных и другие сервисы приложения

Первая стратегия заключается в тестировании каждого адаптера сервиса, возможно, вместе со вспомогательными классами. Например, в подразделе 10.1.1 будет представлен тест, который проверяет корректность сохранения заказов с помощью JPA. Вместо использования API сервиса `Order` он напрямую тестирует класс `OrderRepository`. А в подразделе 10.1.3 вы увидите тест, который, работая с классом `OrderDomainEventPublisher`, проверяет корректность структуры событий, публикуемых сервисом `Order`. Если сосредоточиться на небольшом количестве классов, а не на сервисе в целом, ваши тесты получатся куда более простыми и быстрыми.

Вторая стратегия упрощения интеграционных тестов, которые проверяют межсервисное взаимодействие, состоит в использовании контрактов (см. главу 9). *Контракт* — это конкретный пример взаимодействия между двумя сервисами. Как показано в табл. 10.1, структура контракта зависит от того, как именно сервисы общаются между собой.

Таблица 10.1. Структура контракта зависит от типа взаимодействия между сервисами

Стиль взаимодействия	Потребитель	Провайдер	Контракт
Запросы/ответы на основе REST	API-шлюз	Сервис Order	HTTP-запрос и ответ
Издатель/подписчик	Сервис Order History	Сервис Order	Доменное событие
Асинхронные запросы/ответы	Сервис Order	Сервис Kitchen	Командное и ответное сообщения

Контракт состоит из одного или двух сообщений. Первый случай подходит для взаимодействия в стиле «издатель/подписчик», а второй — для асинхронных запросов/ответов.

Контракты используются для тестирования как потребителя, так и провайдера. Это позволяет убедиться в том, что их API согласованы. Структура контракта варьируется в зависимости от того, какую сторону мы тестируем.

- ❑ *Тесты на стороне потребителя* — тесты для потребительского адаптера. Контракты в них применяются для конфигурации заглушек, симулирующих работу провайдера. Благодаря этому вам не нужен запущенный провайдер, чтобы протестировать потребителя.
- ❑ *Тесты на стороне провайдера* — проверяют адаптер провайдера. Они используют контракты для тестирования адаптеров, заменяя их зависимости макетами.

Позже в этом разделе я покажу примеры такого рода тестов, но сначала рассмотрим тестирование с сохранением.

10.1.1. Интеграционные тесты с сохранением

Обычно сервисы хранят информацию в базе данных. Например, `OrderService` хранит агрегаты, такие как `Order`, в MySQL, используя при этом JPA. Точно так же CQRS-представление сервиса `Order History` размещается в AWS DynamoDB. Модульные тесты, которые мы написали ранее, проверяют только те объекты, которые находятся в памяти. Чтобы убедиться в корректной работе сервиса, мы должны написать интеграционные тесты с сохранением, которые проверяют, работает ли логика доступа к базе данных так, как мы того ожидаем. В случае с сервисом `Order` это означает тестирование JPA-репозитория, таких как `OrderRepository`.

Этапы интеграционного теста с сохранением ведут себя следующим образом.

- ❑ *Подготовка* — подготавливает базу данных, создавая ее схему и приводя ее к известному состоянию. Этот этап также может инициировать транзакцию.
- ❑ *Выполнение* — выполняет операцию с базой данных.
- ❑ *Проверка* — делает вывод о состоянии базы данных и извлеченных из нее объектов.
- ❑ *Очистка* — опциональный этап, который может отменить изменения, внесенные в базу данных, например, путем отката транзакции, инициированной на этапе подготовки.

В листинге 10.1 показан интеграционный тест с сохранением для агрегата `Order` и `OrderRepository`. Помимо использования JPA для создания схемы базы данных, подобные тесты не делают никаких предположений о состоянии БД. Таким образом, им не нужно откатывать изменения, внесенные в базу данных. Это позволяет избежать проблем, связанных с тем, что ORM кэширует измененные данные в память.

Листинг 10.1. Интеграционный тест, который проверяет возможность сохранения заказа

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = OrderJpaTestConfiguration.class)
public class OrderJpaTest {

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private TransactionTemplate transactionTemplate;

    @Test
    public void shouldSaveAndLoadOrder() {

        Long orderId = transactionTemplate.execute((ts) -> {
            Order order =
                new Order(CONSUMER_ID, AJANTA_ID, CHICKEN_VINDALOO_LINE_ITEMS);
            orderRepository.save(order);
            return order.getId();
        });

        transactionTemplate.execute((ts) -> {
            Order order = orderRepository.findById(orderId).get();

            assertEquals(OrderState.APPROVAL_PENDING, order.getState());
            assertEquals(AJANTA_ID, order.getRestaurantId());
            assertEquals(CONSUMER_ID, order.getConsumerId().longValue());
            assertEquals(CHICKEN_VINDALOO_LINE_ITEMS, order.getLineItems());
            return null;
        });
    }
}
```

Тестовый метод `shouldSaveAndLoadOrder()` выполняет две транзакции. Первая сохраняет только что созданный заказ в базе данных. Вторая загружает этот заказ и проверяет, правильно ли инициализированы его поля.

Но как обеспечить базу данных, которая используется в интеграционных тестах с сохранением? Эффективное решение для запуска экземпляра БД во время тестирования — применение Docker. В разделе 10.2 описывается, как автоматически запускать сервисы во время компонентного тестирования с помощью дополнения Docker Compose Gradle. В интеграционных тестах с помощью аналогичного подхода можно запускать, скажем, MySQL.

База данных — это всего лишь один из внешних компонентов, с которым взаимодействует сервис. Посмотрим, как писать интеграционные тесты для взаимодействия между сервисами приложения. Начнем с REST.

10.1.2. Интеграционное тестирование взаимодействия в стиле «запрос/ответ» на основе REST

REST широко используется в качестве механизма межсервисного взаимодействия. Клиент и сервис должны согласовать интерфейс REST API — это относится как к конечным точкам, так и к структуре тела запроса и ответа. Клиент должен послать HTTP-запрос подходящей конечной точке, а сервис — вернуть ответ, которого клиент от него ждет.

Например, в главе 8 описывалось, как API-шлюз приложения FTGO использует REST API для вызова многочисленных сервисов, включая `ConsumerService`, `OrderService` и `DeliveryService`. Конечная точка `GET /orders/{orderId}` сервиса `Order` является одной из тех, к которым обращается API-шлюз. Чтобы без сквозного тестирования убедиться в том, что API-шлюз и сервис `Order` могут взаимодействовать, нужно написать интеграционные тесты.

Как утверждалось в главе 9, хорошая стратегия интеграционного тестирования состоит в применении контрактов с расчетом на потребителя. Взаимодействие между API-шлюзом и конечной точкой `GET /orders/{orderId}` можно описать с помощью набора контрактов, основанных на HTTP. Каждый контракт состоит из HTTP-запроса и HTTP-ответа. Эти контракты используются для тестирования API-шлюза и сервиса `Order`.

На рис. 10.3 показано, как тестируются взаимодействия на основе REST с применением Spring Cloud Contract. Интеграционные тесты потребительской стороны для API-шлюза задействуют контракты для конфигурации фиктивного HTTP-сервера, симулирующего поведение сервиса `Order`. Запрос контракта описывает HTTP-запрос, выполняемый API-шлюзом, а его ответ определяет результат, который заглушка шлет обратно. Spring Cloud Contract использует контракты для генерации кода интеграционных тестов на стороне провайдера, тестирующих контроллеры сервиса `Order` с помощью Spring Mock MVC или Rest Assured Mock MVC. Запрос контракта описывает HTTP-запрос, направляемый контроллеру, а его ответ определяет результат, который контроллер должен вернуть.

Тест потребительской стороны `OrderServiceProxyTest` вызывает объект `OrderServiceProxy`, сконфигурированный для отправки HTTP-запросов к WireMock. WireMock — это инструмент для эффективной симуляции HTTP-серверов, в данном тесте он симулирует сервис `Order`. Spring Cloud Contract настраивает экземпляр WireMock так, чтобы тот отвечал на HTTP-запросы, определенные контрактами.

На стороне провайдера Spring Cloud Contract генерирует тестовый класс под названием `HttpTest`, который тестирует контроллеры сервиса `Order` с помощью Rest Assured Mock MVC. Тестовые классы наподобие `HttpTest` должны наследовать базовый класс, написанный вручную. В данном примере базовый класс `BaseHttp` создает экземпляр `OrderController` с внедренными макетами зависимостей и вызывает метод `RestAssuredMockMvc.standaloneSetup()`, чтобы сконфигурировать Spring MVC.

Давайте разберемся, как все это работает, на примере контракта.

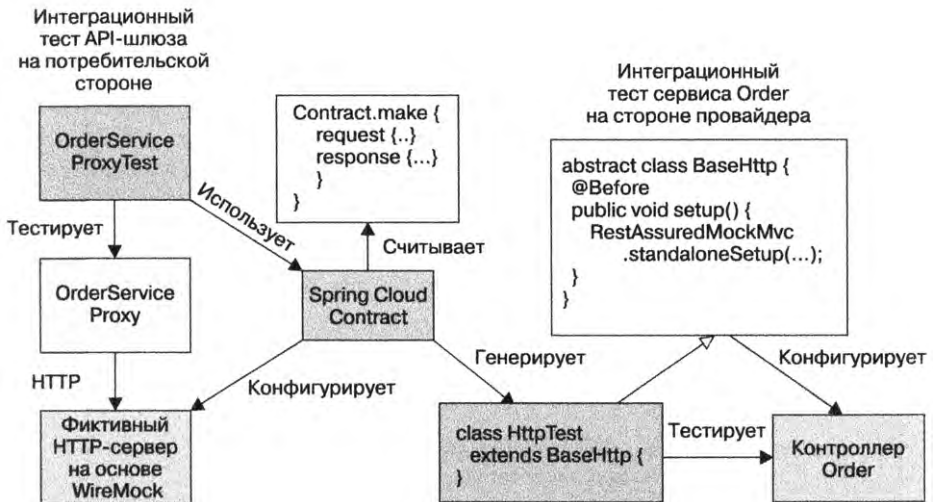


Рис. 10.3. Тест проверяет, соответствуют ли контракту классы адаптеров на обеих сторонах взаимодействия на основе REST между API-шлюзом и сервисом Order. Тесты на стороне потребителя проверяют корректность обращения OrderServiceProxy к сервису Order. Тесты на стороне провайдера проверяют корректность реализации конечных точек REST API контроллером OrderController

Пример контракта для REST API

REST-контракт, пример которого приведен в листинге 10.2, описывает HTTP-запрос, отправляемый REST-клиентом, и HTTP-ответ, который клиент ожидает получить от REST-сервера. Запрос контракта определяет HTTP-метод, путь и опциональные заголовки. В ответе контракта указываются HTTP-код состояния, опциональные заголовки и, если нужно, ожидаемое тело ответа.

Листинг 10.2. Контракт, описывающий взаимодействие в стиле «запрос/ответ» на основе HTTP

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'
        url '/orders/1223232'
    }
    response {
        status 200
        headers {
            header('Content-Type': 'application/json;charset=UTF-8')
        }
        body('{"orderId" : "1223232", "state" : "APPROVAL_PENDING"}')
    }
}
```

Этот конкретный контракт описывает успешную попытку извлечения API-шлюзом заказа из сервиса Order. Теперь посмотрим, как с его помощью написать интеграционные тесты для OrderService.

Интеграционные тесты контракта с расчетом на потребителя для сервиса Order

Интеграционные тесты контракта с расчетом на потребителя для сервиса Order проверяют, отвечает ли его API ожиданиям клиента. В листинге 10.3 показан базовый класс `HttpBase`, из которого Spring Cloud Contract генерирует тестовый класс. Он создает контроллеры с внедренными макетами зависимостей и делает так, чтобы возвращаемые ими значения заставляли контроллер сгенерировать ожидаемый ответ.

Листинг 10.3. Абстрактный базовый класс для тестов, которые генерирует фреймворк Spring Cloud Contract

```
public abstract class HttpBase {

    private StandaloneMockMvcBuilder controllers(Object... controllers) {
        ...
        return MockMvcBuilders.standaloneSetup(controllers)
            .setMessageConverters(...);
    }

    @Before
    public void setup() {
        OrderService orderService = mock(OrderService.class);
        OrderRepository orderRepository = mock(OrderRepository.class);
        OrderController orderController =
            new OrderController(orderService, orderRepository);

        when(orderRepository.findById(1223232L))
            .thenReturn(Optional.of(OrderDetailsMother.CHICKEN_VINDALOO_ORDER));
        ...
        RestAssuredMockMvc.standaloneSetup(controllers(orderController));
    }
}
```

Создаем OrderRepository с внедренными макетами

Конфигурируем OrderController с помощью Spring MVC

Делаем так, чтобы при вызове findById() объект OrderResponse возвращал заказ с полем orderId, указанным в контракте

Аргумент 1223232L, который передается методу `findById()` макета `OrderRepository`, совпадает со значением `orderId`, указанным в контракте из листинга 10.3. Этот тест проверяет наличие у сервиса Order конечной точки GET `/orders/{orderId}`, которая отвечает ожиданиям клиента.

Рассмотрим соответствующий клиентский тест.

Интеграционный тест на стороне потребителя для класса API-шлюза OrderServiceProxy

Класс API-шлюза `OrderServiceProxy` обращается к конечной точке GET `/orders/{orderId}`. В листинге 10.4 показан тестовый класс `OrderServiceProxyIntegrationTest`, который проверяет, соответствует ли этот прокси контракту. Этот класс помечен аннотацией `@AutoConfigureStubRunner` из состава Spring Cloud Contract. Благодаря этому фреймворк Spring Cloud Contract запускает сервер WireMock на случайном порте и конфигурирует его с помощью заданных контрактов. `OrderSer-`

`viceProxyIntegrationTest` делает так, чтобы прокси `OrderServiceProxy` отправлял свои запросы на порт `WireMock`.

Листинг 10.4. Интеграционный тест на стороне потребителя для класса API-шлюза `OrderServiceProxy`

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes=TestConfiguration.class,
    webEnvironment= SpringBootTest.WebEnvironment.NONE)
@AutoConfigureStubRunner(ids =
    {"net.chrisrichardson.ftgo.contracts:ftgo-order-service-contracts"},
    workOffline = false)
@DirtiesContext
public class OrderServiceProxyIntegrationTest {

    @Value("${stubrunner.runningstubs.ftgo-order-service-contracts.port}")
    private int port;
    private OrderDestinations orderDestinations;
    private OrderServiceProxy orderService;

    @Before
    public void setUp() throws Exception {
        orderDestinations = new OrderDestinations();
        String orderServiceUrl = "http://localhost:" + port;
        orderDestinations.setOrderServiceUrl(orderServiceUrl);
        orderService = new OrderServiceProxy(orderDestinations,
            WebClient.create());
    }

    @Test
    public void shouldVerifyExistingCustomer() {
        OrderInfo result = orderService.findOrderById("1223232").block();
        assertEquals("1223232", result.getOrderID());
        assertEquals("APPROVAL_PENDING", result.getState());
    }

    @Test(expected = OrderNotFoundException.class)
    public void shouldFailToFindMissingOrder() {
        orderService.findOrderById("555").block();
    }
}
```

Делаем так, чтобы Spring Cloud Contract сконфигурировал WireMock с помощью контрактов сервиса Order

Получаем случайно назначенный порт, на котором работает WireMock

Создаем прокси OrderServiceProxy, настроенный для выполнения запросов к WireMock

Каждый тестовый метод обращается к прокси `OrderServiceProxy` и проверяет, возвращает ли тот корректные значения или генерирует ожидаемое исключение. Тестовый метод `shouldVerifyExistingCustomer()` проверяет, совпадает ли возвращаемое значение вызова `findOrderById()` с тем, которое указано в ответе контракта. Метод `shouldFailToFindMissingOrder()` пытается извлечь несуществующий заказ и проверяет, генерирует ли `OrderServiceProxy` исключение `OrderNotFoundException`. Тестирование REST-клиента и REST-сервера с помощью одного контракта обеспечивает согласованность их API.

Теперь поговорим о том, как выполнить аналогичное тестирование сервисов, взаимодействующих путем обмена сообщениями.

10.1.3. Интеграционное тестирование взаимодействия в стиле «издатель/подписчик»

Сервисы часто публикуют доменные события, потребляемые другими сервисами (одним или несколькими). При интеграционном тестировании нужно убедиться в том, что издатель и его подписчики согласовали канал сообщений и структуру доменных событий. Например, сервис `Order` публикует события типа `Order*` при каждом создании или обновлении агрегата `Order`. Один из потребителей этих событий — сервис `Order History`. Таким образом, мы должны написать тесты, которые проверяют возможность взаимодействия между этими сервисами.

То, как производится интеграционное тестирование взаимодействия в стиле «издатель/подписчик», показано на рис. 10.4. Процесс похож на тестирование общения по REST. Как и прежде, взаимодействие описывается в виде набора контрактов. Отличие лишь в том, что в каждом контракте задается доменное событие.

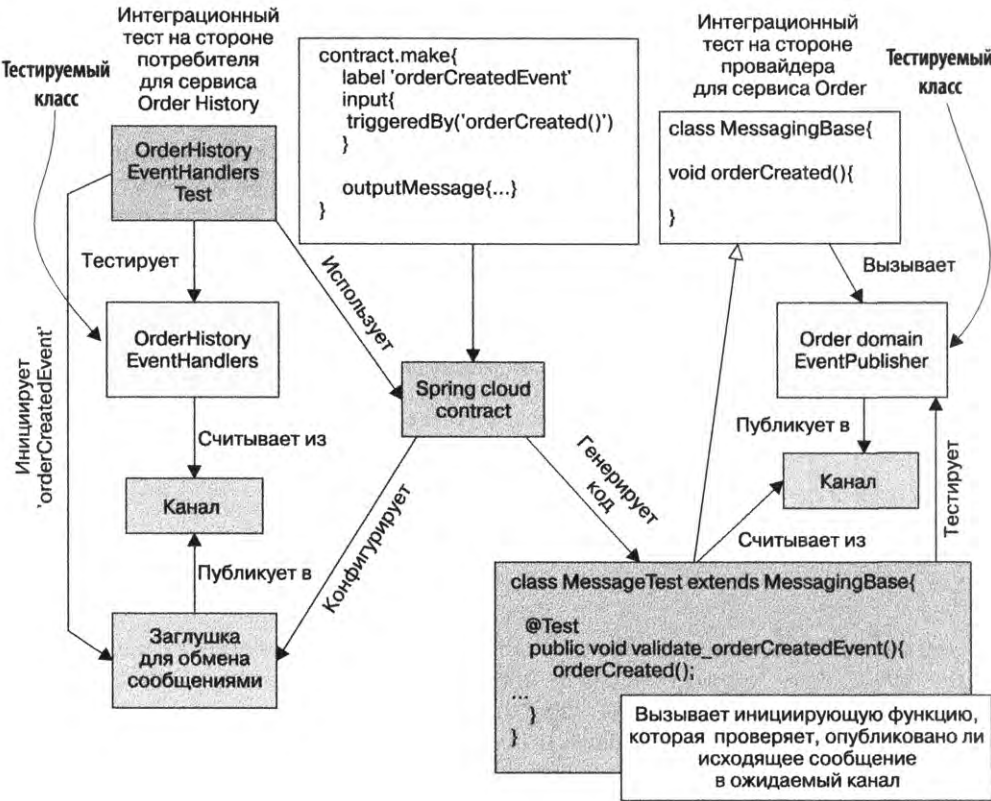


Рис. 10.4. Контракты используются для тестирования обеих сторон взаимодействия в стиле «издатель/подписчик». Тесты на стороне провайдера проверяют, соответствуют ли контракту события, публикуемые издателем `OrderDomainEventPublisher`. Тесты на стороне потребителя проверяют, потребляет ли `OrderHistoryEventHandlers` демонстрационные события из контракта

Каждый тест на стороне потребителя публикует событие, заданное контрактом, и проверяет, корректно ли `OrderHistoryEventHandlers` вызывает свои фиктивные зависимости.

На стороне провайдера Spring Cloud Contract генерирует тестовые классы, наследованные от абстрактного родительского класса `MessagingBase`, написанного вручную. Каждый тестовый метод вызывает метод-перехватчик, определенный в `MessagingBase`, который должен инициировать публикацию события со стороны сервиса. В этом примере каждый метод-перехватчик обращается к издателю `OrderDomainEventPublisher`, ответственному за публикацию событий агрегата `Order`. Затем тестовый метод проверяет, было ли опубликовано ожидаемое событие. Давайте подробно рассмотрим, как работают эти тесты. Начнем с контракта.

Контракт для публикации события `OrderCreated`

В листинге 10.5 показан контракт для события `OrderCreated`. Он определяет канал события, а также ожидаемое тело и заголовки сообщения.

Листинг 10.5. Контракт для взаимодействия в стиле «издатель/подписчик»

```
package contracts;

org.springframework.cloud.contract.spec.Contract.make {
    label 'orderCreatedEvent'
    input {
        triggeredBy('orderCreated()')
    }
    outputMessage {
        sentTo('net.chrisrichardson.ftgo.orderservice.domain.Order')
        body('{"orderDetails":{"lineItems":[{"quantity":5,"menuItemId":"1",
            "name":"Chicken Vindaloo","price":"12.34","total":"61.70"}]},
            "orderTotal":"61.70","restaurantId":1,
            "consumerId":"1511300065921","orderState":"APPROVAL_PENDING"}')
        headers {
            header('event-aggregate-type',
                'net.chrisrichardson.ftgo.orderservice.domain.Order')
            header('event-aggregate-id', '1')
        }
    }
}
```

Используется потребителем, чтобы инициировать публикацию события

Вызывается сгенерированным тестом провайдера

Доменное событие `OrderCreated`

Контракт имеет еще два важных элемента:

- ❑ `label` — используется потребителем, чтобы инициировать публикацию события фреймворком Spring Contract;
- ❑ `triggeredBy` — имя метода родительского класса, который вызывается сгенерированным тестовым методом, чтобы инициировать публикацию события.

Посмотрим, как применять этот контракт. Начнем с теста на стороне провайдера для сервиса `Order`.

Тесты контрактов с расчетом на потребителя для сервиса Order

Тест на стороне провайдера для сервиса Order — это еще один интеграционный тест контракта с расчетом на потребителя. Он позволяет убедиться в том, что издатель `OrderDomainEventPublisher`, который отвечает за публикацию доменных событий агрегата Order, публикует события, соответствующие ожиданиям его клиента. В листинге 10.6 показан базовый класс `MessagingBase`, на котором основаны тестовые классы, генерируемые фреймворком Spring Cloud Contract. Он конфигурирует класс `OrderDomainEventPublisher` так, чтобы тот использовал фиктивную инфраструктуру обмена сообщениями, находящуюся в памяти. Он также определяет методы, такие как `orderCreated()`, с помощью которых сгенерированные тесты иницируют публикацию события.

Листинг 10.6. Абстрактный базовый класс для тестов на стороне провайдера, генерируемых фреймворком Spring Cloud Contract

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = MessagingBase.TestConfiguration.class,
                webEnvironment = SpringBootTest.WebEnvironment.NONE)
@AutoConfigureMessageVerifier
public abstract class MessagingBase {
    @Configuration
    @EnableAutoConfiguration
    @Import({EventuateContractVerifierConfiguration.class,
            TramEventsPublisherConfiguration.class,
            TramInMemoryConfiguration.class})
    public static class TestConfiguration {

        @Bean
        public OrderDomainEventPublisher
            OrderDomainEventPublisher(DomainEventPublisher eventPublisher) {
            return new OrderDomainEventPublisher(eventPublisher);
        }
    }

    @Autowired
    private OrderDomainEventPublisher OrderDomainEventPublisher;

    protected void orderCreated() {
        OrderDomainEventPublisher.publish(CHICKEN_VINDALOO_ORDER,
            singletonList(new OrderCreatedEvent(CHICKEN_VINDALOO_ORDER_DETAILS)
        ));
    }
}
```

`orderCreated()` вызывается
сгенерированным тестовым
подклассом для публикации события

Этот тестовый класс настраивает `OrderDomainEventPublisher` для использования фиктивной инфраструктуры обмена сообщениями. `orderCreated()` вызывается

тестовым методом, сгенерированным из контракта, показанного в листинге 10.5. Он вызывает `OrderDomainEventPublisher`, чтобы опубликовать событие `OrderCreated`. Тестовый метод пытается получить это событие и затем проверяет, совпадает ли оно с тем, что было указано в контракте. Теперь рассмотрим соответствующие тесты на стороне потребителя.

Тест контракта на стороне потребителя для сервиса Order History

Сервис `Order History` потребляет события, публикуемые сервисом `Order`. Как описывалось в главе 7, для обработки этих событий используется адаптер класса `OrderHistoryEventHandlers`. Его обработчики обращаются к объекту `OrderHistoryDao`, чтобы обновить CQRS-представление. В листинге 10.7 показан интеграционный тест на стороне потребителя. Он создает экземпляр `OrderHistoryEventHandlers` с внедренным макетом `OrderHistoryDao`. Каждый тестовый метод сначала применяет `Spring Cloud`, чтобы опубликовать событие, указанное в контракте, а затем проверяет корректность обращения `OrderHistoryEventHandlers` к `OrderHistoryDao`.

Листинг 10.7. Интеграционный тест на стороне потребителя для класса `OrderHistoryEventHandlers`

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes= OrderHistoryEventHandlersTest.TestConfiguration.class,
    webEnvironment= SpringBootTest.WebEnvironment.NONE)
@AutoConfigureStubRunner(ids =
    {"net.chrisrichardson.ftgo.contracts:ftgo-order-service-contracts"},
    workOffline = false)
@DirtiesContext
public class OrderHistoryEventHandlersTest {

    @Configuration
    @EnableAutoConfiguration
    @Import({OrderHistoryServiceMessagingConfiguration.class,
        TramCommandProducerConfiguration.class,
        TramInMemoryConfiguration.class,
        EventuateContractVerifierConfiguration.class})
    public static class TestConfiguration {

        @Bean
        public OrderHistoryDao orderHistoryDao() {
            return mock(OrderHistoryDao.class);
        }
    }

    @Test
    public void shouldHandleOrderCreatedEvent() throws ... {
        stubFinder.trigger("orderCreatedEvent");
    }
}
```

Создаем макет `OrderHistoryDao`, чтобы внедрить его в `OrderHistoryEventHandlers`

Иницирует фиктивный метод `orderCreatedEvent`, который генерирует событие `OrderCreated`


```
eventually(() -> {
    verify(orderHistoryDao).addOrder(any(Order.class), any(Optional.class));
});
```

←
Проверяем, вызывает ли OrderHistoryEventHandlers
метод orderHistoryDao.addOrder()

Тестовый метод `shouldHandleOrderCreatedEvent()` заставляет Spring Cloud Contract опубликовать событие `OrderCreated`. Затем он проверяет, вызвал ли класс `OrderHistoryEventHandlers` метод `orderHistoryDao.addOrder()`. Тестирование издателя и потребителя доменных событий с помощью одного контракта обеспечивает согласованность их API. Теперь посмотрим, как выполнить интеграционное тестирование сервисов, которые взаимодействуют с использованием асинхронных запросов/ответов.

10.1.4. Интеграционные тесты контрактов для взаимодействия на основе асинхронных запросов/ответов

«Подписчик/издатель» — это не единственный стиль взаимодействия на основе сообщений. Сервисы могут общаться также с помощью асинхронных запросов/ответов. Например, в главе 4 мы видели, как сервис `Order` реализует повествования, которые рассылают командные сообщения различным сервисам, таким как `Kitchen`, и обрабатывает полученные ответы.

Во взаимодействии подобного рода одна из сторон является запрашивающей (сервис, отправляющий команды), а другая — отвечающей (сервис, который обрабатывает команду и возвращает ответ). Они должны согласовать название канала для командных сообщений и структуру запросов/ответов. Попробуем написать интеграционные тесты для такого вида взаимодействия.

На рис. 10.5 показано, как протестировать взаимодействие между сервисами `Order` и `Kitchen`. Подход к интеграционному тестированию взаимодействия на основе асинхронных запросов/ответов довольно близок к использованному при тестировании общения по REST. Коммуникация между сервисами определяется набором контрактов. Отличие состоит в том, что вместо HTTP-запросов и ответов контракт описывает входящее и исходящее сообщения.

Тест на стороне потребителя позволяет убедиться в том, что прокси-класс корректно структурирует командные сообщения и корректно обрабатывает ответы. В этом примере класс `KitchenServiceProxyTest` тестирует `KitchenServiceProxy`. С помощью Spring Cloud Contract он конфигурирует заглушки, которые проверяют, совпадает ли команда с входящим сообщением контракта, а после этого выдает соответствующее исходящее сообщение.

Тесты на стороне провайдера генерируются фреймворком Spring Cloud Contract. Каждый тестовый метод относится к какому-то контракту. Он отправляет входящее

сообщение контракта в качестве команды и проверяет, совпадает ли исходящее сообщение того же контракта с полученным ответом. Давайте подробнее рассмотрим этот процесс, начиная с контракта.

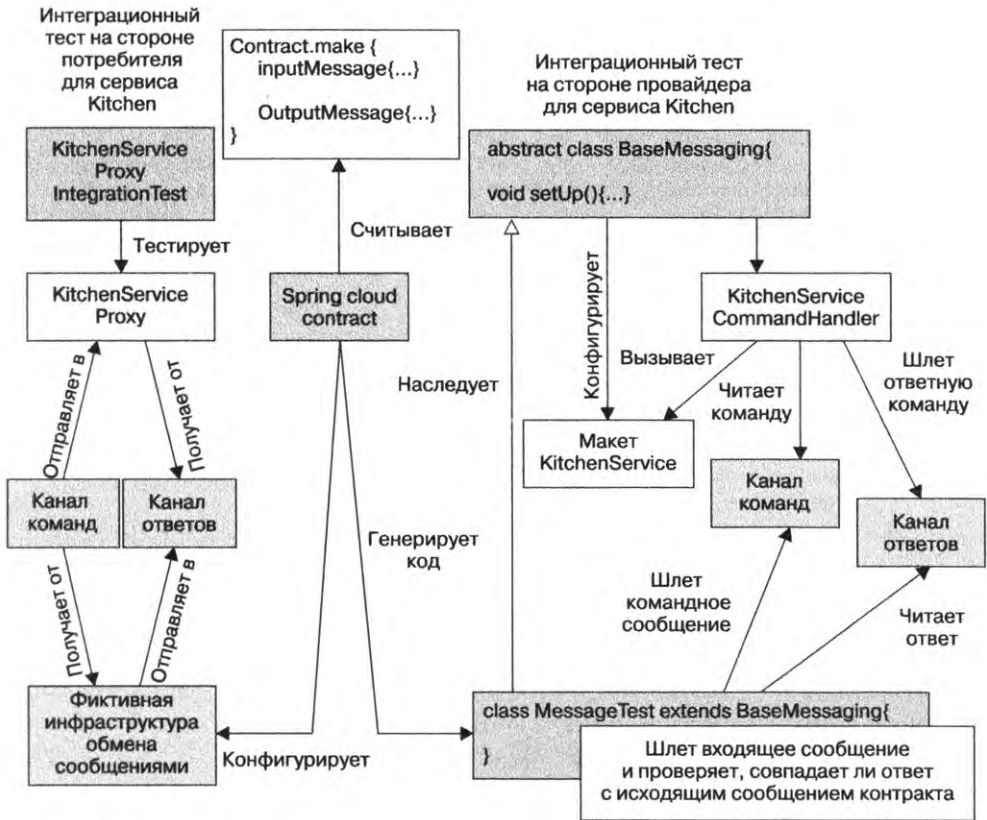


Рис. 10.5. Контракты применяются для тестирования классов адаптеров, реализующих обе стороны асинхронного взаимодействия вида «запрос/ответ». Тесты на стороне провайдера проверяют, обрабатывает ли класс `KitchenServiceCommandHandler` команды и возвращает ли ответы. Тесты на стороне потребителя проверяют, отправляет ли класс `KitchenServiceProxy` команды, которые соответствуют контракту, и обрабатывает ли демонстрационные ответы, возвращаемые контрактом

Пример контракта для асинхронных запросов/ответов

В листинге 10.8 показан контракт для одного взаимодействия. Он состоит из входящего и исходящего сообщений. У обоих сообщений есть канал, тело и заголовок. Соглашение об именовании показано с точки зрения провайдера. Элемент `messageFrom` входящего сообщения указывает канал, из которого происходит чтение. Точно так же элемент `sentTo` исходящего сообщения определяет канал, в который будут отправляться ответы.

Листинг 10.8. Контракт, описывающий, как сервис Order асинхронно обращается к сервису Kitchen
 package contracts;

```
org.springframework.cloud.contract.spec.Contract.make {
    label 'createTicket'
    input {
        messageFrom('kitchenService')
        messageBody('{"orderId":1,"restaurantId":1,"ticketDetails":{...}}')
        messageHeaders {
            header('command_type', 'net.chrisrichardson...CreateTicket')
            header('command_saga_type', 'net.chrisrichardson...CreateOrderSaga')
            header('command_saga_id', $(consumer(regex('[0-9a-f]{16}-[0-9a-f]{16}'))))
            header('command_reply_to', 'net.chrisrichardson...CreateOrderSaga-Reply')
        }
    }
    outputMessage {
        sentTo('net.chrisrichardson...CreateOrderSaga-reply')
        body([
            ticketId: 1
        ])
        headers {
            header('reply_type', 'net.chrisrichardson...CreateTicketReply')
            header('reply_outcome-type', 'SUCCESS')
        }
    }
}
```

Командное сообщение,
отправленное сервисом Order
в канал kitchenService

Ответное сообщение,
отправленное сервисом Kitchen

В этом примере контракта роль входящего сообщения играет команда `CreateTicket`, которая отправляется в канал `kitchenService`. Исходящее сообщение представлено успешным ответом, который отправляется в канал ответов повествования `CreateOrderSaga`. Посмотрим, как эти контракты используются в тестах. Начнем с тестов на стороне потребителя для сервиса `Order`.

Интеграционные тесты контрактов на стороне потребителя для взаимодействия в виде асинхронных запросов/ответов

Написание интеграционных тестов на стороне потребителя для асинхронного взаимодействия в стиле «запрос/ответ» похоже на тестирование REST-клиента. Тест обращается к прокси-классу сервиса, отвечающему за обмен сообщениями, и проверяет два аспекта его поведения: соответствует ли контракту командное сообщение, отправленное прокси-классом, и корректно ли последний обрабатывает ответ.

В листинге 10.9 показан интеграционный тест на стороне потребителя для прокси-класса `KitchenServiceProxy`, с помощью которого сервис `Order` обращается к сервису `Kitchen`. Каждый тест шлет классу `KitchenServiceProxy` командное сообщение и проверяет, возвращает ли тот ожидаемый результат. Он применяет Spring Cloud Contract, чтобы сконфигурировать заглушки для сервиса `Kitchen`, которые ищут контракт, чье входящее сообщение совпадает с командой, и затем отправляет свое исходящее сообщение в качестве ответа. Для упрощения и ускорения процесса тесты используют фиктивную инфраструктуру для обмена сообщениями, загруженную в оперативную память.

Листинг 10.9. Интеграционный тест на стороне потребителя для сервиса Order

```

@RunWith(SpringRunner.class)
@SpringBootTest(classes=
    KitchenServiceProxyIntegrationTest.TestConfiguration.class,
    webEnvironment= SpringBootTest.WebEnvironment.NONE)
@AutoConfigureStubRunner(ids =
    {"net.chrisrichardson.ftgo.contracts:ftgo-kitchen-service-contracts"},
    workOffline = false)
@DirtiesContext
public class KitchenServiceProxyIntegrationTest {

    @Configuration
    @EnableAutoConfiguration
    @Import({TramCommandProducerConfiguration.class,
        TramInMemoryConfiguration.class,
        EventuateContractVerifierConfiguration.class})
    public static class TestConfiguration { ... }

    @Autowired
    private SagaMessagingTestHelper sagaMessagingTestHelper;

    @Autowired
    private KitchenServiceProxy kitchenServiceProxy;

    @Test
    public void shouldSuccessfullyCreateTicket() {
        CreateTicket command = new CreateTicket(AJANTA_ID,
            OrderDetailsMother.ORDER_ID,
            new TicketDetails(Collections.singletonList(
                new TicketLineItem(CHICKEN_VINDALOO_MENU_ITEM_ID,
                    CHICKEN_VINDALOO,
                    CHICKEN_VINDALOO_QUANTITY))));

        String sagaType = CreateOrderSaga.class.getName();

        CreateTicketReply reply =
            sagaMessagingTestHelper
                .sendAndReceiveCommand(kitchenServiceProxy.create,
                    command,
                    CreateTicketReply.class, sagaType);

        assertEquals(new CreateTicketReply(OrderDetailsMother.ORDER_ID), reply);
    }
}

```

← Настраиваем фиктивный сервис Kitchen, который будет отвечать на сообщения

← Отправляем команду и ждем ответа

← Проверяем ответ

Тестовый метод `shouldSuccessfullyCreateTicket()` отправляет командное сообщение `CreateTicket` и проверяет, содержит ли ответ ожидаемые данные. Он использует вспомогательный класс `SagaMessagingTestHelper`, чтобы отправлять и получать сообщения синхронно.

Теперь посмотрим, как пишутся интеграционные тесты на стороне провайдера.

Написание тестов на стороне провайдера с расчетом на потребителя для взаимодействия в виде асинхронных запросов/ответов

Интеграционный тест на стороне провайдера должен убедиться в том, что в результате обработки командного сообщения провайдер возвращает корректный ответ. Spring Cloud Contract генерирует тестовые классы с тестовым методом для каждого контракта. Каждый тестовый метод шлет входящее сообщение контракта и проверяет, совпадает ли исходящее сообщение контракта с ответом.

Интеграционные тесты на стороне провайдера для сервиса `Kitchen` тестируют класс `KitchenServiceCommandHandler`, обрабатывающий сообщения путем вызова `KitchenService`. В листинге 10.10 показан класс `AbstractKitchenServiceConsumerContractTest`, который наследуют тесты, сгенерированные фреймворком Spring Cloud Contract. Он создает экземпляр `KitchenServiceCommandHandler` с внедренным макетом `KitchenService`.

Листинг 10.10. Родительский класс тестов на стороне провайдера, ориентированных на потребителя и предназначенных для сервиса `Kitchen`

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes =
    AbstractKitchenServiceConsumerContractTest.TestConfiguration.class,
    webEnvironment = SpringBootTest.WebEnvironment.NONE)
@AutoConfigureMessageVerifier
public abstract class AbstractKitchenServiceConsumerContractTest {

    @Configuration
    @Import(RestaurantMessageHandlersConfiguration.class)
    public static class TestConfiguration {
        ...
        @Bean
        public KitchenService kitchenService() {
            return mock(KitchenService.class);
        }
    }

    @Autowired
    private KitchenService kitchenService;

    @Before
    public void setup() {
        reset(kitchenService);
        when(kitchenService
            .createTicket(eq(1L), eq(1L),
                any(TicketDetails.class)))
            .thenReturn(new Ticket(1L, 1L,
                new TicketDetails(Collections.emptyList())));
    }
}
```

← Заменяет определение `kitchenService` @Bean-макетом

← Настраивает макет так, чтобы тот возвращал значения, совпадающие с исходящим сообщением контракта

`KitchenServiceCommandHandler` вызывает `KitchenService` с аргументами, сформированными на основе входящего сообщения контракта, и создает ответ, полученный из возвращаемого значения. Метод тестового класса `setup()` конфигурирует фик-

тивный сервис `Kitchen`, чтобы тот возвращал значения, совпадающие с исходящим сообщением контракта.

Интеграционные и модульные тесты проверяют поведение отдельных частей сервиса. Интеграционные тесты позволяют убедиться в том, что сервисы могут взаимодействовать со своими клиентами и зависимостями. Модульные тесты подтверждают корректность логики сервиса. Но ни те ни другие не охватывают весь сервис целиком. Чтобы проверить работу всего сервиса, мы продвинемся вверх по пирамиде и посмотрим, как создаются компонентные тесты.

10.2. Разработка компонентных тестов

До сих пор мы обсуждали тестирование отдельных классов и их наборов. Но представьте, что нужно убедиться в надлежащей работе сервиса `Order`. Иными словами, мы хотим написать приемочные тесты, которые работают с сервисом как с единым целым и проверяют его поведение через его API. Для этого можно написать практически сквозные тесты и развернуть сервис `Order` вместе со всеми его транзитивными зависимостями. Но, как вы уже должны понимать, это медленный, ненадежный и затратный подход.

Шаблон «Компонентный тест сервиса»

Тестирует отдельно взятый сервис. См. microservices.io/patterns/testing/service-component-test.html.

Компонентное тестирование — гораздо лучший способ написания приемочных тестов для сервисов. Как видно на рис. 10.6, *компонентные тесты* находятся между интеграционными и сквозными. Их задача — проверка поведения отдельно взятого сервиса. Они подменяют заглушками зависимости сервиса и симулируют их работу. Они могут даже использовать фиктивные версии таких компонентов, как базы данных, размещая их в оперативной памяти.

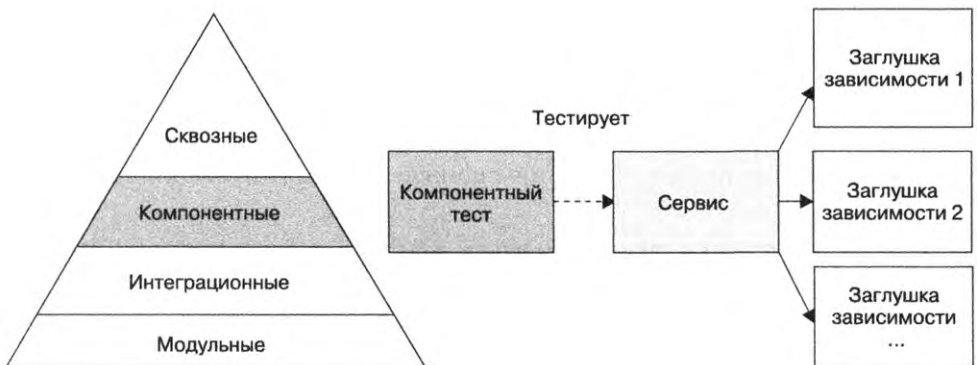


Рис. 10.6. Компонентный тест тестирует отдельно взятый сервис. Обычно он применяет заглушки для зависимостей сервиса

Вначале я кратко продемонстрирую, как с помощью языка DSL с названием Gherkin можно писать приемочные тесты для таких сервисов, как `Order`. После этого мы обсудим многочисленные архитектурные трудности, присущие компонентному тестированию. Затем я покажу, как написать приемочные тесты для сервиса `Order`.

Посмотрим, как выглядят приемочные тесты на языке Gherkin.

10.2.1. Определение приемочных тестов

Приемочные тесты относятся к бизнес-аспектам программного компонента. Они описывают предпочтительное поведение, которое наблюдают его клиенты, игнорируя внутреннюю реализацию. Эти тесты формируются на основе пользовательских историй или сценариев. Например, `Place Order` — это одна из ключевых историй сервиса `Order`:

Как у потребителя сервиса `Order`,
У меня должна быть возможность разместить заказ

Мы можем расширить ее до такого сценария:

Дано: действительный потребитель
Дано: действительная банковская карта
Дано: ресторан принимает заказы
Когда я заказываю виндалу из курицы в `Ajanta`
Тогда заказ должен быть принят
И должно быть опубликовано событие `OrderAuthorized`

Этот сценарий описывает желаемое поведение сервиса `Order` с точки зрения его API.

Каждый сценарий определяет приемочный тест. Разделы **Дано** соответствуют подготовительному этапу теста, раздел **Когда** соотносится с этапом выполнения, а **Тогда** и **И** — с проверкой. Позже вам будет представлен тест для этого сценария, который делает следующее.

1. Создает заказ, обращаясь к конечной точке `POST /orders`.
2. Проверяет состояние заказа, обращаясь к конечной точке `GET /orders/{orderId}`.
3. Подписывается на подходящий канал сообщений, чтобы проверить, опубликовал ли сервис `Order` событие `OrderAuthorized`.

Мы могли бы перевести каждый сценарий в код на Java. Но более простым решением будет создание приемочных тестов с помощью языка DSL, такого как Gherkin.

10.2.2. Написание приемочных тестов с помощью Gherkin

Написание приемочных тестов на Java сопровождается определенными трудностями. Существует риск расхождения тестов и их сценариев. Также нет прямой связи между высокоуровневыми сценариями и кодом, состоящим из низкоуровневых подробностей реализации. К тому же некоторым тестам не хватает ясности, по-

этому их просто нельзя перевести в код на языке Java. Лучше сразу писать такие сценарии, которые можно выполнять, — это позволит избавиться от ручного преобразования.

Gherkin — это язык DSL для написания исполняемых спецификаций. Приемочные тесты на нем выглядят как сценарии на английском языке, подобные показанному ранее. Спецификация выполняется с помощью Cucumber — фреймворка автоматизации тестирования для Gherkin. Два инструмента, Gherkin и Cucumber, позволяют избежать ручного преобразования сценариев в запускаемый код.

Спецификация сервиса, такого как `Order`, состоит из возможностей. Каждая *возможность* описывается в виде набора сценариев, похожих на тот, что вы видели ранее. Сценарий имеет структуру «дано — когда — тогда». «Дано» — это предварительные условия, «когда» — это действие или происходящее событие, а «тогда/и» — ожидаемый результат.

Например, желаемое поведение сервиса `Order` определяется несколькими возможностями, такими как `Place Order`, `Cancel Order` и `Revise Order`. В листинге 10.11 представлен фрагмент возможности `Place Order`, которая состоит из нескольких элементов:

- ❑ *названия* — в данном случае это `Place Order`;
- ❑ *краткого описания спецификации* — описывает назначение возможности. В данном случае указана пользовательская история;
- ❑ *сценариев* `Order authorized` (заказ авторизован) и `Order rejected due to expired credit card` (заказ отклонен из-за просроченной банковской карты).

Листинг 10.11. Определение возможности `Place Order` и некоторые ее сценарии на языке Gherkin

Feature: Place Order

```
As a consumer of the Order Service
I should be able to place an order
```

```
Scenario: Order authorized
```

```
  Given a valid consumer
```

```
  Given using a valid credit card
```

```
  Given the restaurant is accepting orders
```

```
  When I place an order for Chicken Vindaloo at Ajanta
```

```
  Then the order should be APPROVED
```

```
  And an OrderAuthorized event should be published
```

```
Scenario: Order rejected due to expired credit card
```

```
  Given a valid consumer
```

```
  Given using an expired credit card
```

```
  Given the restaurant is accepting orders
```

```
  When I place an order for Chicken Vindaloo at Ajanta
```

```
  Then the order should be REJECTED
```

```
  And an OrderRejected event should be published
```

...

В обоих сценариях клиент пытается разместить заказ. В первом случае это удастся. Во втором заказ отклоняется, потому что у клиента просрочена банковская карта. Больше информации о языке Gherkin можно найти в книге Камиля Ничеи (Kamil Nicieja) *Writing Great Specifications: Using Specification by Example and Gherkin* (Manning, 2017).

Выполнение спецификации формата Gherkin с помощью Cucumber

Cucumber — это фреймворк для автоматического тестирования, выполняющий тесты, написанные на Gherkin. Он доступен для разных языков, включая Java. При использовании его в Java вы должны написать класс пошагового определения (листинг 10.12). Класс *пошагового определения* состоит из методов, которые определяют значение каждого из шагов в цепочке «дано — тогда — когда». Каждый метод имеет одну из следующих аннотаций: `@Given`, `@When`, `@Then` или `@And`. У всех аннотаций есть элемент `value`, он содержит регулярное выражение, которое Cucumber сопоставляет с шагами.

Листинг 10.12. Класс пошагового определения на Java делает сценарии Gherkin исполняемыми

```
public class StepDefinitions ... {

    ...

    @Given("A valid consumer")
    public void useConsumer() { ... }

    @Given("using a(?:) (.*) credit card")
    public void useCreditCard(String ignore, String creditCard) { ... }

    @When("I place an order for Chicken Vindaloo at Ajanta")
    public void placeOrder() { ... }

    @Then("the order should be (.*)")
    public void theOrderShouldBe(String desiredOrderState) { ... }

    @And("an (.*) event should be published")
    public void verifyEventPublished(String expectedEventClass) { ... }

}
```

Каждый из типов методов является частью определенного этапа теста:

- ❑ `@Given` — подготовительный этап;
- ❑ `@When` — этап выполнения;
- ❑ `@Then` и `@And` — этап проверки.

В подразделе 10.2.4, когда я опишу этот класс подробнее, вы увидите, что многие из упомянутых методов шлюют REST-вызовы сервису `Order`. Например, метод

`placeOrder()` создает заказ, обращаясь к конечной точке `POST /orders`. Метод `theOrderShouldBe()` проверяет состояние заказа, делая вызов `GET /orders/ {orderId}`.

Прежде чем приступать к написанию пошаговых классов, исследуем некоторые архитектурные проблемы, присущие компонентным тестам.

10.2.3. Проектирование компонентных тестов

Представьте, что вы пишете компонентный тест для сервиса `Order`. В предыдущем подразделе было показано, как определить желаемое поведение на языке Gherkin и выполнить его с помощью Cucumber. Но перед выполнением сценария компонентный тест должен запустить сервис `Order` и подготовить его зависимости. Мы выполняем изолированное тестирование, поэтому компонентному тесту нужно сконфигурировать заглушки для нескольких сервисов, включая `Kitchen`. Ему также необходимо предоставить базу данных и инфраструктуру обмена сообщениями. Существует несколько решений, которые позволяют выбирать между реализмом и скоростью/простотой.

Компонентные тесты внутри процесса

Одно из решений состоит в написании *внутрипроцессных компонентных тестов*, которые подменяют зависимости сервиса заглушками и макетами, загружаемыми в оперативную память. Например, вы можете задействовать фреймворк Spring Boot для написания как самого сервиса, так и его компонентных тестов. Тестовый класс, помеченный аннотацией `@SpringBootTest`, запускает сервис на той же JVM-машине, что и сам тест. Он внедряет зависимости, чтобы заставить сервис обращаться к макетам и заглушкам. Например, тест может сконфигурировать сервис `Order` так, чтобы тот использовал резидентную базу данных типа JDBC, такую как H2, HSQLDB или Derby, и резидентные заглушки для Eventuate Tram. Внутрипроцессные тесты более быстрые и простые в написании. Их слабая сторона связана с тем, что они не тестируют развертываемый сервис.

Компонентное тестирование за пределами процесса

Более реалистичным решением будет упаковать сервис в формат, готовый к промышленному применению, и запустить его в виде отдельного процесса. Например, в главе 12 пойдет речь о том, что упаковка сервисов в качестве образов контейнеров для Docker становится все более популярной. *Внепроцессный компонентный тест* задействует настоящую инфраструктуру, включая базу данных и брокер сообщений, но при этом подменяет заглушками все зависимости, которые являются сервисами приложения. Например, внепроцессный компонентный тест для сервиса `Order` будет использовать MySQL и Apache Kafka, заменяя заглушками сервисы `Comsumer` и `Accounting`. Заглушки будут потреблять сообщения, полученные от Apache Kafka, и отправлять обратно свои ответы.

Ключевое преимущество внепроцессного компонентного тестирования — более широкое покрытие тестов, поскольку тестируемые компоненты значительно меньше отличаются от кода, который будет развертываться. Недостаток этих тестов по сравнению с внутривпроцессными в том, что они сложнее в написании, выполняются медленнее и могут оказаться не такими надежными. К тому же вам нужно разобраться с тем, как подменять сервисы приложения. Посмотрим, как это делается.

Как подменить сервисы во внепроцессных тестах

Тестируемый сервис часто взаимодействует со своими зависимостями в стиле, который подразумевает возвращение ответа. Сервис `Order`, к примеру, использует асинхронные запросы/ответы и рассылает командные сообщения различным сервисам. API-шлюз применяет протокол HTTP, который работает по принципу «запрос/ответ». Внепроцессный тест должен сконфигурировать заглушки для этих зависимостей так, чтобы они обрабатывали запросы и возвращали ответы.

Одно из решений — применение фреймворка `Spring Cloud Contract`, с которым мы познакомились в разделе 10.1 при обсуждении интеграционного тестирования. Мы могли бы написать контракты, которые конфигурируют заглушки для компонентных тестов. При этом необходимо понимать, что эти контракты, в отличие от интеграционных, будут использоваться лишь компонентными тестами.

Еще один недостаток фреймворка `Spring Cloud Contract` в контексте компонентного тестирования состоит в том, что он сосредоточен на тестировании потребительских контрактов, поэтому используемый им подход довольно тяжеловесен. JAR-файлы с контрактами внутри должны быть развернуты в репозитории Maven — их недостаточно просто добавить в `classpath`. Управление взаимодействием, в котором применяются динамически генерируемые значения, тоже сопряжено с определенными трудностями. Поэтому более простым вариантом будет конфигурация заглушек прямо из теста.

Тест, к примеру, может сконфигурировать HTTP-заглушку с помощью специального языка `WireMock`, который называется `WireMock`. Тест сервиса, использующего `Eventuate Tram`, может аналогичным образом симулировать инфраструктуру для обмена сообщениями. Позже в этом разделе я покажу простую в применении библиотеку, которая этим занимается.

Разобравшись с тем, как проектировать компонентные тесты, может перейти к их созданию.

10.2.4. Написание компонентных тестов для сервиса `Order`

Как вы уже видели в этом разделе, компонентные тесты можно реализовать несколькими разными способами. Здесь мы обсудим компонентные тесты для сервиса `Order`, запущенного в виде контейнера `Docker`. Они будут задействовать внепроцессную стратегию. Вы увидите, как они будут запускать и останавливать контейнер `Docker`

с помощью дополнения Gradle. Я также покажу, как использовать Cucumber для выполнения сценариев на языке Gherkin, которые описывают желаемое поведение сервиса Order.

Структура компонентных тестов для сервиса Order показана на рис. 10.7. Тестовый класс `OrderServiceComponentTest` запускает Cucumber:

```
@RunWith(Cucumber.class)
@cucumberOptions(features = "src/component-test/resources/features")
public class OrderServiceComponentTest {
}
```

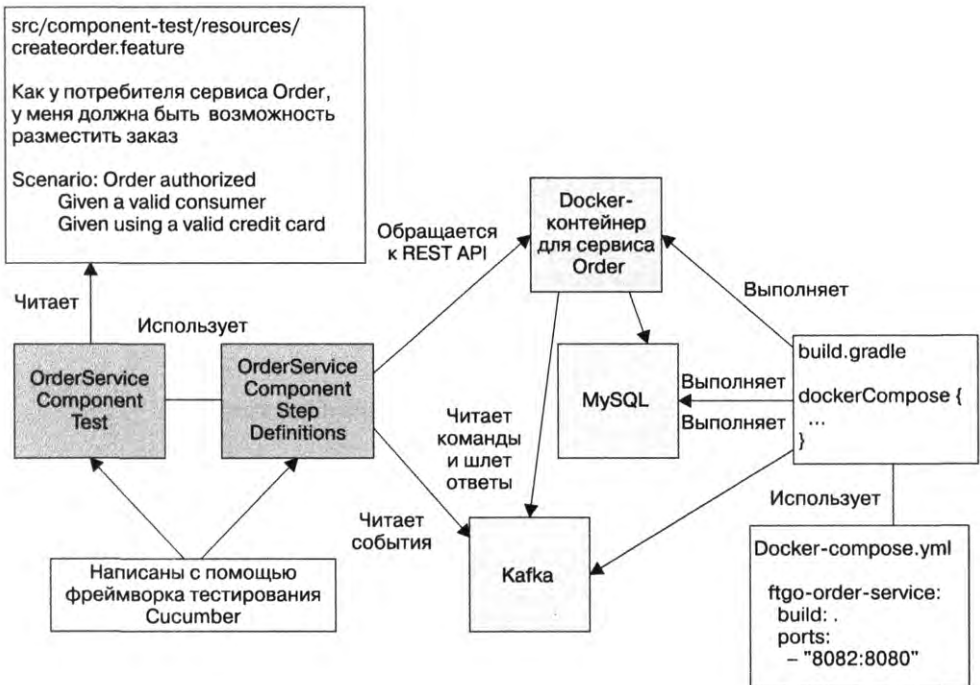


Рис. 10.7. Компонентные тесты для сервиса Order с помощью тестового фреймворка Cucumber выполняют тестовые сценарии, написанные на языке DSL для приемочного тестирования Gherkin. Тесты применяют Docker для выполнения сервиса Order и его инфраструктурных компонентов, таких как Apache Kafka и MySQL

Он имеет аннотацию `@CucumberOptions`, которая задает местоположение файлов с возможностями для Gherkin. Также он помечен аннотацией `@RunWith(Cucumber.class)`, которая заставляет JUNIT использовать средство выполнения тестов из состава Cucumber. Но, в отличие от тестовых классов, основанных на JUNIR, этот класс не содержит никаких тестовых методов. Для определения тестов он считывает возможности Gherkin и делает их исполняемыми с помощью класса `OrderServiceComponentTestStepDefinitions`.

Применение Cucumber в сочетании с фреймворком тестирования Spring Boot требует немного необычной структуры. Класс `OrderServiceComponentTestStepDefinitions` не тестовый, но все равно помечен аннотацией `@ContextConfiguration` из состава фреймворка тестирования Spring. Он создает контекст Spring-приложения, `ApplicationContext`, который определяет различные компоненты Spring, включая фиктивную инфраструктуру для обмена сообщениями. Давайте подробно рассмотрим пошаговые определения.

Класс `OrderServiceComponentTestStepDefinitions`

В основе тестов лежит класс `OrderServiceComponentTestStepDefinitions`. Он определяет значение каждого шага в компонентных тестах сервиса `Order`. В листинге 10.13 показан метод `usingCreditCard()`, который определяет значение шага `Given using ... credit card`.

Листинг 10.13. Метод `@Given useCreditCard()` определяет значение шага `Given using ... credit card`

```
@ContextConfiguration(classes =
    OrderServiceComponentTestStepDefinitions.TestConfiguration.class)
public class OrderServiceComponentTestStepDefinitions {

    ...

    @Autowired
    protected SagaParticipantStubManager sagaParticipantStubManager;

    @Given("using a(?:) (.*) credit card")
    public void useCreditCard(String ignore, String creditCard) {
        if (creditCard.equals("valid"))
            sagaParticipantStubManager
                .forChannel("accountingService")
                .when(AuthorizeCommand.class).replyWithSuccess();
            ← Возвращает успешный ответ
        else if (creditCard.equals("invalid"))
            sagaParticipantStubManager
                .forChannel("accountingService")
                .when(AuthorizeCommand.class).replyWithFailure();
            ← Возвращает сообщение об отказе
        else
            fail("Don't know what to do with this credit card");
    }
```

Этот метод использует вспомогательный класс `SagaParticipantStubManager`, который конфигурирует заглушки для участников повествования. С его помощью метод `useCreditCard()` подготавливает фиктивный сервис `Accounting`, который будет возвращать сообщение об успешном выполнении или отказе в зависимости от указанной банковской карты.

В листинге 10.14 представлен метод `placeOrder()`, определяющий шаг `When I place an order for Chicken Vindaloo at Ajanta`. Он обращается к REST API сервиса `Order`, чтобы создать заказ, и сохраняет ответ для последующей проверки.

Листинг 10.14. Метод `placeOrder()` определяет шаг When I place an order for Chicken Vindaloo at Ajanta

```
@ContextConfiguration(classes =
    OrderServiceComponentTestStepDefinitions.TestConfiguration.class)
public class OrderServiceComponentTestStepDefinitions {

    private int port = 8082;
    private String host = System.getenv("DOCKER_HOST_IP");

    protected String baseUrl(String path) {
        return String.format("http://%s:%s%s", host, port, path);
    }

    private Response response;

    @When("I place an order for Chicken Vindaloo at Ajanta")
    public void placeOrder() {
        response = given().
            body(new CreateOrderRequest(consumerId,
                RestaurantMother.AJANTA_ID, Collections.singletonList(
                    new CreateOrderRequest.LineItem(
                        RestaurantMother.CHICKEN_VINDALOO_MENU_ITEM_ID,
                        OrderDetailsMother.CHICKEN_VINDALOO_QUANTITY)))
            .contentType("application/json").
            when().
            post(baseUrl("/orders"));
    }
```

Обращается к REST API сервиса Order, чтобы создать заказ

Вспомогательный метод `baseUrl()` возвращает URL-адрес сервиса Order.

В листинге 10.15 показан метод `theOrderShouldBe()`, который определяет значение шага Then the order should be... Он проверяет успешность заказа и то, находится ли он в ожидаемом состоянии.

Листинг 10.15. Метод `@Then theOrderShouldBe()` проверяет, был ли HTTP-запрос успешным

```
@ContextConfiguration(classes =
    OrderServiceComponentTestStepDefinitions.TestConfiguration.class)
public class OrderServiceComponentTestStepDefinitions {

    @Then("the order should be (.*)")
    public void theOrderShouldBe(String desiredOrderState) {

        Integer orderId =
            this.response. then(). statusCode(200).
                extract(). path("orderId");

        assertNotNull(orderId);

        eventually(() -> {
            String state = given().
                when().
                get(baseUrl("/orders/" + orderId)).
    }
```

Проверяет успешность создания заказа

```

        then().
        statusCode(200)
        .extract().
            path("state");
    assertEquals(desiredOrderState, state); ← Проверяет состояние заказа
});
}
]

```

Утверждение об ожидаемом состоянии завернуто в вызов `eventually()`, который выполняет его несколько раз.

В листинге 10.16 можно видеть метод `verifyEventPublished()`, который определяет шаг `And an ... event should be published`. Он проверяет, было ли опубликовано ожидаемое событие.

Листинг 10.16. Класс пошагового определения Cucumber для компонентных тестов сервиса Order

```

@ContextConfiguration(classes =
    OrderServiceComponentTestStepDefinitions.TestConfiguration.class)
public class OrderServiceComponentTestStepDefinitions {

    @Autowired
    protected MessageTracker messageTracker;

    @And("an (.*?) event should be published")
    public void verifyEventPublished(String expectedEventClass) throws ClassNotFoundException {
        messageTracker.assertDomainEventPublished("net.chrisrichardson.ftgo.order
            service.domain.Order",
            (Class<DomainEvent>)Class.forName("net.chrisrichardson.ftgo.order
            service.domain." + expectedEventClass));
    }
    ....
}

```

Метод `verifyEventPublished()` использует вспомогательный тестовый класс `MessageTracker`, который записывает события, опубликованные во время теста. Экземпляры этого класса и `SagaParticipantStubManager` создаются классом `TestConfiguration @Configuration`.

Мы рассмотрели пошаговое определение. Теперь можно перейти к выполнению компонентных тестов.

Выполнение компонентных тестов

Эти тесты довольно медленные, поэтому их не стоит выполнять в рамках команды `./gradlew test`. Вместо этого разместим их код в отдельном каталоге `src/component-test/java` и будем запускать их как `./gradlew componentTest`. Конфигурация Gradle находится в файле `ftgo-order-service/build.gradle`.

Для запуска сервиса `Order` и его зависимостей тесты используют `Docker`. Как вы узнаете из главы 12, контейнер `Docker` представляет собой легковесный механизм виртуализации операционной системы, который позволяет развертывать экземпляры сервисов в изолированной среде. Чрезвычайно полезным инструментом является `Docker Compose`, с помощью которого можно определить набор контейнеров и запускать/останавливать их как единое целое. В корневом каталоге приложения `FTGO` находится файл `docker-compose`, который описывает контейнеры для всех сервисов и инфраструктурных компонентов.

Мы можем воспользоваться дополнением `Docker Compose` для `Gradle`, чтобы запускать контейнеры перед выполнением тестов и останавливать их после завершения тестирования:

```
apply plugin: 'docker-compose'

dockerCompose.isRequiredBy(componentTest)
componentTest.dependsOn(assemble)

dockerCompose {
    startedServices = [ 'ftgo-order-service' ]
}
```

Приведенный фрагмент конфигурации `Gradle` делает две вещи. Во-первых, он настраивает дополнение `Gradle Docker Compose` для выполнения компонентных тестов и запуска сервиса `Order` вместе с инфраструктурными компонентами, от которых он зависит согласно конфигурации. Во-вторых, он делает так, чтобы тест `componentTest` зависел от этапа `assemble`. Благодаря этому будет предварительно собираться `JAR`-файл, который нужен образу `Docker`. Подготовив все это, мы можем запустить компонентные тесты с помощью следующих команд:

```
./gradlew :ftgo-order-service:componentTest
```

Эти команды работают несколько минут и выполняют следующие действия.

1. Сборка сервиса `Order`.
2. Запуск сервиса и его инфраструктурных зависимостей.
3. Запуск тестов.
4. Остановка запущенных сервисов.

Теперь вы знаете, как писать тесты для отдельных сервисов. Пришло время протестировать приложение целиком.

10.3. Написание сквозных тестов

Компонентные тесты тестируют каждый сервис по отдельности. Сквозные тесты тестируют приложение целиком. Сквозные тесты находятся на вершине пирамиды тестов (рис. 10.8), так как они (повторяйте за мной) медленные и ненадежные, а на их разработку уходит много времени.

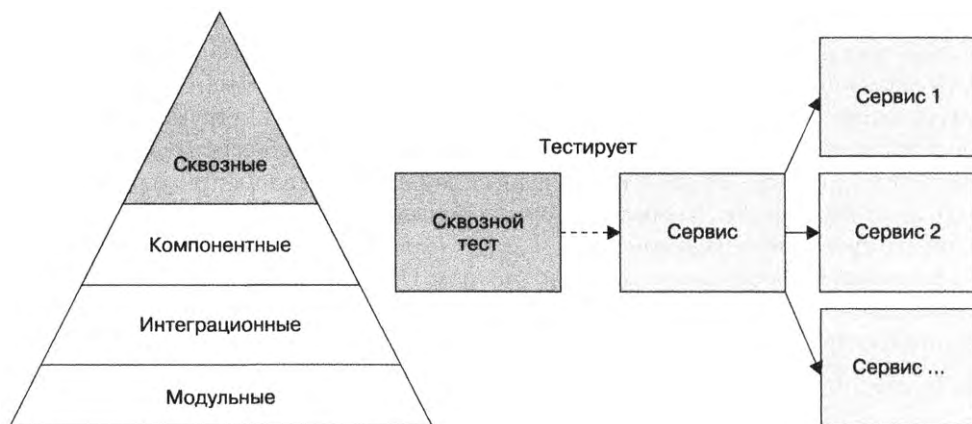


Рис. 10.8. Сквозные тесты располагаются на вершине пирамиды тестирования. Они медленные и ненадежные, а на их разработку затрачивается много времени. Количество сквозных тестов следует минимизировать

Сквозные тесты состоят из множества элементов. Они требуют развертывания большого количества сервисов и инфраструктурных компонентов, что их замедляет. К тому же, если для работы теста нужно развернуть большое количество сервисов, вполне вероятно, что развертывание одного из них окажется неудачным, из-за чего тест окажется ненадежным. Так что вы должны свести количество сквозных тестов к минимуму.

10.3.1. Проектирование сквозных тестов

Как я уже объяснил, следует писать как можно меньше таких тестов. Лучше сделать так, чтобы они были основаны на *пользовательских путешествиях*, которые описывают перемещения пользователя по системе. Например, проверку создания, просмотра и отмены заказа можно объединить в один тест. Этот подход существенно уменьшает количество тестов, которые вам нужно написать, и сокращает время их выполнения.

10.3.2. Написание сквозных тестов

Сквозные тесты, как и приемочное тестирование, рассмотренное в разделе 10.2, сосредоточены на бизнес-аспектах приложения. Их лучше писать на высокоуровневом языке DSL, доступном для понимания людям, которые занимаются бизнес-процессами. Вы можете, к примеру, создавать сквозные тесты на языке Gherkin и выполнять их с помощью Cucumber. Это продемонстрировано в листинге 10.17. Код напоминает приемочные тесты, которые мы рассматривали ранее. Основное отличие состоит в том, что вместо одного выражения `Then` здесь имеется несколько действий.

Листинг 10.17. Спецификация пользовательского путешествия на языке Gherkin**Feature:** Place Revise and Cancel

As a consumer of the Order Service
 I should be able to place, revise, and cancel an order

Scenario: Order created, revised, and cancelled

Given a valid consumer

Given using a valid credit card

Given the restaurant is accepting orders

When I place an order for Chicken Vindaloo at Ajanta ← Создать заказ

Then the order should be APPROVED

Then the order total should be 16.33

And when I revise the order by adding 2 vegetable samosas ← Пересмотреть заказ

Then the order total should be 20.97

And when I cancel the order

Then the order should be CANCELLED ← Отменить заказ

Этот сценарий размещает заказ, пересматривает и затем отменяет его. Посмотрим, как его выполнить.

10.3.3. Выполнение сквозных тестов

Сквозные тесты должны запускать целое приложение, включая все инфраструктурные компоненты. Как вы видели в разделе 10.2, дополнение Gradle Docker Compose предоставляет удобный способ сделать это. Однако вместо выполнения отдельных сервисов приложения файл Docker Compose должен запустить их все.

Мы уже познакомились с разными аспектами проектирования и написания сквозных тестов. Теперь рассмотрим конкретный пример.

Сквозные тесты для приложения FTGO находятся в модуле `ftgo-end-to-end-test`. Их реализация похожа на код компонентных тестов, рассмотренных в разделе 10.2. Они написаны на языке Gherkin и выполняются с помощью Cucumber. Перед выполнением тестов дополнение Gradle Docker Compose запускает контейнеры. На все это уходит 4–5 мин.

На первый взгляд может показаться, что это не так уж и долго, но мы имеем дело с относительно простым приложением, которое состоит из горстки контейнеров и тестов. Представьте, что бы было, если бы количество контейнеров и тестов исчислялось сотнями! Тестирование могло бы занять довольно много времени. Поэтому лучше сосредоточиться на написании тестов, находящихся на более низких ступенях пирамиды.

Резюме

- Используйте контракты (примеры сообщений) для тестирования взаимодействия между сервисами. Пишите тесты, которые проверяют адаптеры сервисов на соответствие контрактам, не запуская сами сервисы и их зависимости.

- ❑ Для проверки поведения сервиса через его API пишите компонентные тесты. Для их упрощения и ускорения сервисы следует тестировать отдельно друг от друга, задействуя заглушки вместо зависимостей.
- ❑ Сквозные тесты медлительны и ненадежны, а поэтому отнимают много времени. Чтобы свести их количество к минимуму, пишите их на основе пользовательских путешествий. Пользовательское путешествие симулирует перемещения пользователя по приложению и проверяет высокоуровневое поведение довольно крупных аспектов функциональности. Чем меньше тестов, тем меньше накладных расходов (например, на их подготовку), что ускоряет тестирование.

Разработка сервисов, готовых к промышленному использованию

В этой главе

- Разработка безопасных сервисов.
- Вынесение конфигурации вовне.
- Применение шаблонов наблюдаемости:
 - ✓ API для проверки работоспособности;
 - ✓ агрегация журналов;
 - ✓ распределенная трассировка;
 - ✓ отслеживание исключений;
 - ✓ показатели приложения;
 - ✓ ведение журнала аудита.
- Упрощение разработки сервисов путем применения шаблона «Шасси микросервисов».

Мэри и ее команда, по их собственным ощущениям, овладели такими аспектами разработки, как декомпозиция сервисов, межсервисное взаимодействие, управление транзакциями, проектирование запросов и бизнес-логики и тестирование. Они были уверены в том, что им по силам разработать сервисы, удовлетворяющие их функциональным требованиям. Однако сервис можно считать готовым к развертыванию только в том случае, если ему присущи три критически важные качественные характеристики: безопасность, конфигурируемость и наблюдаемость.

Первой качественной характеристикой является *безопасность приложения*. Если вы не хотите обнаружить название своей компании в новостных заголовках,

сообщающих об утечке данных, ваше приложение должно быть безопасным. К счастью, с точки зрения безопасности микросервисная архитектура и монолитное приложение имеют много общего. Команда FTGO знала, что многие навыки, приобретенные в ходе разработки монолита, пригодятся и в работе над микросервисами. Однако некоторые аспекты безопасности на уровне приложения все же различаются. Например, в микросервисной архитектуре необходимо реализовать механизм для передачи пользовательских данных от одного сервиса к другому.

Вторая качественная характеристика, которую вы должны учитывать, — *конфигурируемость сервисов*. Сервис обычно имеет одну или несколько внешних зависимостей, таких как брокеры сообщений или базы данных. Сетевое размещение и учетные данные каждого внешнего компонента зависят от того, в какой среде выполняется сервис. Вы не можете хранить конфигурационные свойства прямо в его коде. Вместо этого должны использовать внешний механизм, который предоставляет сервису конфигурацию на этапе выполнения.

Третья качественная характеристика — *наблюдаемость*. Команда FTGO реализовала мониторинг и ведение журнала для существующего приложения. Однако распределенность микросервисной архитектуры создает дополнительные сложности. Каждый запрос обрабатывается API-шлюзом и как минимум одним сервисом. Представьте, к примеру, что вы пытаетесь определить, какой из шести сервисов приводит к проблемам с латентностью. Или что вам нужно разобраться в том, как обрабатывается запрос, когда журнальные записи разбросаны по пяти разным сервисам. Чтобы было легче понимать поведение приложения и диагностировать проблемы, вы должны реализовать несколько шаблонов наблюдаемости.

Я начну эту главу с описания того, как обеспечить безопасность в микросервисной архитектуре. Затем мы обсудим проектирование конфигурируемых сервисов. Будут рассмотрены несколько механизмов конфигурации. После этого мы поговорим о том, как сделать ваши сервисы более простыми для понимания и диагностики с помощью шаблонов наблюдаемости. В конце я продемонстрирую простую реализацию этих и других аспектов на примере сервисов, основанных на фреймворке микросервисного шасси.

Начнем с безопасности.

11.1. Разработка безопасных сервисов

Кибербезопасность превратилась в насущную проблему для любой организации. Почти каждый день в новостях можно видеть сообщения о том, как хакеры похитили данные компании. Чтобы разрабатывать безопасное программное обеспечение и не попадать в новостные ленты, организация должна решить широкий диапазон проблем, связанных с безопасностью, включая физическую безопасность оборудования, шифрование данных в процессе передачи и при хранении, аутентификацию и авторизацию, а также политику исправления программных уязвимостей. Большинство этих проблем в равной степени относятся как к монолитной, так и к микросервисной архитектуре. Этот раздел посвящен тому, как микросервисы влияют на безопасность всего приложения.

Разработчик в первую очередь несет ответственность за то, как реализованы четыре аспекта безопасности.

- ❑ *Аутентификация.* Проверяет подлинность программы или человека (*субъекта безопасности*), которые пытаются получить доступ к приложению. Приложение обычно проверяет учетные данные субъекта, такие как ID и пароль или API-ключ и секретный токен.
- ❑ *Авторизация.* Проверяет, позволено ли субъекту выполнять запрошенную операцию с заданными данными. Приложения часто применяют безопасность на основе ролей в сочетании со списками управления доступом (Access Control List, ACL). Каждый пользователь получает одну или несколько ролей, которые дают им право вызывать определенные операции. Списки ACL разрешают пользователям или ролям выполнять операции с определенным бизнес-объектом или агрегатом.
- ❑ *Аудит.* Отслеживает операции, выполняемые субъектом, чтобы обнаруживать проблемы с безопасностью, помогать службе поддержки и обеспечивать соблюдение нормативно-правовых норм.
- ❑ *Безопасное межпроцессное взаимодействие.* В идеале любое взаимодействие внутри сервисов и за их пределами должно производиться поверх TLS (Transport Layer Security — протокол защиты транспортного уровня). Для межпроцессного взаимодействия может даже понадобиться аутентификация.

Аудит подробно описывается в разделе 11.3, а защиту межпроцессного взаимодействия мы затронем в ходе обсуждения сетей сервисов в подразделе 11.4.1. Здесь же сосредоточимся на реализации аутентификации и авторизации.

Вначале я объясню, как безопасность была реализована в монолитной версии приложения FTGO. Затем перечислю трудности, возникающие при обеспечении безопасности в микросервисной архитектуре, и методики, которые хорошо подходят для монолита, но не годятся для микросервисов. После этого мы займемся реализацией безопасности в микросервисной архитектуре.

Начнем с того, как обеспечивается безопасность в монолитной версии FTGO.

11.1.1. Обзор безопасности в традиционном монолитном приложении

У приложения FTGO есть несколько разновидностей живых пользователей — клиенты, курьеры и работники ресторанов. Для доступа к приложению они используют браузерный и мобильный веб-интерфейсы. Все пользователи FTGO должны вначале войти в систему. На рис. 11.1 показано, как аутентифицируются и выполняют запросы клиенты монолитного приложения FTGO.

Когда пользователь входит в систему со своими идентификатором и паролем, клиент отправляет приложению FTGO POST-запрос, содержащий его учетные данные. Приложение проверяет эти данные и возвращает клиенту токен сеанса. Клиент включает этот токен во все свои последующие запросы.

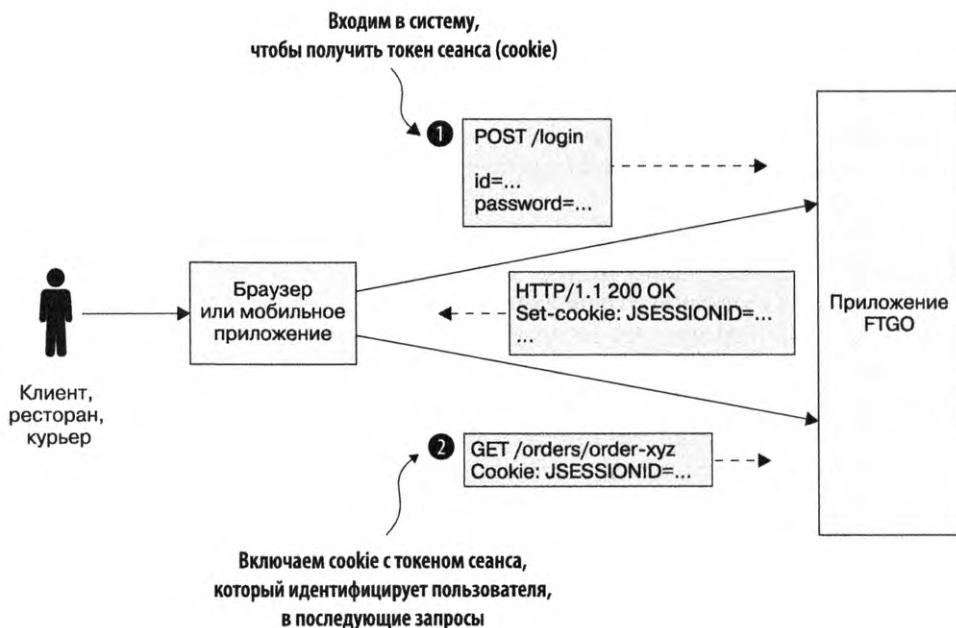


Рис. 11.1. Вначале клиент приложения FTGO входит в систему, чтобы получить токен сеанса, в качестве которого часто выступает cookie. Клиент включает этот токен в каждый последующий запрос, отправляемый приложению

На рис. 11.2 представлены общие принципы обеспечения безопасности в приложении FTGO. Оно написано на Java с применением фреймворка Spring Security, но для описания его архитектуры я буду использовать общие понятия, пригодные и для других технологий, таких как Passport для NodeJS.

Использование фреймворка безопасности

Корректная реализация аутентификации и авторизации — задача не из простых. Для этого лучше применять проверенный временем фреймворк безопасности. Конкретный выбор зависит от стека технологий вашего приложения. Среди популярных фреймворков можно выделить следующие:

- Spring Security (<https://spring.io/projects/spring-security>) — популярный фреймворк для Java-приложений. Он имеет развитый механизм, реализующий аутентификацию и авторизацию;
- Apache Shiro (<https://shiro.apache.org>) — еще один фреймворк для Java;
- Passport (<http://www.passportjs.org>) — популярный фреймворк безопасности для NodeJS-приложений, в первую очередь предназначенный для аутентификации.

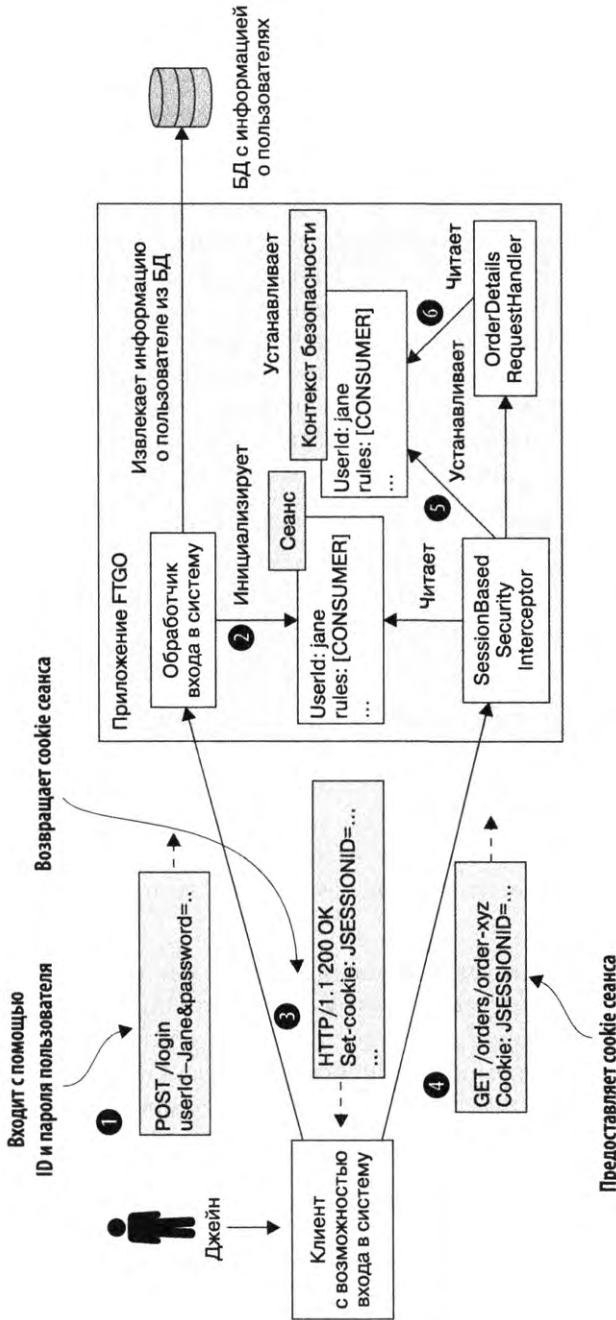


Рис. 11.2. Когда клиент приложения FTGO запрашивает вход в систему, LoginHandler аутентифицирует пользователя, инициализирует сеанс с информацией о нем и возвращает cookie с токеном сеанса, который безопасно его идентифицирует. Затем, когда клиент выполняет запрос с этим токеном, SessionBasedSecurityInterceptor извлекает сведения о пользователе из заданного сеанса и устанавливает контекст безопасности. Обработчик запросов, такой как OrderDetailsRequestHandler, извлекает пользовательские данные из контекста безопасности

Ключевую роль в архитектуре безопасности играет сеанс, который хранит ID и роль субъекта. FTGO — это традиционное приложение на основе Java EE, поэтому сеанс представлен объектом `HttpSession`, находящимся в оперативной памяти. *Сеанс* идентифицируется соответствующим токеном, который клиент включает в каждый запрос. Обычно это непрозрачное значение наподобие надежно зашифрованного случайного числа. Токен сеанса в приложении FTGO имеет вид HTTP-cookie `JSESSIONID`.

Еще одним ключевым аспектом безопасности является *контекст*, который хранит информацию о пользователе, выполняющем текущий запрос. Фреймворк Spring Security использует стандартный для Java EE подход, храня контекст в статической переменной, локальной для потока приложения. К ней легко может обратиться любой код, вовлеченный в обработку запроса. Чтобы получить информацию о текущем пользователе, такую как идентификатор и роль, обработчик может вызвать `SecurityContextHolder.getContext().getAuthentication()`. Для сравнения: фреймворк Passport хранит контекст безопасности в атрибуте запроса `user`.

Последовательность событий, представленных на рис. 11.2, выглядит так.

1. Клиент шлет приложению FTGO запрос на вход в систему.
2. Запрос входа в систему обрабатывается объектом `LoginHandler`, который проверяет учетные данные, создает сеанс и сохраняет туда информацию о субъекте.
3. `LoginHandler` возвращает клиенту токен сеанса.
4. Клиент включает токен сеанса в запросы, выполняющие операции.
5. Эти запросы вначале обрабатываются перехватчиком `SessionBasedSecurityInterceptor`. Он аутентифицирует каждый запрос, проверяя токен сеанса, и устанавливает контекст безопасности. Контекст безопасности описывает субъект и его роли.
6. С помощью контекста безопасности обработчик запросов определяет, разрешено ли пользователю выполнять запрошенную операцию, и получает его уникальный идентификатор.

Приложение FTGO использует авторизацию *на основе ролей*. Оно поддерживает несколько ролей, которые относятся к разным категориям пользователей: `CONSUMER`, `RESTAURANT`, `COURIER` и `ADMIN`. А еще применяет декларативный механизм безопасности из состава Spring Security, чтобы ограничить доступ к URL-адресам и методам сервисов для определенных ролей. Кроме того, роли интегрированы в бизнес-логику. Например, клиенты видят только свои заказы, тогда как у администраторов есть доступ к заказам всех пользователей.

Данная архитектура — лишь один из способов обеспечения безопасности в монолитной версии приложения FTGO. К примеру, из-за того, что сеансы хранятся в оперативной памяти, все запросы в рамках отдельного сеанса должны быть направлены к одному и тому же экземпляру приложения. Это требование усложняет балансирование нагрузки и администрирование. Например, это требует реализации

дренажного механизма, который, прежде чем остановить сервер с экземпляром приложения, ждет истечения срока действия всех его сеансов. Чтобы избежать этой проблемы, сеансы можно хранить в базе данных.

В некоторых случаях от сеансов на стороне сервера можно полностью избавиться. Например, клиенты многих приложений предоставляют свои учетные данные, такие как API-ключ и секретный токен, в каждом API-запросе. Благодаря этому отпадает необходимость в поддержании сеанса на серверной стороне. Как вариант, приложение может хранить состояние сеанса в его токене. Позже я покажу один из способов, как это можно сделать. Но для начала рассмотрим трудности обеспечения безопасности в микросервисной архитектуре.

11.1.2. Обеспечение безопасности в микросервисной архитектуре

Микросервисные приложения являются распределенными. Каждый запрос обрабатывается API-шлюзом и как минимум одним сервисом. Возьмем, к примеру, запрос `getOrderDetails()`, который мы обсуждали в главе 8. Для его обработки API-шлюз обращается к нескольким сервисам, таким как `Order`, `Kitchen` и `Accounting`. Каждый из них должен реализовать некоторые аспекты безопасности. Скажем, сервис `Order` должен возвращать клиентам только их собственные заказы, что требует сочетания аутентификации и авторизации. Чтобы обеспечить безопасность в микросервисной архитектуре, нам нужно определиться с тем, кто отвечает за аутентификацию пользователя, а кто — за его авторизацию.

Одна из сложностей реализации безопасности в микросервисном приложении связана с тем, что мы не можем просто скопировать соответствующие решения из монолитной архитектуры. Это происходит из-за того, что механизмы безопасности в монолитных приложениях имеют два аспекта, которые совершенно не подходят для микросервисов.

- ❑ *Контекст безопасности в оперативной памяти.* Хранение контекста безопасности в оперативной памяти, например внутри потока, для раздачи данных о пользователе. Сервисы не способны разделять память, поэтому они не могут использовать подобного рода механизм. Микросервисная архитектура требует другого подхода к передаче пользовательских данных от одного сервиса к другому.
- ❑ *Централизованный сеанс.* Поскольку контекст безопасности нельзя размещать в памяти, это ограничение распространяется и на сеанс. Теоретически разные сервисы могли бы получать доступ к сеансу, который хранится в базе данных, но это нарушило бы принцип слабой связанности. Для микросервисной архитектуры нужен другой механизм сеансов.

Начнем исследование безопасности в микросервисной архитектуре с рассмотрения аутентификации.

Выполнение аутентификации в API-шлюзе

Аутентификацию пользователей можно реализовать несколькими способами. Например, эту функцию могут взять на себя отдельные сервисы. Проблема этого подхода в том, что он допускает попадание неаутентифицированных запросов во внутреннюю сеть. К тому же каждая команда разработчиков должна обеспечить надлежащую безопасность своих сервисов. В итоге существенно возрастает риск возникновения уязвимостей.

Еще одна проблема выполнения аутентификации на уровне сервисов связана с тем, что разные клиенты могут по-разному себя аутентифицировать. Клиенты, работающие исключительно через API, предоставляют учетные данные в каждом запросе (так, например, делается при HTTP-аутентификации). Другие клиенты могут сначала войти в систему, а затем прилагать токен сеанса к каждому вызову. Мы не хотим, чтобы сервисы отвечали за поддержку разнообразных механизмов аутентификации.

Лучше сделать так, чтобы любой запрос, прежде чем попасть к сервису, аутентифицировался API-шлюзом. Благодаря такому централизованному подходу мы можем сосредоточиться на одном участке приложения, что существенно снижает риск возникновения уязвимостей. Еще одно преимущество состоит в том, что за работу с разными механизмами аутентификации отвечает лишь API-шлюз. Сервисы ограждены от всех этих нюансов.

Принцип работы этого подхода показан на рис. 11.3. Клиенты аутентифицируются API-шлюзом и включают свои учетные данные в каждый запрос. Клиенты, которым нужно сначала войти в систему, шлют API-шлюзу сведения о пользователе методом POST, получая в ответ токен сеанса. Аутентифицировав запрос, API-шлюз обращается к одному или нескольким сервисам.

Шаблон «Токен доступа»

API-шлюз передает токен с информацией о пользователе, включая его идентификатор и роли, сервисам, к которым тот обращается. См. microservices.io/patterns/security/access-token.html.

Сервис, к которому обратился API-шлюз, должен опознать субъекта, выполняющего запрос. Он также должен проверить, был ли этот запрос аутентифицирован. Для этого при каждом обращении к сервису API-шлюз указывает токен. С помощью токена сервис проверяет подлинность запроса и извлекает информацию о субъекте. API-шлюз может выдавать этот токен и клиентам, ориентированным на сеансы, в этом случае он становится токеном сеанса.

Для API-клиентов последовательность событий выглядит так.

1. Клиент делает запрос, содержащий учетные данные.
2. API-шлюз аутентифицирует учетные данные, создает токен безопасности и передает его сервису (-ам).

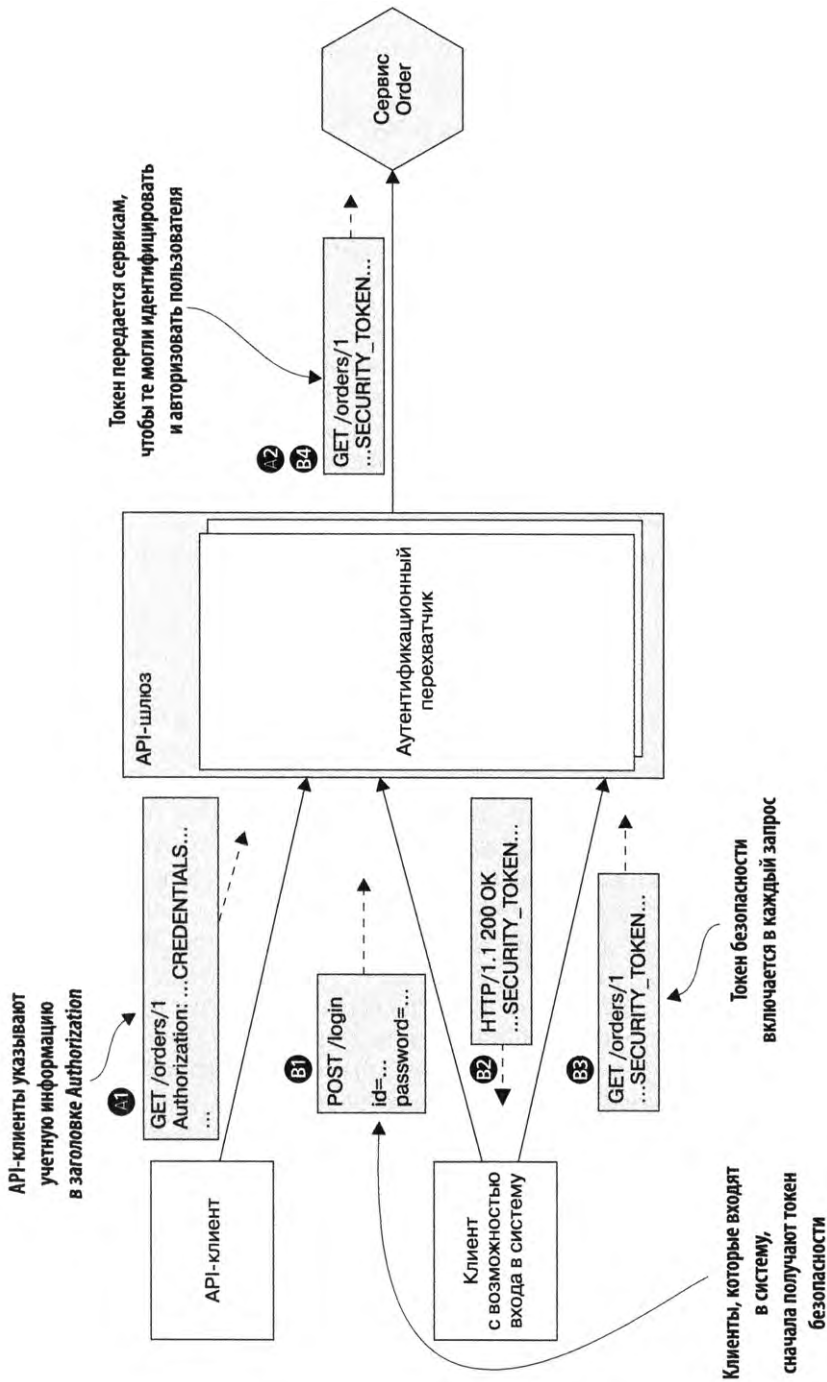


Рис. 11.3. API-шлюз получает клиентские запросы и указывает токен безопасности при обращении к сервисам. Сервисы задействуют этот токен для получения информации о субъекте. API-шлюз может использовать токен безопасности в качестве токена сеанса

Клиенты, которые входят в систему, проходят через такую цепочку событий.

1. Клиент делает запрос на вход в систему, содержащий учетные данные.
2. API-шлюз возвращает токен безопасности.
3. Клиент включает токен безопасности в запрос на выполнение операции.
4. API-шлюз проверяет токен безопасности и направляет запрос к сервису (-ам).

Чуть позже в этой главе я опишу, как реализовывать токены, но сначала рассмотрим другой ключевой аспект безопасности — авторизацию.

Выполнение авторизации

Аутентификация учетных данных клиентов — важная задача, но этого недостаточно. Приложение должно также реализовать механизм авторизации, который проверяет, позволено ли клиенту выполнять запрошенную операцию. Например, в приложении FTGO операцию `getOrderDetails()` может вызывать только клиент, разместивший соответствующий заказ (пример безопасности уровня экземпляра), и работник службы поддержки, который ему помогает.

Авторизацию можно реализовать в API-шлюзе. Таким образом мы можем, к примеру, открыть доступ к точке `GET /orders/{orderId}` только тем пользователям, которые являются клиентами или работниками службы поддержки. Если пользователю не разрешено обращаться к определенному пути, API-шлюз может отклонить его запрос до того, как он будет направлен к сервису. Как и в случае с аутентификацией, централизованное размещение авторизации в рамках API-шлюза снижает риск возникновения уязвимостей. Для этого можно использовать фреймворк безопасности наподобие Spring Security.

Одним из недостатков этого подхода является риск привязки API-шлюза к сервисам, что потребует синхронизации их обновлений. Более того, API-шлюз обычно способен реализовать только ролевой доступ к URL-адресам. Как правило, списки ACL, которые управляют доступом к отдельным доменным объектам, реализуют в другом месте, поскольку для этого требуется хорошее знание доменной логики сервиса.

Авторизацию можно реализовать и внутри сервисов. Сервис может выполнять ролевую авторизацию для URL-адресов и своих методов. Он также может поддерживать списки ACL для управления доступом к агрегатам. Сервис `Order`, к примеру, может реализовать механизм авторизации на основе ролей и ACL для контроля за доступом к заказам. В этом случае другие сервисы приложения FTGO реализуют аналогичную авторизационную логику.

Использование JWT для передачи ролей и учетных данных пользователя

При обеспечении безопасности в микросервисной архитектуре вам нужно решить, с помощью какого рода токена API-шлюз будет передавать сервисам пользовательскую информацию. Существует два типа токенов, из которых вы можете выбрать.

Один из вариантов — *непрозрачные* токены, которые обычно имеют формат UUID. Их недостатком является понижение производительности и доступности, а также увеличение латентности. Это связано с тем, что получатели таких токенов должны делать синхронные RPC-вызовы к сервису безопасности, чтобы проверить их корректность и извлечь информацию о пользователе.

Альтернативным подходом, который устраняет необходимость в обращении к сервису безопасности, является применение *прозрачных* токенов, содержащих пользовательские данные. В качестве одного из популярных стандартов для такого рода токенов можно привести JWT (JSON Web token). JWT — это стандартный способ безопасного представления между двумя сторонами такой информации, как идентификаторы и роли пользователя. Токен JWT содержит так называемую полезную нагрузку (payload) в виде JSON-объекта со сведениями о пользователе, такими как его идентификаторы и роли, а также другими метаданными, например сроком годности. Он подписывается секретным ключом, известным только его создателю (например, API-шлюзу) и получателю (например, сервису). Благодаря этому ключу посторонние не могут подделать токен JWT.

Из-за автономности у токена JWT есть одна проблема: его нельзя отозвать. После проверки подписи и срока годности сервис обязательно выполнит запрошенную операцию. Таким образом, вы не можете отозвать отдельный токен, который попал в руки злоумышленнику. Чтобы с этим бороться, можно устанавливать короткие сроки годности — это ограничит задумавшим недоброе пространство для маневра. Недостаток этого подхода состоит в том, что приложение должно постоянно переиздавать токены JWT, чтобы поддерживать сеанс в активном состоянии. К счастью, эта и многие другие проблемы уже решены в стандарте безопасности OAuth 2.0. Посмотрим, как это работает.

Использование OAuth 2.0 в микросервисной архитектуре

Представьте, что в приложении FTGO нужно реализовать сервис `User`, который управляет базой данных с пользовательской информацией, такой как учетные данные и роли. API-шлюз обращается к сервису `User`, чтобы аутентифицировать клиентский запрос и получить JWT. Вы могли бы спроектировать API сервиса `User` с помощью любимого веб-фреймворка. Но это общая функциональность, которая не имеет прямого отношения к приложению FTGO, — создание такого сервиса было бы неэффективной тратой времени разработчиков.

К счастью, вам не нужно разрабатывать такого рода инфраструктуру безопасности. Можно воспользоваться готовым сервисом или фреймворком, который реализует стандарт OAuth 2.0. OAuth 2.0 — это протокол авторизации, который изначально создавался для того, чтобы пользователи облачных сервисов, таких как GitHub или Google, могли открывать доступ к своей информации сторонним приложениям, не требуя ввода пароля. Например, с помощью OAuth 2.0 можно дать доступ к своему репозиторию на GitHub стороннему облачному сервису непрерывной интеграции.

Изначально идея OAuth 2.0 заключалась в авторизации доступа к публичным облачным приложениям, но вы можете задействовать эту технологию для

аутентификации и авторизации в своих проектах. Взглянем на то, как OAuth 2.0 можно интегрировать в микросервисную архитектуру.

Об OAuth 2.0

OAuth 2.0 — это сложная тема. В этой главе содержатся лишь краткий обзор данного стандарта и описание того, как его можно использовать в микросервисной архитектуре. Больше информации об OAuth 2.0 приведено в онлайн-книге Аарона Парецки (Aaron Parecki) *OAuth 2.0 Servers* (www.oauth.com). Эта тема рассматривается также в главе 7 книги *Spring Microservices in Action* (Manning, 2017; livebook.manning.com/#!/book/spring-microservices-in-action/chapter-7/).

Стандарт OAuth 2.0 основан на следующих концепциях.

- ❑ *Сервер авторизации* — предоставляет API для аутентификации пользователей и получения токенов доступа и обновления. Spring OAuth — хороший пример фреймворка для построения сервера авторизации OAuth 2.0.
- ❑ *Токен доступа* — токен, дающий доступ к *серверу ресурсов*. Его формат зависит от реализации. Но некоторые фреймворки, например Spring OAuth, используют для этого токены JWT.
- ❑ *Токен обновления* — долгоживущий токен с возможностью отзыва, с помощью которого клиент получает *токен доступа*.
- ❑ *Сервер ресурсов* — задействует токен доступа для авторизации. В микросервисной архитектуре серверами ресурсов выступают сами сервисы.
- ❑ *Клиент* — хочет получить доступ к *серверу ресурсов*. В микросервисной архитектуре роль клиента OAuth 2.0 играет API-шлюз.

Позже в этом разделе я покажу, как поддерживать клиенты, требующие входа в систему. Но сначала поговорим о том, как аутентифицировать API-клиенты.

На рис. 11.4 показано, как API-шлюз выполняет аутентификацию запроса, отправленного API-клиентом. Для этого он обращается к серверу авторизации OAuth 2.0, который возвращает токен доступа. Затем API-шлюз использует этот токен, чтобы сделать один или несколько запросов к сервисам.

Последовательность событий выглядит так.

1. Клиент делает запрос, предоставляя свои учетные данные в процессе HTTP-аутентификации.
2. API-шлюз делает запрос типа OAuth 2.0 Password Grant (www.oauth.com/oauth2-servers/access-tokens/password-grant/) к серверу аутентификации OAuth 2.0.
3. Сервер аутентификации проверяет учетные данные API-клиента и возвращает токены доступа и обновления.
4. API-шлюз включает токен доступа в запросы, которые он отправляет сервисам. Сервис проверяет токен доступа и использует его для авторизации запроса.

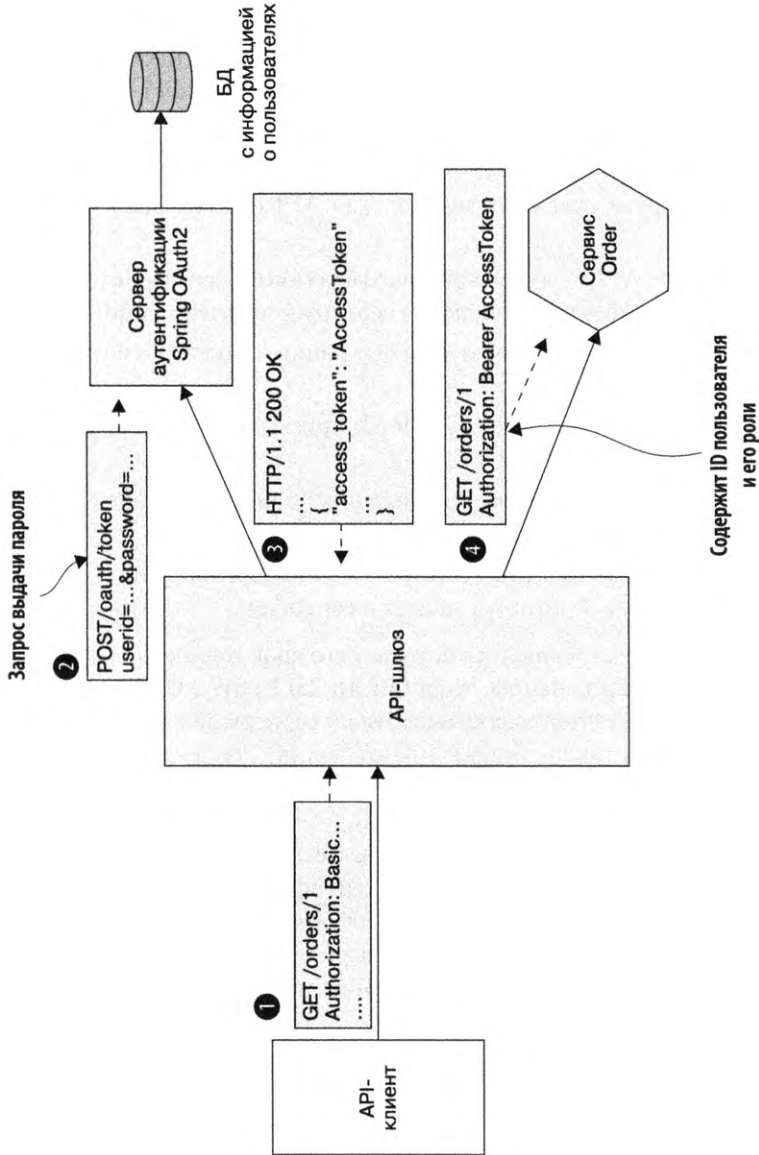


Рис. 11.4. API-шлюз аутентифицирует API-клиент, запрашивая сервер аутентификации о выдаче пароля. Сервер возвращает токен доступа, который API-шлюз передает сервисам. Сервис проверяет подпись токена и извлекает из него информацию о пользователе, включая его учетные данные и роли

API-шлюз, основанный на OAuth 2.0, может задействовать токен доступа в качестве токена сеанса, чтобы аутентифицировать клиенты соответствующего типа. Более того, когда заканчивается срок годности токена доступа, шлюз может получить новый, используя токен обновления. На рис. 11.5 показано, как API-шлюз применяет OAuth 2.0 для работы с клиентами, ориентированными на сеансы. Чтобы инициализировать сеанс, API-клиент передает свои учетные данные методом POST конечной точке API-шлюза `/login`. Шлюз возвращает клиенту токены доступа и обновления, а тот указывает их при обращении к шлюзу.

Последовательность событий выглядит так.

1. Клиент, требующий входа в систему, передает API-шлюзу свои учетные данные методом POST.
2. Объект `LoginHandler` API-шлюза направляет запрос на выдачу пароля (www.oauth.com/oauth2-servers/access-tokens/password-grant/) серверу аутентификации OAuth 2.0.
3. Сервер аутентификации проверяет учетные данные клиента и возвращает токены доступа и обновления.
4. API-шлюз возвращает токены доступа и обновления клиенту, например, в виде cookie.
5. Клиент включает токены доступа и обновления в запросы, которые делает к API-шлюзу.
6. Перехватчик аутентификации сеанса API-шлюза проверяет токен доступа и включает его в запросы, которые делает к сервисам.

Если токен доступа просрочен или истекает его срок годности, API-шлюз получает новый токен, выполняя запрос типа OAuth 2.0 Refresh Grant (www.oauth.com/oauth2-servers/access-tokens/refreshing-access-tokens/) к серверу авторизации и указывая токен обновления. Если токен обновления не был просрочен или отозван, сервер авторизации возвращает новый токен доступа, а API-шлюз передает его сервисам и возвращает клиенту.

Важное преимущество OAuth 2.0 состоит в том, что это устоявшийся стандарт безопасности. Используя готовый сервер аутентификации OAuth 2.0, вы можете не тратить время на изобретение велосипеда, чреватое уязвимостями в архитектуре. Однако OAuth 2.0 — это не единственный способ обеспечения безопасности в микросервисных приложениях. Вне зависимости от того, какой подход вы выберете, следует помнить о трех ключевых принципах.

- ❑ API-шлюз ответственен за аутентификацию клиентов.
- ❑ API-шлюз и сервисы задействуют прозрачные токены, такие как JWT, для обмена информацией о субъекте безопасности.
- ❑ Сервис использует токен для получения учетных данных и ролей субъекта.

Вы узнали, как обезопасить свои сервисы. Теперь посмотрим, как сделать их конфигурируемыми.

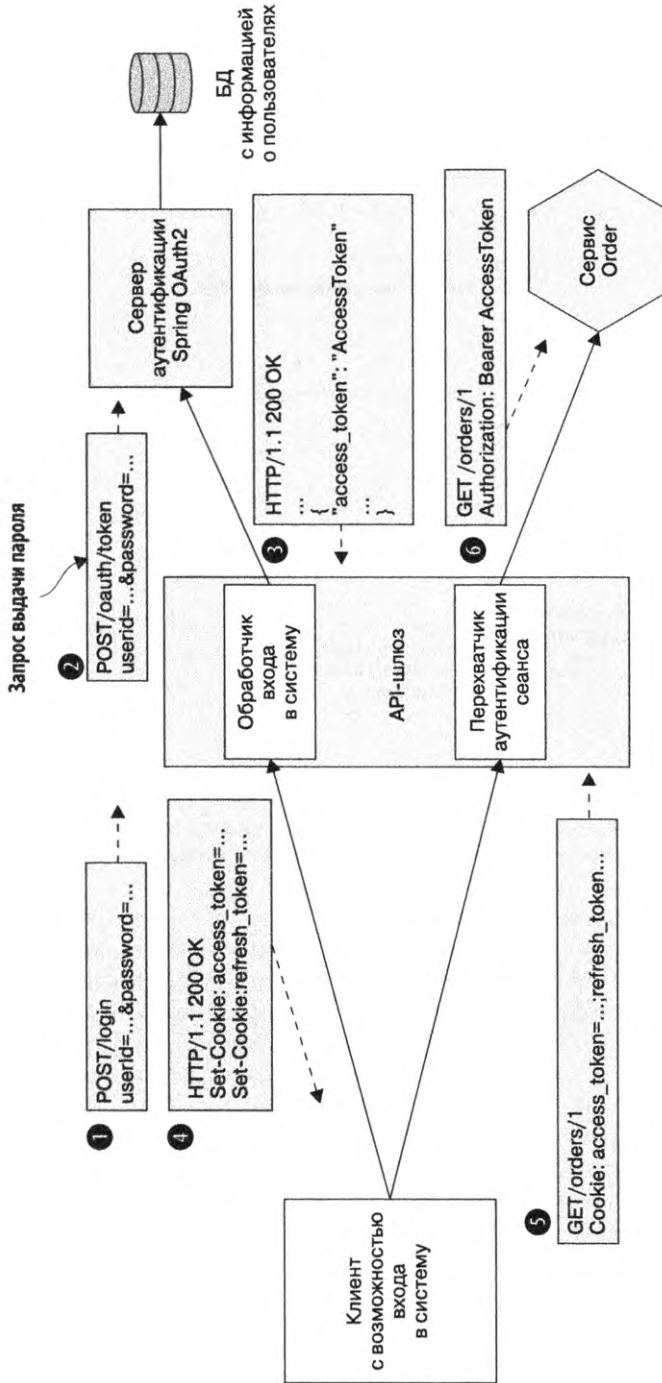


Рис. 11.5. Клиент входит в систему, отправляя API-шлюзу запрос типа POST со своими учетными данными. API-шлюз аутентифицирует учетные данные, используя сервер аутентификации OAuth 2.0, и возвращает токены доступа и обновления в виде cookie. Клиент включает эти токены в запросы, которые делает к API-шлюзу

11.2. Проектирование конфигурируемых сервисов

Представьте, что вы отвечаете за сервис `Order History`. Он занимается перехватом событий из Apache Kafka и чтением/записью элементов таблицы AWS DynamoDB (рис. 11.6). Для работы этому сервису нужны различные конфигурационные свойства, такие как сетевое расположение Apache Kafka, а также адрес и учетные данные AWS DynamoDB.

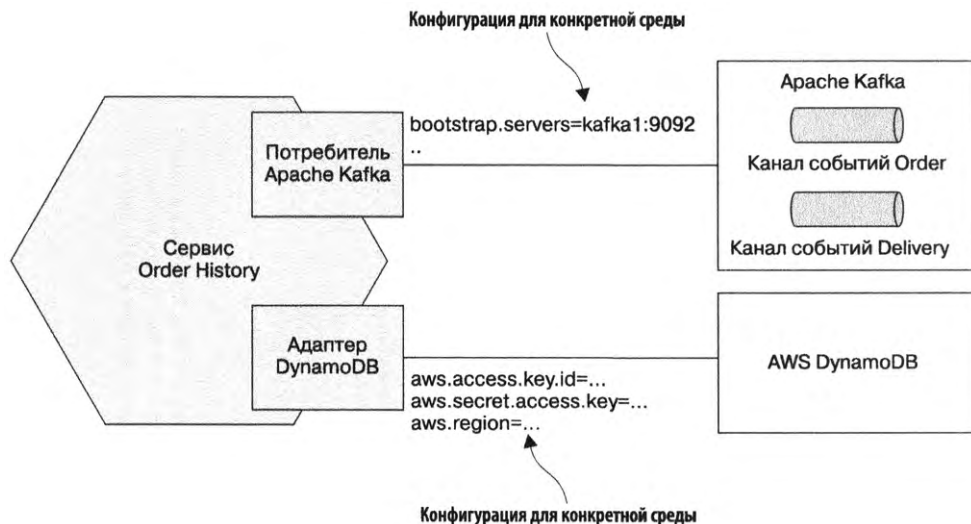


Рис. 11.6. Сервис `Order History` использует Apache Kafka и AWS DynamoDB. Ему нужно предоставить конфигурацию с местоположением каждого компонента, учетными данными и т. д.

Значения этих конфигурационных свойств зависят от того, в какой среде выполняется сервис. Например, промышленная и отладочная среды используют разные брокеры Apache Kafka и разные учетные данные для AWS. Нет никакого смысла сохранять значения конфигурационных свойств прямо в коде развертываемого сервиса, поскольку в этом случае его пришлось бы повторно собирать для каждой отдельной среды. Вместо этого сервис следует собирать один раз и затем развертывать в разных средах.

Не стоит также сохранять в исходном коде наборы конфигурационных свойств и в дальнейшем задействовать механизм профилей из фреймворка Spring для выбора подходящего набора во время выполнения. Это создает брешь в системе безопасности и ограничивает выбор сред для развертывания. Кроме того, конфиденциальная информация, такая как учетные данные, должна безопасно храниться с использованием механизма секретных ключей, такого как Hashicorp Vault (www.vaultproject.io) или AWS Parameter Store (<https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-paramstore.html>). Вы должны предоставить соответствующие конфигурационные свойства на этапе запуска сервиса с помощью конфигурации, вынесенной вовне.

Шаблон «Конфигурация, вынесенная вовне»

Предоставляет запущенному сервису значения конфигурационных свойств, такие как учетные данные для доступа к БД и сетевое размещение. См. microservices.io/patterns/externalized-configuration.html.

Механизм вынесения конфигурации вовне предоставляет экземпляру сервиса значения свойств во время его выполнения. Есть два основных подхода.

- ❑ *Пассивная модель.* Инфраструктура развертывания передает экземпляру сервиса конфигурационные свойства, используя, к примеру, переменные системного окружения или конфигурационный файл.
- ❑ *Активная модель.* Экземпляр сервиса сам берет конфигурационные свойства с сервера конфигурации.

Рассмотрим обе эти модели, начиная с пассивной.

11.2.1. Вынесение конфигурации вовне с помощью пассивной модели

Пассивная модель полагается на совместную работу среды развертывания и сервиса. Среда развертывания предоставляет конфигурационные свойства при создании экземпляра сервиса. Она может передать их в виде переменных окружения (рис. 11.7) или поместить в конфигурационный файл. Затем экземпляр сервиса прочитает эти свойства во время запуска.

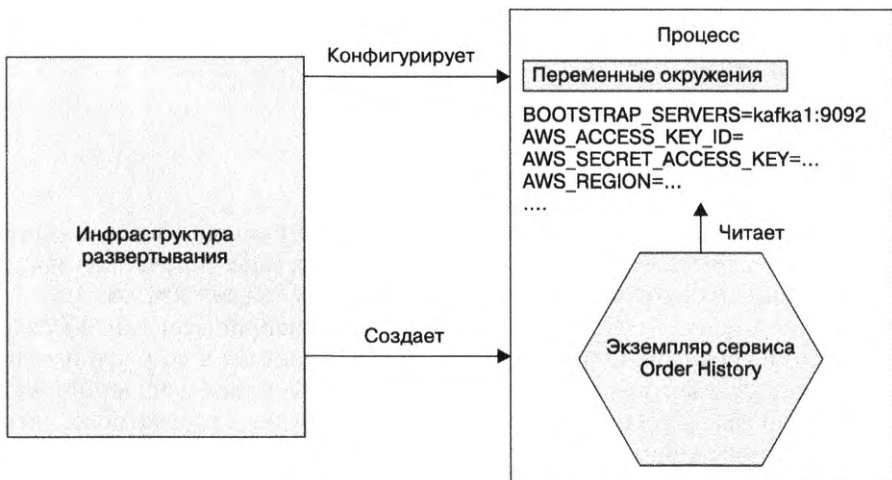


Рис. 11.7. Когда инфраструктура развертывания создает экземпляр сервиса Order History, она устанавливает переменные окружения с конфигурацией, вынесенной вовне. Сервис Order History считывает эти переменные окружения

Среда развертывания и сервис должны согласовать способ предоставления конфигурационных свойств. Выбор конкретного механизма зависит от среды развертывания. Например, в главе 12 описывается, как переменные окружения можно указать для контейнера Docker.

Допустим, вы решили предоставлять значения внешних конфигурационных свойств в виде переменных окружения. Для получения этих значений можно использовать вызов `System.getenv()`. Но если вы Java-разработчик, у вас, скорее всего, уже есть фреймворк с более удобным механизмом. Сервисы FTGO написаны с помощью фреймворка Spring Boot, обладающего чрезвычайно гибким механизмом вынесения конфигурации вовне. Он позволяет извлекать конфигурационные свойства из целого ряда источников с четкими приоритетами (<https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html>). Посмотрим, как это работает.

Spring Boot считывает свойства из целого ряда источников. Далее приведены те из них, которые, с моей точки зрения, хорошо подходят для микросервисной архитектуры.

1. Аргументы командной строки.
2. Переменная системного окружения `SPRING_APPLICATION_JSON` или системное свойство JVM с JSON внутри.
3. Системные свойства JVM.
4. Переменные системного окружения.
5. Конфигурационный файл в текущем каталоге.

Свойства, которые в этом списке находятся выше, переопределяют свойства, следующие за ними. Например, переменные системного окружения переопределяют свойства, прочитанные из конфигурационного файла.

Spring Boot делает эти свойства доступными для контекста `ApplicationContext` в Spring Framework. Чтобы получить значение свойства, сервис может воспользоваться аннотацией `@Value`:

```
public class OrderHistoryDynamoDBConfiguration {  
  
    @Value("${aws.region}")  
    private String awsRegion;
```

Spring Framework инициализирует поле `awsRegion` с помощью значения свойства `aws.region`. Это считывается из одного из приведенных ранее источников, например из конфигурационного файла или переменной окружения `AWS_REGION`.

Пассивная модель — эффективный и широко распространенный механизм конфигурации сервисов. Одно из ее ограничений состоит в том, что изменение конфигурации уже запущенного сервиса способно оказаться непростой или даже невыполнимой задачей. Инфраструктура развертывания может не позволить вам изменить внешние свойства сервиса без его перезапуска. Например, нельзя изменить переменные окружения запущенного процесса. Еще одно ограничение связано с тем, что значения конфигурационных свойств могут быть разбросаны по определениям многочисленных сервисов. В связи с этим стоит подумать об использовании активной модели. Посмотрим, как она работает.

11.2.2. Вынесение конфигурации вовне с помощью активной модели

В активной модели экземпляр сервиса считывает конфигурационные свойства с конфигурационного сервера. На рис. 11.8 показано, как это выглядит. При запуске экземпляр сервиса обращается к конфигурационному сервису за своей конфигурацией. Конфигурационные свойства для доступа к сервису конфигурации (например, его сетевое размещение) предоставляются через пассивный механизм, такой как переменные окружения.

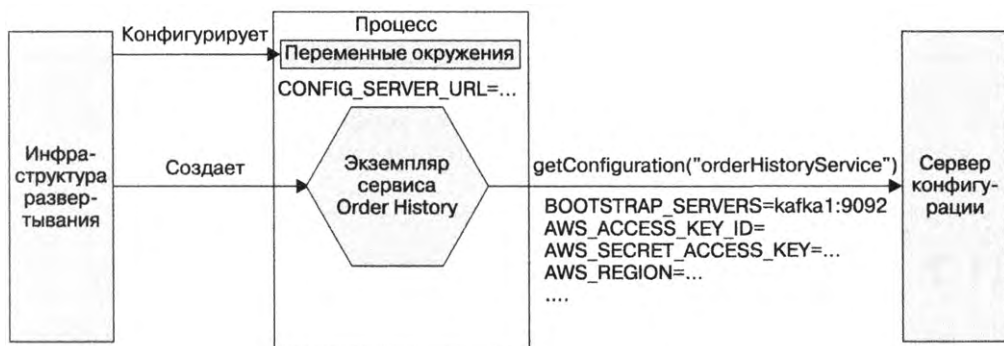


Рис. 11.8. При запуске экземпляр сервиса извлекает свои конфигурационные свойства из сервера конфигурации. Конфигурационные свойства для доступа к серверу конфигурации предоставляются инфраструктурой развертывания

Существует целый ряд способов реализации конфигурационного сервера, включая следующие:

- ❑ систему управления версиями, такую как Git;
- ❑ базы данных (SQL или NoSQL);
- ❑ специализированные серверы конфигурации, такие как Spring Cloud Config Server, Hashicorp Vault (для хранения конфиденциальной информации наподобие учетных данных) или AWS Parameter Store.

Хорошим примером фреймворка для работы с конфигурацией на основе сервера является Spring Cloud Config. Он состоит из сервера и клиента. Сервер поддерживает разнообразные хранилища для размещения конфигурационных свойств, включая системы управления версиями, базы данных и Hashicorp Vault. Клиент извлекает конфигурационные свойства из сервера и внедряет их в контекст `ApplicationContext` Spring-приложения.

Использование сервера конфигурации дает несколько преимуществ.

- ❑ **Централизованная конфигурация.** Все конфигурационные свойства хранятся в одном месте, благодаря чему ими легче управлять. Кроме того, чтобы не допустить дублирования свойств, некоторые реализации позволяют определять значения по умолчанию, которые переопределяются на уровне отдельных сервисов.

- ❑ *Прозрачная расшифровка конфиденциальных данных.* Информацию особого рода, такую как учетные данные для доступа к БД, рекомендуется шифровать. Но при этом может возникнуть проблема: экземпляру сервиса обычно приходится ее расшифровывать. Это означает, что ему нужны ключи шифрования. Некоторые серверы конфигурации автоматически расшифровывают свойства перед тем, как вернуть их сервису.
- ❑ *Динамическое изменение конфигурации.* Сервис потенциально может отслеживать обновления своих свойств, например периодически проверяя их, и изменять свою конфигурацию.

Основной недостаток использования сервера конфигурации — то, что это еще один инфраструктурный компонент, который нужно настраивать и обслуживать (разве что он предоставляется самой инфраструктурой). К счастью, открытые фреймворки, такие как Spring Cloud Config, упрощают работу с конфигурационным сервером.

Итак, вы узнали, как проектировать конфигурируемые сервисы. Теперь поговорим о том, как сделать их наблюдаемыми.

11.3. Проектирование наблюдаемых сервисов

Представьте, что вы развернули приложение FTGO в промышленной среде. Вам, скорее всего, будет интересно узнать такие его показатели, как количество запросов в секунду, степень использования ресурсов и т. д. Если возникнет проблема, например отказ экземпляра сервиса или переполнение диска, вы должны о ней узнать, и желательно до того, как она затронет пользователей. К тому же у вас должна быть возможность диагностировать эту проблему и определить ее первопричину.

Многие аспекты управления приложением в промышленных условиях, такие как мониторинг доступности и возможности применения аппаратных ресурсов, находятся вне зоны ответственности разработчиков. Эта обязанность явно принадлежит системным администраторам. Но вы, как разработчик, обязаны использовать определенные шаблоны проектирования, чтобы сделать свои сервисы более простыми для управления и диагностики. Эти шаблоны (рис. 11.9) предоставляют информацию о поведении и работоспособности сервиса. Они позволяют системе мониторинга отслеживать и визуализировать его состояние и генерировать оповещения при возникновении проблемы. К тому же шаблоны упрощают процесс диагностики.

Проектировать наблюдаемые сервисы вы можете с помощью следующих шаблонов.

- ❑ *API проверки работоспособности* — предоставляет конечную точку, которая возвращает данные о работоспособности сервиса.
- ❑ *Агрегация журналов* — ведет журналы активности сервисов и сохраняет их на центральном журнальном сервере с поддержкой поиска и оповещений.
- ❑ *Распределенная трассировка* — назначает каждому внешнему запросу уникальный идентификатор и отслеживает запросы по мере их перемещения между сервисами.

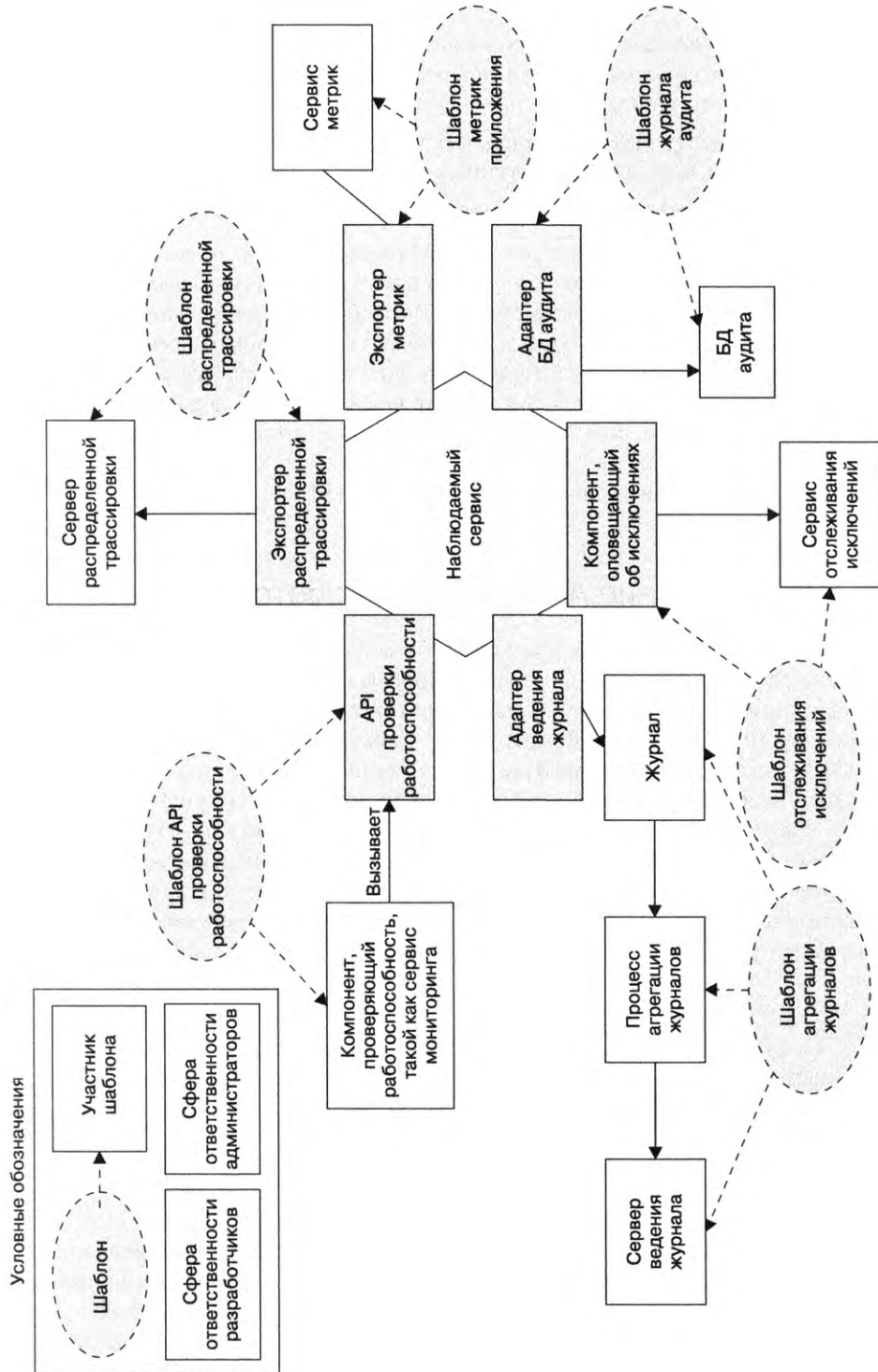


Рис. 11.9. Шаблоны наблюдаемости позволяют разработчикам и администраторам понять поведение приложения и диагностировать проблемы. Разработчики обязаны сделать свои сервисы наблюдаемыми. Администраторы отвечают за инфраструктуру для сбора информации, которую предоставляют сервисы

- ❑ *Отслеживание исключений* — за исключениями следит отдельный сервис, который избавляется от дубликатов, оповещает разработчиков и отслеживает обработку каждого исключения.
- ❑ *Метрики приложения* — сервисы собирают метрики, такие как счетчики и оценочные показатели, и делают их доступными серверу метрик.
- ❑ *Ведение журнала аудита* — ведет журнал действий пользователей.

Отличительной чертой большинства этих шаблонов является то, что они состоят из двух компонентов: для разработчиков и для администраторов. Возьмем, к примеру, шаблон API проверки работоспособности. Разработчик ответственен за реализацию конечной точки с данными о работоспособности. Системный администратор отвечает за систему мониторинга, которая периодически обращается к этому API. Аналогично работает шаблон агрегации журналов: разработчик делает так, чтобы его сервисы вели журналы с полезной информацией, а администратор занимается агрегированием этих журналов.

Рассмотрим каждый из этих шаблонов, начиная с API проверки работоспособности.

11.3.1. Использование API проверки работоспособности

Иногда запущенный сервис не в состоянии обрабатывать запросы. Например, экземпляр сервиса может быть не готов принимать запросы сразу после запуска. Скажем, для инициализации адаптеров обмена сообщениями и базы данных сервису `Consumer` требуется около 10 с. Поэтому инфраструктуре развертывания не следует направлять ему HTTP-запросы, пока он не будет в состоянии их обрабатывать.

Кроме того, экземпляр сервиса может дать сбой, не завершая при этом работу. Например, из-за ошибки сервис `Consumer` может исчерпать доступные соединения с базой данных, в результате чего он будет не способен к ней обращаться. Инфраструктура развертывания не должна направлять запросы экземплярам сервисов, которые дали сбой, но все еще работают. И если такой сервис не восстанавливается, инфраструктура должна его удалить и создать вместо него новый экземпляр.

Шаблон «API проверки работоспособности»

Сервис открывает доступ к конечной точке наподобие `GET /health`, которая возвращает данные о работоспособности сервиса. См. microservices.io/patterns/observability/health-check-api.html.

Экземпляр сервиса должен иметь возможность сообщить инфраструктуре развертывания о том, способен ли он обрабатывать запросы. Хорошим решением будет реализация в сервисе конечной точки для проверки работоспособности (рис. 11.10). Например, Java-библиотека `Spring Boot Actuator` предоставляет конечную точку `GET /actuator/health`, которая возвращает коды 200 или 503 в зави-

симости от состояния сервиса. Для .NET есть аналогичная библиотека HealthChecks (<https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/implement-resilient-applications/monitor-app-health>), реализующая конечную точку GET /hc. Инфраструктура развертывания периодически обращается по соответствующему адресу, чтобы понять, в каком состоянии находится экземпляр сервиса, и принять необходимые меры, если тот оказался неработоспособным.

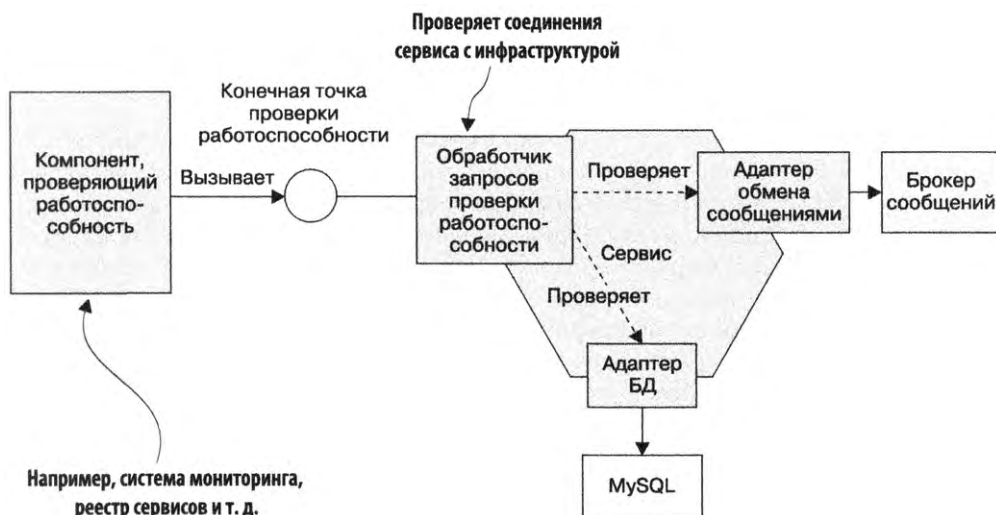


Рис. 11.10. Сервис реализует конечную точку для проверки работоспособности, к которой периодически обращается инфраструктура развертывания, чтобы определить состояние экземпляра сервиса

Обработчик запросов проверки работоспособности обычно тестирует соединения экземпляра сервиса с внешними компонентами. Он может, к примеру, выполнить тестовый запрос к базе данных. Если все тесты прошли успешно, он возвращает соответствующий ответ, такой как HTTP-код состояния 200. Если любой из них завершится неудачно, ответ будет сигнализировать о плохой работоспособности (например, в виде HTTP-кода состояния 500).

Обработчик запросов проверки работоспособности может просто вернуть пустой HTTP-ответ с подходящим кодом или подробно описать работоспособность каждого адаптера. Детальная информация может пригодиться для диагностики. Но поскольку некоторые ее аспекты могут оказаться конфиденциальными, такие фреймворки, как Spring Boot Actuator, позволяют настраивать уровень детализации ответов конечной точки.

При использовании проверок работоспособности следует обратить внимание на два момента. Во-первых, это реализация конечной точки, которая будет отчитываться о состоянии экземпляра сервиса. Во-вторых, инфраструктура развертывания должна быть сконфигурирована для обращения к этой конечной точке. Начнем с реализации.

Реализация конечной точки для проверки работоспособности

Код, реализующий конечную точку для проверки работоспособности, должен каким-то образом определять состояние экземпляра сервиса. Один из простых подходов заключается в том, чтобы проверять, может ли тот обращаться к внешним компонентам. То, как это сделать, зависит от отдельных инфраструктурных сервисов. Например, код проверки работоспособности может получить соединение к СУРБД и выполнить тестовый запрос. Более сложным было бы выполнение синтетической транзакции, которая симулирует вызов API сервиса со стороны клиента. Подобные проверки получаются более глубокими, но обычно требуют больше времени на разработку и выполнение.

Отличный пример библиотеки для проверки работоспособности — Spring Boot Actuator. Как упоминалось ранее, она предоставляет конечную точку `/actuator/health`. Код, реализующий этот вызов, возвращает результат выполнения целого набора проверок. Применяя соглашение о конфигурации, Spring Boot Actuator выполняет подходящий набор проверок с учетом того, какие инфраструктурные компоненты использует сервис. Например, если сервис работает с объектом `DataSource` из JDBC, Spring Boot Actuator конфигурирует проверку для выполнения тестового запроса. Точно так же, если сервис задействует брокер сообщений RabbitMQ, он автоматически подготавливает проверку доступности сервера RabbitMQ.

Это поведение можно изменять, выполняя дополнительные проверки работоспособности вашего сервиса. Для этого нужно определить класс, который реализует интерфейс `HealthIndicator`. У этого интерфейса есть метод `health()`, вызываемый реализацией конечной точки `/actuator/health`. Он возвращает результат проверки работоспособности.

Обращение к конечной точке проверки работоспособности

Конечная точка проверки работоспособности окажется практически бесполезной, если ее некому будет вызывать. При развертывании своего сервиса вы должны сконфигурировать инфраструктуру так, чтобы она обращалась к его конечной точке. То, как это сделать, зависит от специфики вашей инфраструктуры развертывания. Например, как говорилось в главе 3, вы можете подготовить реестр сервисов наподобие Netflix Eureka, чтобы тот обращался к конечным точкам для проверки работоспособности и решал, стоит ли направлять трафик к экземпляру сервиса. В главе 12 вы узнаете, как сконфигурировать Docker и Kubernetes для вызова таких конечных точек.

11.3.2. Применение шаблона агрегации журналов

Журналы незаменимы при диагностике. Если вы хотите узнать, что не так с вашим приложением, лучше всего начать с журнальных файлов. Однако ведение журналов в микросервисной архитектуре сопряжено с определенными трудностями. Представьте, например, что вы отлаживаете проблемный запрос `getOrderDetails()`. Как говорилось в главе 8, приложение FTGO реализует его, объединяя API-интерфейсы. В итоге нужные вам журнальные записи оказываются разбросанными между API-шлюзом и несколькими сервисами, включая `Order` и `Kitchen`.

Шаблон «Агрегация журналов»

Агрегирует журналы всех сервисов в центральной базе данных с поддержкой поиска и уведомлений. См. microservices.io/patterns/observability/application-logging.html.

Решение состоит в использовании агрегации журналов. Этот процесс отправляет журналы всех сервисов на центральный журнальный сервер (рис. 11.11). Как только сервер их сохранит, вы сможете их просматривать, искать и анализировать, а также сконфигурировать оповещения, которые срабатывают при появлении в журналах определенного сообщения.

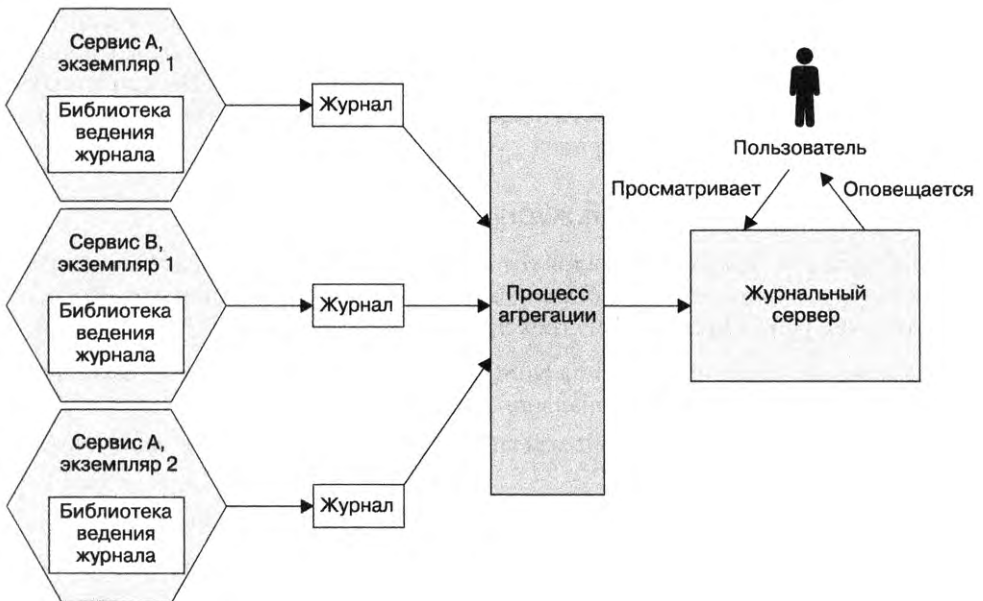


Рис. 11.11. Инфраструктура агрегации журналов шлет журналы со всех экземпляров всех сервисов на центральный журнальный сервер. Пользователи могут просматривать журналы и искать по ним. Они также могут настраивать оповещения, которые срабатывают, когда журнальная запись соответствует поисковым критериям

За процесс агрегации и журнальный сервис обычно отвечают системные администраторы. А вот написание сервисов, генерирующих осмысленные журналы, ложится на разработчиков. Сначала посмотрим, как сервис генерирует журнал.

Как сервис генерирует журнал

Вы, как разработчик сервиса, должны позаботиться о нескольких важных моментах. Во-первых, вам нужно выбрать библиотеку ведения журнала. Во-вторых, вы должны решить, куда вносить журнальные записи. Начнем с библиотеки ведения журнала.

В большинстве языков программирования есть одна или несколько библиотек для ведения журналов, которые облегчают генерацию правильно структурированных записей. Например, в Java для этого есть три популярных решения: Logback, log4j и JUL (`java.util.logging`). Можно также вспомнить SLF4J — API для различных журнальных фреймворков. В NodeJS доступен аналогичный фреймворк Log4JS. Один из разумных способов ведения журнала состоит в интеграции вызовов к этим библиотекам в код ваших сервисов. Но если существуют строгие требования к ведению журнала, которые нельзя удовлетворить с помощью сторонних библиотек, вам, возможно, придется написать собственный API, который будет служить оберткой для одной из них.

Также нужно определиться с тем, куда записывать журнал. Традиционно для этого используется журнальный файл, размещенный в известном всем каталоге. Но в случае работы с современными технологиями развертывания, такими как контейнеры и бессерверные платформы (см. главу 12), это будет не лучшим решением. В некоторых средах, таких как AWS Lambda, нет даже такого понятия, как постоянная файловая система, где можно хранить журнальные записи! Вместо этого ваш сервис должен записывать журнал в `stdout`, а инфраструктура развертывания будет решать, что делать с выводом сервиса.

Инфраструктура агрегации журналов

Инфраструктура ведения журналов отвечает за их агрегацию, хранение и предоставление пользователям функции поиска. Популярным решением в этой области является стек ELK. Он состоит из трех продуктов с открытым исходным кодом:

- ❑ *Elasticsearch* — база данных типа NoSQL, ориентированная на текстовый поиск. Используется в качестве журнального сервера;
- ❑ *Logstash* — конвейер, который агрегирует журналы сервисов и записывает их в Elasticsearch;
- ❑ *Kibana* — инструмент визуализации для Elasticsearch.

В качестве примеров других открытых решений для работы с журналами можно привести Fluentd и Apache Flume. В число журнальных сервисов можно включить как облачные сервисы, такие как AWS CloudWatch Logs, так и многочисленные коммерческие продукты. Агрегация журналов служит полезным инструментом отладки в микросервисной архитектуре.

Теперь поговорим о распределенной трассировке — еще одном подходе к пониманию поведения приложений, основанных на микросервисах.

11.3.3. Использование шаблона распределенной трассировки

Представьте, что вы разработчик системы FTGO и пытаетесь понять причину замедления запроса `getOrderDetails()`. Вы уже определили, что это не внешние сетевые проблемы. Виновником возросшей латентности должен быть либо API-шлюз, либо один из сервисов, к которому он обращается. Один из вариантов заключается в из-

мерении среднего времени ответа каждого сервиса. Но проблема в том, что это не позволяет исследовать выполнение отдельно взятых запросов. К тому же в сложных сценариях вам, вероятно, придется иметь дело с множеством вложенных обращений к сервисам, часть из которых могут оказаться незнакомыми. Это затрудняет диагностику подобного рода проблем с производительностью и их устранение в микросервисной архитектуре.

Шаблон «Распределенная трассировка»

Назначает каждому внешнему запросу уникальный идентификатор и отправляет данные о его перемещениях по системе от одного сервиса к другому на центральный сервер, который предоставляет визуализацию и возможность анализа. См. microservices.io/patterns/observability/distributed-tracing.html.

Распределенная трассировка — это хороший способ разобраться в том, чем занимается ваше приложение. Она является аналогом профайлеров производительности в монолитных системах. Она записывает информацию (например, начальное и конечное время), относящуюся к иерархии обращений к сервисам, выполняемых во время обработки запроса. Это позволяет понять, как сервисы взаимодействуют между собой при поступлении внешних запросов и на что именно затрачивается время.

Пример того, как сервер распределенной трассировки визуализирует обработку запроса API-шлюзом, показан на рис. 11.12. Вы видите входящий запрос к шлюзу, а также запрос, который шлюз делает к сервису **Order**. В обоих случаях сервер распределенной трассировки показывает выполняемую операцию и временные рамки запроса.

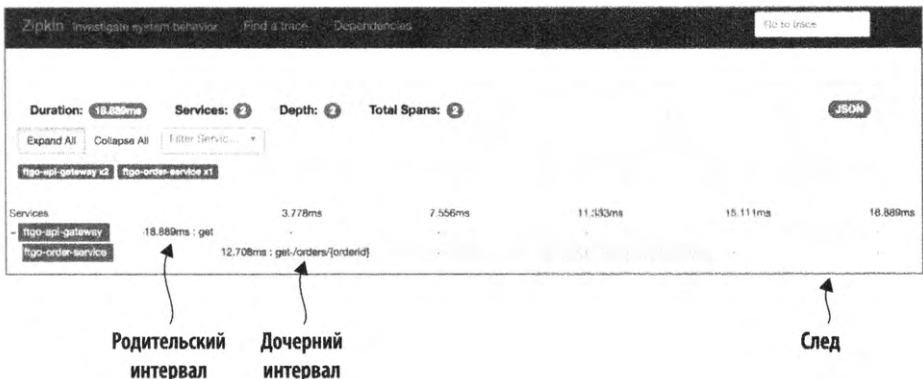


Рис. 11.12. Сервер Zipkin показывает, как приложение FTGO обрабатывает запрос, который API-сервер направляет к сервису Order. Каждый запрос представлен в виде следа. След — это набор интервалов. Каждый интервал описывает вызов сервиса и может содержать дочерние интервалы. В зависимости от детализации собранных данных интервал может также относиться к вызову операции внутри сервиса

На рис. 11.12 показано то, что в терминологии распределенной трассировки называется *следом*. След описывает внешний запрос и состоит из одного или нескольких интервалов. *Интервал* представляет собой операцию с такими ключевыми атрибутами, как название, начальное и конечное время. Интервал может содержать дочерние интервалы, которые представляют вложенные операции. Например, интервал верхнего уровня может описывать обращение к API-шлюзу (см. рис. 11.12). Его дочерние интервалы относятся к вызовам, которые API-шлюз направляет сервисам.

Полезным побочным эффектом распределенной трассировки является назначение уникального идентификатора каждому внешнему запросу. Сервис может включать эти идентификаторы в свои журнальные записи. В сочетании с агрегацией журналов это позволяет легко находить записи для отдельных внешних вызовов. Далее показан пример журнальной записи из сервиса Order:

```
2018-03-04 17:38:12.032 DEBUG [ftgo-orderservice,
      8d8fdc37be104cc6,8d8fdc37be104cc6,false]
  7 --- [nio-8080-exec-6] org.hibernate.SQL :
    select order0_.id as id1_3_0_, order0_.consumer_id as consumer2_3_0_, order
      0_.city as city3_3_0_,
    order0_.delivery_state as delivery4_3_0_, order0_.street1 as street5_3_0_,
    order0_.street2 as street6_3_0_, order0_.zip as zip7_3_0_,
    order0_.delivery_time as delivery8_3_0_, order0_.a
```

Фрагмент журнальной записи [ftgo-order-service,8d8fdc37be104cc6,8d8fdc37be104cc6,false] (сопоставляемый контекст диагностики в SLF4J — см. www.slf4j.org/manual.html) содержит информацию, предоставленную инфраструктурой распределенной трассировки. Он состоит из четырех значений:

- ❑ ftgo-order-service — название приложения;
- ❑ 8d8fdc37be104cc6 — поле traceId;
- ❑ 8d8fdc37be104cc6 — поле spanId;
- ❑ false — говорит о том, что этот интервал не был экспортирован в сервер распределенной трассировки.

Если поискать в журналах 8d8fdc37be104cc6, можно найти все записи, относящиеся к этому запросу.

На рис. 11.13 показано, как работает распределенная трассировка. Она состоит из двух частей: библиотеки инструментирования, которую использует каждый сервис, и сервера распределенной трассировки. Библиотека инструментирования управляет следами и интервалами. Она также включает в исходящие запросы трассировочную информацию, такую как идентификаторы текущего и родительского следов. Например, один из распространенных стандартов для передачи трассировочной информации, B3 (github.com/openzipkin/b3-propagation), применяет заголовки наподобие X-B3-TraceId и X-B3-ParentSpanId. Кроме того, библиотека инструментирования передает следы серверу распределенной трассировки, который хранит их и предоставляет пользовательский интерфейс для их визуализации.

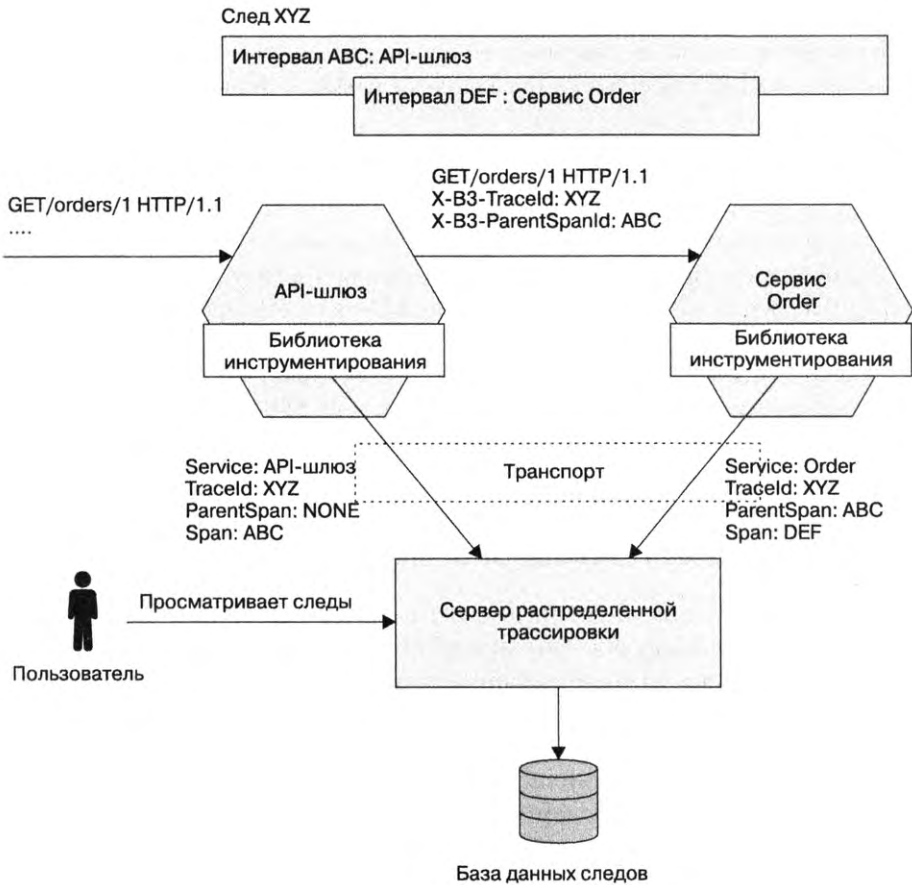


Рис. 11.13. Библиотека инструментирования применяется во всех сервисах, включая API-шлюз. Она назначает ID для каждого внешнего запроса, распространяет состояние трассировки между сервисами и передает следы серверу распределенной трассировки

Рассмотрим по очереди библиотеку инструментирования и сервер распределенной трассировки.

Использование библиотеки инструментирования

Библиотека инструментирования создает иерархию следов и передает ее серверу распределенной трассировки. Она может вызываться напрямую в коде сервиса, но это будет вмешательством в бизнес-логику и другую функциональность. Более элегантный подход — применение перехватчиков или аспектно-ориентированного программирования (АОП).

Отличным примером фреймворка, основанного на АОП, служит Spring Cloud Sleuth. С помощью механизма АОП из состава Spring он автоматически интегрирует

распределенную трассировку в сервисы. Поэтому вам следует добавить Spring Cloud Sleuth в качестве одной из зависимостей проекта. Вашему сервису нужно вызывать API распределенной трассировки только в тех случаях, которые не охвачены данным фреймворком.

О сервере распределенной трассировки

Библиотека инструментирования шлет следы на сервер распределенной трассировки. Тот собирает их в единую иерархию и сохраняет в базу данных. Одним из популярных серверов распределенной трассировки является Open Zipkin, изначально разработанный компанией Twitter. Для доставки следов в Zipkin сервисы могут использовать либо HTTP, либо брокер сообщений. Zipkin помещает следы в хранилище, роль которого может играть база данных типа SQL или NoSQL. У него есть пользовательский интерфейс, который вы видели ранее на рис. 11.12. Еще одним сервером распределенной трассировки можно считать AWS X-ray.

11.3.4. Применение шаблона «Показатели приложения»

Ключевую роль в промышленной среде играют мониторинг и оповещения. Система мониторинга собирает показатели всех компонентов стека технологий, чтобы получить критически важную информацию о работоспособности приложения (рис. 11.14). Показатели могут быть инфраструктурными (например, нагрузка на процессор, память и диск) и программными (например, латентность обращений к сервисам и количество выполненных запросов). Так, сервис Order собирает информацию о количестве размещенных, принятых и отклоненных заказов. Показатели агрегирует отдельный сервис, который предоставляет визуализацию и оповещения.

Шаблон «Показатели приложения»

Сервис шлет отчеты с показателями центральному серверу, который предоставляет агрегацию, визуализацию и оповещения.

Показатели снимаются периодически. Каждый образец содержит три свойства:

- ❑ *название* — название показателя, такое как `jvm_memory_max_bytes` или `placed_orders`;
- ❑ *значение* — числовое значение;
- ❑ *временную метку* — время снятия показателя.

Кроме того, некоторые системы мониторинга поддерживают концепцию *измерений*, которые представляют собой произвольные пары «имя — значение».

Например, показатель `jvm_memory_max_bytes` предоставляется с такими измерениями, как `area="heap"`, `id="PS Eden Space"` и `area="heap"`, `id="PS Old Gen"`. Измерения часто несут в себе дополнительную информацию: имя компьютера или сервиса, идентификатор экземпляра сервиса и т. д. Система мониторинга обычно *агрегирует* (суммирует или вычисляет среднее значение) выборки по одному или нескольким измерениям.

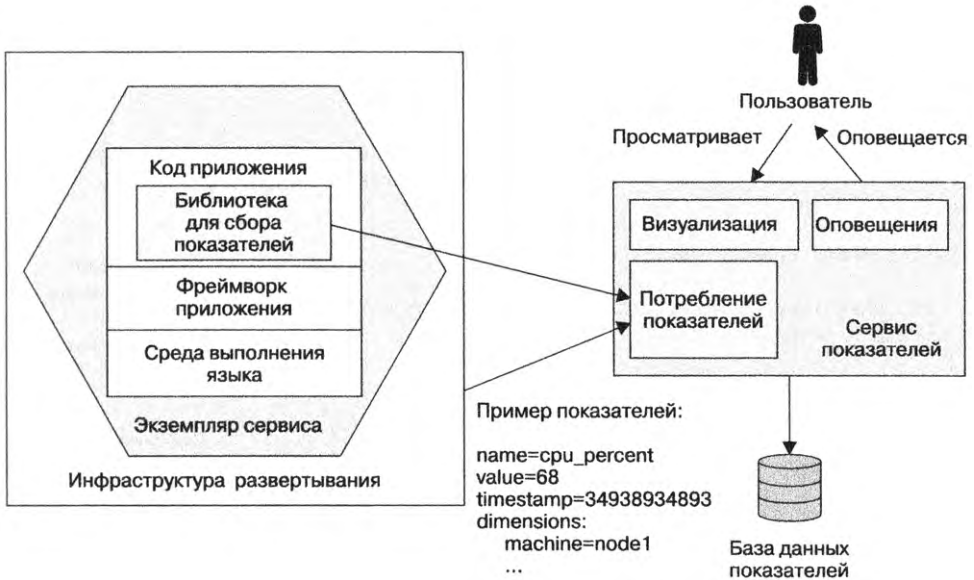


Рис. 11.14. Показатели собирают на каждом уровне стека и хранят в отдельном сервисе, который предоставляет визуализацию и оповещения

За многие аспекты мониторинга отвечают системные администраторы. Но и у разработчиков сервисов есть две обязанности, связанные с показателями. Во-первых, они должны сделать так, чтобы их сервисы собирали сведения о своем поведении. Во-вторых, им нужно открыть эти сведения вместе с информацией из JVM и фреймворка приложения для сервера показателей.

Посмотрим, как собирают показатели уровня сервиса.

Сбор показателей уровня сервиса

Количество усилий, которые нужно приложить для сбора показателей, зависит от фреймворка вашего приложения и того, какая информация вас интересует. Например, сервис, основанный на Spring Boot, может собирать и делать доступными извне базовые показатели, относящиеся к JVM. Для этого нужно указать в списке зависимостей библиотеку Micrometer Metrics и написать несколько строчек в конфигурационном файле. Механизм автоконфигурации Spring Boot берет на себя настройку

библиотеки Micrometer Metrics и открытие доступа к собранным показателям. Использование API библиотеки Micrometer Metrics напрямую имеет смысл только в случае, если сервис собирает сведения о приложении.

В листинге 11.1 показано, как сервис Order собирает показатели количества размещенных, принятых и отклоненных заказов. Для сбора нестандартных показателей он задействует интерфейс MeterRegistry из состава Micrometer Metrics. Каждый метод инкрементирует счетчик с соответствующим именем.

Листинг 11.1. Сервис Order отслеживает количество размещенных, принятых и отклоненных заказов

```
public class OrderService {

    @Autowired
    private MeterRegistry meterRegistry;

    public Order createOrder(...) {
        ...
        meterRegistry.counter("placed_orders").increment();
        return order;
    }

    public void approveOrder(long orderId) {
        ...
        meterRegistry.counter("approved_orders").increment();
    }

    public void rejectOrder(long orderId) {
        ...
        meterRegistry.counter("rejected_orders").increment();
    }
}
```

API библиотеки Micrometer Metrics для работы с показателями уровня приложения

Инкрементирует счетчик placedOrders при успешном размещении заказа

Инкрементирует счетчик approvedOrders, когда заказ принимается

Инкрементирует счетчик rejectedOrders, когда заказ отклоняется

Доставка собранных данных на сервер показателей

Собранные сведения могут попасть от сервиса к серверу показателей одним из двух способов — пассивным или активным. В *пассивной* модели экземпляр сервиса сам шлет показатели серверу, вызывая его API. Эта модель реализована, к примеру, в AWS Cloudwatch.

В *активной* модели сервер показателей или его агент, запущенный локально, обращается к API сервиса, чтобы извлечь собранную им информацию. Этот подход применяется в Prometheus — популярной системе для мониторинга и рассылки оповещений с открытым исходным кодом.

Для интеграции с Prometheus сервис Order приложения FTGO использует библиотеку micrometer-registry-prometheus. Поскольку эта библиотека указана в списке путей classpath, Spring Boot предоставляет конечную точку GET /actuator/prometheus, которая возвращает показатели в формате, совместимом с Prometheus. Отчет о пользовательских показателях, собранных сервисом Order, выглядит так:

```
$ curl -v http://localhost:8080/actuator/prometheus | grep _orders
# HELP placed_orders_total
# TYPE placed_orders_total counter
placed_orders_total{service="ftgo-order-service",} 1.0
# HELP approved_orders_total
# TYPE approved_orders_total counter
approved_orders_total{service="ftgo-order-service",} 1.0
```

Например, счетчик `placed_orders` передается в качестве показателя типа `counter`.

Сервер Prometheus периодически обращается к этой конечной точке для извлечения показателей. Оказавшись на сервере, показатели становятся доступными для просмотра с помощью Grafana — инструмента для визуализации данных (grafana.com). Вы также можете настроить оповещения для этих показателей, например, на случай, если скорость изменения `placed_orders_total` станет ниже определенного значения.

Показатели приложения предоставляют ценную информацию о его поведении. Оповещения, которые срабатывают в зависимости от их изменения, позволяют быстро реагировать на проблемы в промышленной среде — возможно, даже до того, как они затронут пользователей. Давайте посмотрим, как отслеживать еще один вид оповещений — исключения — и реагировать на них.

11.3.5. Шаблон отслеживания исключений

Исключения редко попадают в журнал сервиса, но когда это происходит, очень важно определить их первопричину. Исключение может быть симптомом отказа или ошибки в коде. Традиционно исключения просматриваются в журнале. Вы даже можете настроить журнальный сервер таким образом, чтобы он оповещал вас о любом записанном исключении. Однако у этого подхода есть несколько проблем.

- ❑ Журнальные файлы рассчитаны на однострочные записи, тогда как исключения состоят из множества строчек.
- ❑ Нет механизма для отслеживания того, как решаются проблемы с записанными исключениями. Вам придется вручную копировать и вставлять исключение в систему отслеживания проблем.
- ❑ Исключения, скорее всего, будут дублироваться, но механизма для их автоматической группировки не существует.

Шаблон «Отслеживание исключений»

Сервис сообщает об исключениях центральному сервису, который их дедуплицирует, отслеживает их исправление и генерирует оповещения. См. microservices.io/patterns/observability/audit-logging.html.

Более подходящей альтернативой является использование сервиса для отслеживания исключений. Ваш код сообщает этому сервису об исключениях через API — например, REST (рис. 11.15). Тот их дедуплицирует, управляет процессом их исправления и генерирует оповещения.

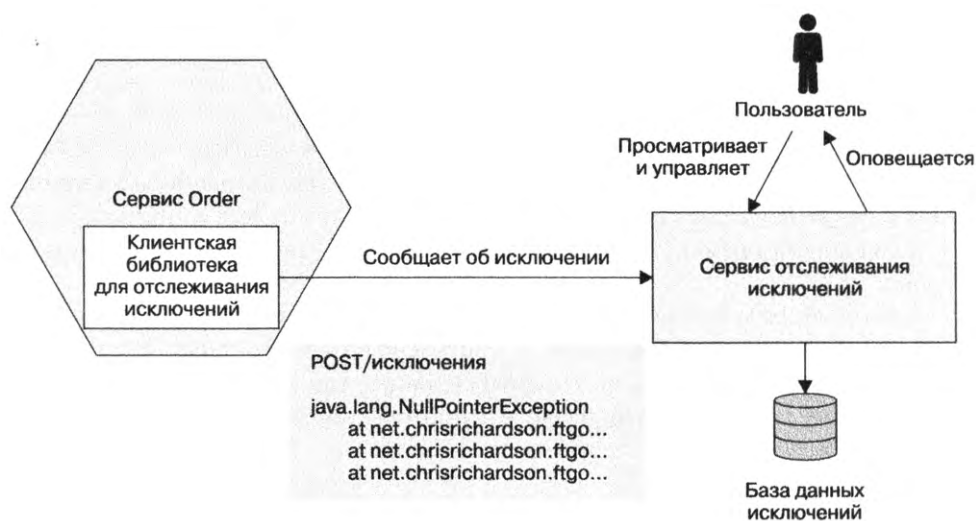


Рис. 11.15. Ваш код сообщает об исключениях специальному сервису, который их дедуплицирует и оповещает разработчиков. У него есть пользовательский интерфейс для просмотра исключений и управления ими

Сервис отслеживания исключений можно интегрировать в приложение несколькими способами. Код может вызывать его API напрямую. Но лучше использовать для этого клиентскую библиотеку, предоставляемую этим сервисом. Например, клиентская библиотека HoneyBadger поддерживает несколько простых механизмов интеграции, включая Servlet Filter, который перехватывает исключения и сообщает о них.

Сервисы отслеживания исключений

Существует несколько сервисов отслеживания исключений. Некоторые из них, такие как Honeybadger (www.honeybadger.io), — чисто облачные, другие, например Sentry.io (sentry.io/welcome/), имеют версию с открытым исходным кодом и могут быть развернуты в рамках вашей собственной инфраструктуры. Они принимают исключения, переданные вашим приложением, и генерируют оповещения. Они предоставляют консоль для просмотра исключений и позволяют управлять их исправлением. У сервисов отслеживания исключений обычно есть клиентские библиотеки для разных языков.

Шаблон отслеживания исключений помогает быстро идентифицировать проблемы в промышленной среде и реагировать на них.

Отслеживать поведение пользователей тоже очень важно. Посмотрим, как это делается.

11.3.6. Применение шаблона «Ведение журнала аудита»

Цель ведения журнала аудита — запись всех действий пользователя. Журнал аудита обычно используется для помощи службе поддержки, соблюдения нормативно-правовых норм и обнаружения подозрительной активности. Каждая запись в таком журнале содержит идентификатор пользователя, выполненное им действие и бизнес-объект (-ы). Приложения обычно хранят журналы аудита в базах данных.

Шаблон «Ведение журнала аудита»

Записывает действия пользователя в базу данных, чтобы помочь службе поддержки, обеспечить соблюдение нормативно-правовых норм и обнаружить подозрительную активность. См. microservices.io/patterns/observability/audit-logging.html.

Ведение журнала аудита можно реализовать несколькими разными способами.

- ☐ Добавить код ведения журнала аудита в бизнес-логику.
- ☐ Задействовать аспектно-ориентированное программирование.
- ☐ Использовать порождение событий.

Рассмотрим каждый из этих вариантов.

Добавление кода для ведения журнала аудита в бизнес-логику

Первый и самый простой вариант — внедрить код ведения журнала аудита в бизнес-логику вашего сервиса. Например, каждый метод сервиса может создавать новую запись и сохранять ее в базе данных. Недостаток этого подхода состоит в том, что он связывает код ведения журнала аудита и бизнес-логику, что усложняет обслуживание приложения. Еще один отрицательный аспект связан с повышенным риском возникновения ошибок, так как за написание кода для ведения журнала аудита отвечает разработчик.

Использование аспектно-ориентированного программирования

Второй вариант заключается в применении АОП. Вы можете воспользоваться АОП-фреймворком, таким как Spring AOP, для создания так называемых *советов*, которые автоматически перехватывают вызов каждого метода внутри сервиса и сохраняют запись в журнал аудита. Это куда более надежный подход, поскольку он

автоматически записывает все вызовы. Его основным недостатком является то, что совет имеет доступ только к названию метода и его аргументам, поэтому у вас могут возникнуть проблемы с определением бизнес-объекта, на который направлено действие, и генерацией бизнес-ориентированной журнальной записи.

Использование порождения событий

Третьим, последним вариантом является реализация бизнес-логики с применением порождения событий. Как упоминалось в главе 6, *порождение событий* автоматически предоставляет журнал аудита для операций создания и обновления. При этом вам нужно записывать учетные данные пользователя в каждое событие. Одно из ограничений этого подхода связано с тем, что он не записывает запросы. Если ваш сервис должен создавать журнальные записи для запросов, следует выбрать другой вариант.

11.4. Разработка сервисов с помощью шаблона микросервисного шасси

В этой главе описано множество функций, которые должен поддерживать ваш сервис, включая показатели, отправку отчетов об исключениях, ведение журнала, проверку работоспособности, работу с внешней конфигурацией и безопасность. Кроме того, как вы уже знаете из главы 3, сервис отвечает также за свое обнаружение и реализацию предохранителей. Вряд ли вам захочется подготавливать все это с нуля при создании каждого нового сервиса, ведь это может занять дни, а то и недели. И в это время вы не сможете написать ни единой строчки бизнес-логики.

Шаблон «Шасси микросервисов»

Создание сервисов на основе фреймворка или набора фреймворков, которые реализуют такие общие функции, как отслеживание исключений, ведение журнала, проверка работоспособности, работа с внешней конфигурацией и распределенная трассировка. См. microservices.io/patterns/microservice-chassis.html.

Чтобы значительно ускорить этот процесс, сервисы можно разрабатывать поверх микросервисных шасси. *Шасси микросервисов* (рис. 11.16) — это фреймворк или набор фреймворков, которые берут на себя перечисленные обязанности. Для реализации всех этих функций от вас не требуется почти (или даже совсем) никакого кода.

Я начну раздел с описания концепции микросервисного шасси и предложу несколько замечательных фреймворков, которые ее воплощают. Вы также познакомитесь с понятием «сеть сервисов», которое на момент написания книги является новой интригующей альтернативой фреймворкам и библиотекам.

Сначала обсудим шасси микросервисов.



Рис. 11.16. Шасси микросервисов — это фреймворк, который берет на себя многочисленные обязанности, такие как отслеживание исключений, ведение журнала, проверки работоспособности, работа с внешней конфигурацией и распределенная трассировка

11.4.1. Использование шасси микросервисов

Шасси микросервисов — это фреймворк или набор фреймворков, которые берут на себя многочисленные обязанности, включая следующие:

- ☐ конфигурацию, вынесенную вовне;
- ☐ проверку работоспособности;
- ☐ показатели приложения;
- ☐ обнаружение сервисов;
- ☐ предохранители;
- ☐ распределенную трассировку.

Это может значительно уменьшить объем кода, который вам необходимо написать, или даже свести его к нулю. Вам нужно лишь адаптировать микросервисное

шасси под свои требования. Это позволяет сосредоточиться на разработке бизнес-логики сервисов.

В качестве микросервисного шасси приложение FTGO использует фреймворки Spring Boot и Spring Cloud. Spring Boot предоставляет такие функции, как поддержка внешней конфигурации. Spring Cloud реализует шаблоны наподобие предохранителя и обнаружения сервисов на стороне клиента (хотя у FTGO есть своя инфраструктура для обнаружения сервисов). Помимо Spring Boot и Spring Cloud, есть и другие фреймворки микросервисных шасси. Например, если вы пишете сервисы на языке GoLang, вам доступны проекты Go Kit (github.com/go-kit/kit) и Micro (github.com/micro/micro).

У этого шаблона есть один недостаток: вам нужно будет подбирать отдельное шасси для каждой комбинации языка/платформы, которую вы используете в разработке сервисов. К счастью, многие функции микросервисного шасси, скорее всего, уже реализованы на уровне инфраструктуры. Например, как описывалось в главе 3, многие среды развертывания занимаются обнаружением сервисов. Более того, многие сетевые возможности микросервисных шасси уже реализованы в виде сети сервисов (внешнего инфраструктурного слоя).

11.4.2. От микросервисного шасси до сети сервисов

Шасси микросервисов — хороший способ реализации универсальных функций, таких как предохранители. Однако для каждого языка программирования, который вы используете, требуется отдельное шасси. Например, Spring Boot и Spring Cloud подходят для разработчиков на Java/Spring, но они мало чем помогут, если вы пишете сервис на основе NodeJS.

Шаблон «Сеть сервисов»

Пропускает весь трафик между сервисами через сетевой слой, реализующий такие функции, как предохранители, распределенная трассировка, обнаружение сервисов, балансирование нагрузки и маршрутизация трафика с учетом правил. См. microservices.io/patterns/deployment/service-mesh.html.

У этого подхода появляется новая альтернатива, которая заключается в реализации этих функций в рамках так называемой сети сервисов. *Сеть сервисов* — это сетевая инфраструктура, которая служит промежуточным звеном между вашим сервисом и другими внутренними и внешними компонентами. На рис. 11.17 показано, как весь входящий и исходящий трафик сервиса проходит через эту сеть, которая берет на себя функции предохранителей, распределенной трассировки, обнаружения сервисов, балансирования нагрузки и маршрутизации трафика с учетом правил. Сеть сервисов может также защищать межсервисное взаимодействие с помощью IPC на основе TLS. Благодаря этому больше не нужно реализовывать эти возможности внутри сервисов.

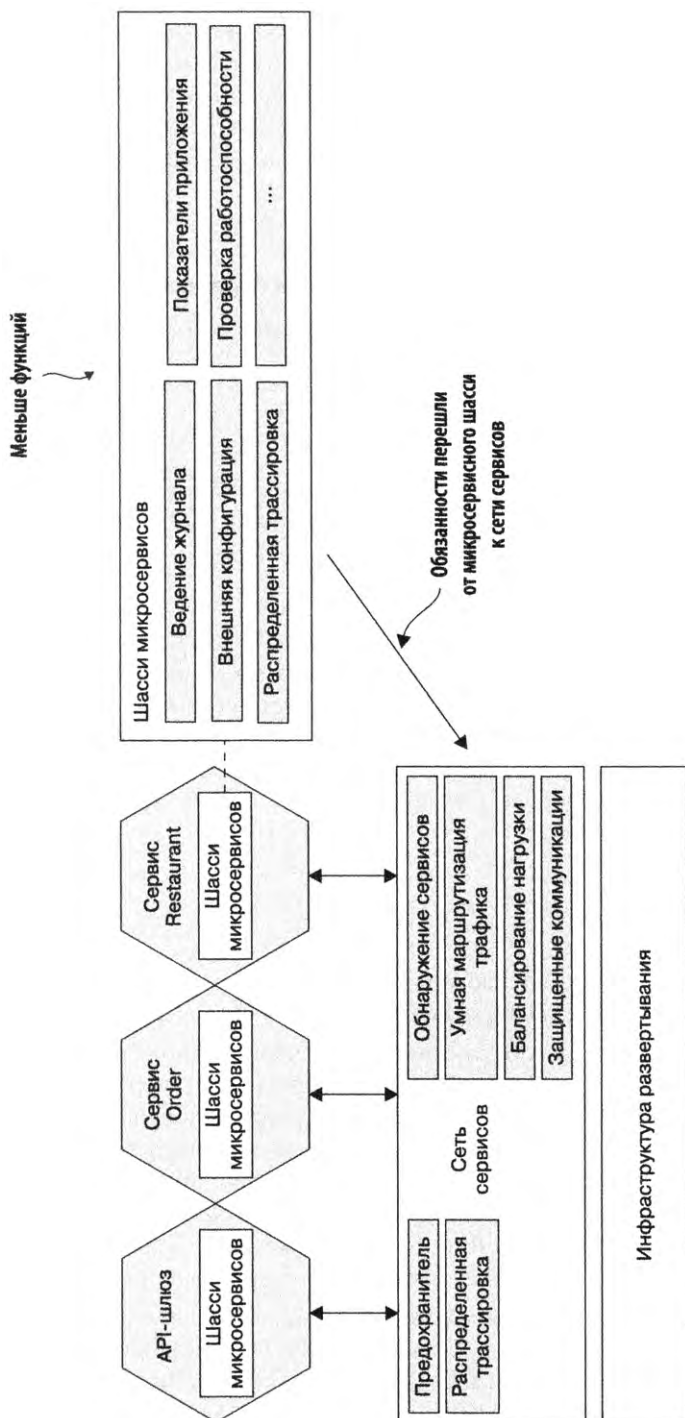


Рис. 11.17. Весь входящий и исходящий трафик сервиса проходит через сеть сервисов, которая реализует функции предохранителей, распределенной трассировки, обнаружения сервисов и балансирования нагрузки. Остальные возможности можно реализовать в микросервисном шасси. Также сеть сервисов защищает межпроцессные коммуникации с помощью TLS

Использование сети сервисов значительно упрощает микросервисное шасси, которое в этом случае отвечает только за функции, глубоко интегрированные в код приложения, например за работу с внешней конфигурацией и проверку работоспособности. Микросервисное шасси также должно поддерживать распределенную трассировку путем распространения такой информации, как заголовки стандарта V3, рассмотренные в подразделе 11.3.3.

Современные реализации сети сервисов

Сеть сервисов представлена несколькими реализациями, включая:

- Istio (istio.io);
- Linkerd (linkerd.io);
- Conduit (conduit.io).

На момент написания этих строк самой зрелой из них является Linkerd, а Istio и Conduit находятся в стадии активного развития. Больше об этих захватывающих технологиях можно прочитать в документации к продуктам.

Концепция сети сервисов очень многообещающая. Она освобождает разработчика от реализации различных общих функций. Кроме того, благодаря маршрутизации трафика вы можете разделить развертывание от выпуска обновлений. Это позволяет развертывать новые версии в промышленной среде, делая их доступными только для некоторых пользователей, например для внутренней команды тестирования. Мы поговорим об этом подробнее в главе 12, где речь пойдет о развертывании сервисов с помощью Kubernetes.

Резюме

- ❑ Важно, чтобы сервис соответствовал функциональным требованиям, но вместе с тем он должен быть безопасным, конфигурируемым и наблюдаемым.
- ❑ В том, что касается безопасности, микросервисная и монолитная архитектуры имеют много общего. Но есть и некоторые различия, например передача учетных данных между API-шлюзом и сервисами или выбор компонента, ответственного за аутентификацию и авторизацию. Аутентификацией клиентов обычно занимается API-шлюз. В каждый запрос к сервису он добавляет прозрачный токен, такой как JWT. Он содержит учетные данные субъекта и его роли. Сервис использует эту информацию для авторизации доступа к ресурсам. Хорошей основой для безопасности в микросервисной архитектуре может быть стандарт OAuth 2.0.
- ❑ Сервис обычно общается с одним или несколькими внешними компонентами, такими как брокер сообщений или база данных. Их сетевое размещение и учетные данные часто зависят от среды, в которой запущен сервис. Вы должны применить шаблон вынесения конфигурации вовне и реализовать механизм, который предо-

ставляет сервису конфигурационные свойства на этапе выполнения. Во многих случаях инфраструктура развертывания делает эти свойства доступными через переменные системного окружения или конфигурационный файл во время создания экземпляра сервиса. В качестве альтернативы экземпляр сервиса может сам извлекать свою конфигурацию из специального сервера.

- ❑ Системные администраторы и разработчики разделяют ответственность за реализацию шаблонов наблюдаемости. Администраторы отвечают за соответствующую инфраструктуру, такую как серверы для агрегации журналов, сбора показателей, отслеживания исключений и распределенной трассировки. Разработчики должны сделать так, чтобы их сервисы были наблюдаемыми. Сервис должен иметь конечные точки для проверки работоспособности, генерировать журнальные записи, собирать и «выставлять наружу» показатели, отправлять отчеты серверу для отслеживания исключений и поддерживать распределенную трассировку.
- ❑ Чтобы упростить и ускорить разработку, вы должны писать свои сервисы поверх микросервисного шасси. Микросервисное шасси — это фреймворк или набор фреймворков для реализации различных общих функций, включая те, что были описаны в этой главе. Хотя, скорее всего, со временем многие сетевые обязанности перейдут от микросервисного шасси к сети сервисов — инфраструктурной прослойке, через которую проходит весь сетевой трафик сервисов.

Развертывание микросервисов

В этой главе

- Четыре ключевых шаблона развертывания:
 - ✓ формат пакетов для отдельных языков;
 - ✓ развертывание сервиса в виде VM;
 - ✓ развертывание сервиса в виде контейнера;
 - ✓ бессерверное развертывание;
 - ✓ их принцип работы, преимущества и недостатки.
- Развертывание сервисов с помощью Kubernetes.
- Использование сети сервисов для разделения развертывания и выпуска обновлений.
- Развертывание сервисов с помощью AWS Lambda.
- Выбор шаблона развертывания.

Мэри и ее команда уже почти закончили писать свой первый сервис. И хотя ему все еще недостает некоторых функций, он успешно выполняется на ноутбуках разработчиков и сервере Jenkins CI. Но этого недостаточно. Чтобы приносить прибыль компании FTGO, программное обеспечение должно быть развернуто в промышленной среде и доступно пользователям. Разработчикам необходимо развернуть свой сервис в промышленных условиях.

Развертывание — это сочетание двух взаимосвязанных концепций — процесса и архитектуры. Процесс развертывания заключается в доставке кода в промышленную среду и состоит из этапов, которые должны выполнить люди — разработчики или системные администраторы. Архитектура развертывания определяет структуру

среды, в который этот код будет выполняться. С конца 1990-х годов, когда я начал писать промышленные Java-приложения, оба эти аспекта радикально изменились. Процесс перебрасывания разработчиками кода на рабочий сервер вручную стал высокоавтоматизированным. На смену физической промышленной среде пришла легковесная динамическая вычислительная инфраструктура (рис. 12.1).

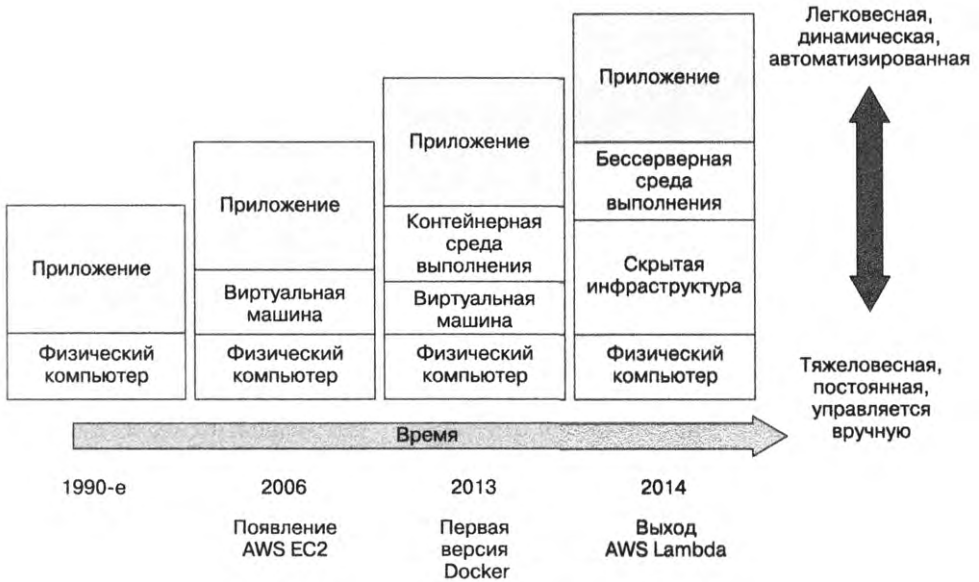


Рис. 12.1. Тяжеловесные компьютеры с продолжительным временем работы скрываются за все более легковесными динамическими технологиями

В 1990-е годы, если вы хотели развернуть приложение в промышленной среде, вам нужно было передать свой код вместе с набором инструкций по его обслуживанию системным администраторам. Вы могли, к примеру, оформить заявку с просьбой развернуть приложение. Все, что происходило после этого, ложилось на плечи администраторов, исключением были ситуации, когда обнаруживалась проблема, которую нельзя было решить без вашей помощи. Обычно системные администраторы покупали и устанавливали дорогие и тяжеловесные серверы приложений, такие как WebLogic или WebSphere. Затем заходили в консоль этих серверов и развертывали ваш код. Они с любовью заботились об этих компьютерах, словно о собственных питомцах, устанавливая заплатки и обновляя программное обеспечение.

В середине 2000-х на смену дорогим серверам приложений пришли открытые легковесные веб-контейнеры наподобие Apache Tomcat и Jetty. Теперь стало возможным выделять целый веб-контейнер лишь для одного приложения, хотя это не было обязательным требованием. Примерно в то же время вместо физического оборудования начали применять виртуальные машины. Но к серверам по-прежнему относились как к домашним животным, а развертывание, как и раньше, было принципиально ручным процессом.

В наши дни процесс развертывания выглядит совсем иначе. Вместо передачи кода отдельной команде системных администраторов используется концепция DevOps, согласно которой ответственность за развертывание приложений и сервисов частично ложится и на разработчиков. В некоторых организациях администраторы предоставляют разработчикам консоль для развертывания проектов. Но еще лучше, когда после прохождения тестов код автоматически доставляется в промышленную среду.

Радикально изменились и вычислительные ресурсы, которые используются в промышленной среде. Вместо физических серверов мы имеем дело с виртуальными машинами, запущенными в высокоавтоматизированных облаках наподобие AWS. Современные ВМ неизменяемы. Это уже не домашние любимцы, а безликие одноразовые сущности, которые легче удалить и воссоздать, чем конфигурировать заново. Все большую популярность набирают *контейнеры* — еще более легковесный слой абстракции поверх виртуальных машин. Дальнейшее развитие этой идеи — бессерверные платформы развертывания, такие как AWS Lambda, которые имеют множество сфер применения.

Эволюция процессов и архитектур развертывания не случайно происходит одновременно с ростом популярности микросервисов. Приложение может состоять из десятков и сотен сервисов, написанных с помощью разных языков и фреймворков. Поскольку каждый сервис автономен, это означает, что в вашей промышленной среде могут находиться десятки и сотни программ. Конфигурацию серверов и сервисов больше не имеет смысла поручать администраторам. Если вы хотите развертывать свои микросервисы в крупных масштабах, вам необходимы высокоавтоматизированные процесс и инфраструктура.

На рис. 12.2 представлена обобщенная схема промышленной среды, которая позволяет разработчикам конфигурировать и администрировать свои сервисы, процессу развертывания — доставлять новые версии кода, а пользователям — получать доступ к возможностям, реализованным этими сервисами.

Промышленная среда должна поддерживать четыре ключевые возможности.

- ❑ *Интерфейс управления сервисами* позволяет разработчикам создавать, обновлять и конфигурировать сервисы. В идеале это интерфейс REST API, с которым работают консольные и графические инструменты развертывания.
- ❑ *Управление запущенными сервисами* пытается следить за тем, чтобы в промышленной среде всегда выполнялось желаемое количество экземпляров сервиса. Если экземпляр сервиса вышел из строя или по какой-то причине больше не может обслуживать запросы, промышленная среда должна его перезапустить. Если отказал целый сервер, все его сервисы должны быть перезапущены на другом компьютере.
- ❑ *Мониторинг* предоставляет разработчикам сведения о работе их сервисов, включая журнальные файлы и показатели. В случае возникновения проблем промышленная среда должна оповестить разработчиков. Мониторинг (или *наблюдаемость*) описывается в главе 11.
- ❑ *Маршрутизация* направляет запросы от пользователей к сервисам.

В этой главе обсуждаются четыре основных варианта развертывания.

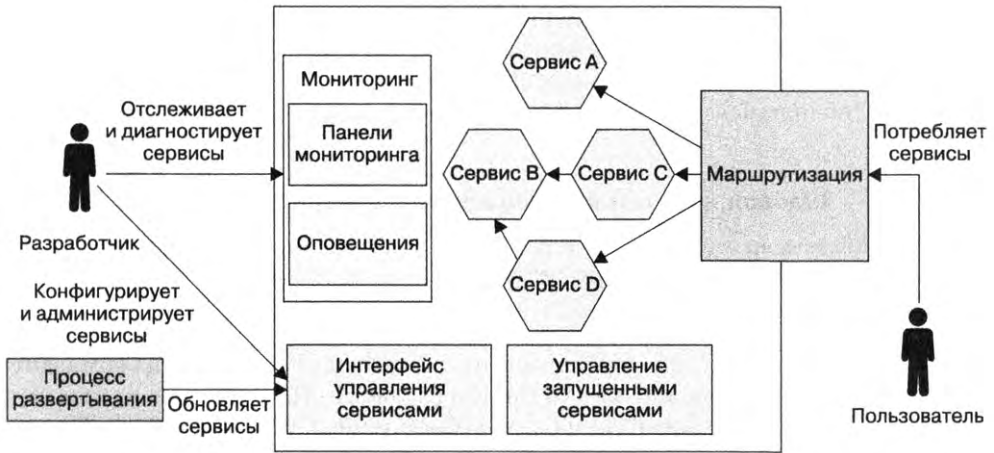


Рис. 12.2. Упрощенная схема промышленной среды. Она предоставляет четыре основные возможности: управление сервисами позволяет разработчикам развертывать и администрировать свой код, управление на этапе выполнения обеспечивает работу сервисов, мониторинг визуализирует поведение сервисов и генерирует оповещения, маршрутизация направляет сервисам запросы пользователей

- ❑ Развертывание сервиса в виде пакетов для определенных языков, таких как JAR- или WAR-файлы в Java. Этот подход заслуживает внимания в качестве демонстрации недостатков, из-за которых я рекомендую использовать один из других вариантов.
- ❑ Применение виртуальных машин упрощает процесс развертывания, так как сервисы упаковываются в виде образов для ВМ, которые инкапсулируют их технологический стек.
- ❑ Развертывание сервисов в виде контейнеров, более легковесных по сравнению с виртуальными машинами. Я покажу, как развернуть сервис **Restaurant** из приложения FTGO с помощью популярного фреймворка для оркестрации Docker под названием Kubernetes.
- ❑ Бессерверное развертывание является даже более современным, чем контейнеры. Мы посмотрим, как развернуть сервис **Restaurant** с помощью популярной бессерверной платформы AWS Lambda.

Обсудим развертывание сервисов в виде пакетов для определенных языков.

12.1. Развертывание сервисов с помощью пакетов для отдельных языков

Представьте, что вам нужно развернуть сервис **Restaurant** из приложения FTGO, написанный на Java с применением Spring Boot. Для этого можно воспользоваться форматом пакетов, предназначенным для определенного языка. При использовании этого подхода пакет развертывается в промышленную среду, а управляет им среда

выполнения сервиса. В случае с сервисом `Restaurant` это будет исполняемый файл формата `JAR` или `WAR`. В других языках, таких как `NodeJS`, сервис представляет собой каталог с исходным кодом и модулями. Такие языки, как `GoLang`, компилируют сервис в исполняемый файл конкретной операционной системы.

Шаблон «Формат пакетов для определенного языка»

Развертывает в промышленную среду пакеты, предназначенные для определенного языка. См. microservices.io/patterns/deployment/language-specific-packaging.html.

Чтобы развернуть сервис `Restaurant`, нужно сначала установить на компьютере необходимую среду выполнения — в данном случае это `JDK`. Если вы используете `WAR`-файл, вам также нужно установить веб-контейнер, такой как `Apache Tomcat`. Подготовив компьютер, вы должны скопировать на него свой пакет и запустить сервис. Каждый экземпляр сервиса выполняется в виде процесса `JVM`.

В идеале ваш процесс развертывания должен уметь автоматически доставлять сервисы в промышленную среду (рис. 12.3). Сначала собирается файл формата `JAR` или `WAR`. Затем вызывается интерфейс управления сервисами промышленной среды, который развертывает новую версию.

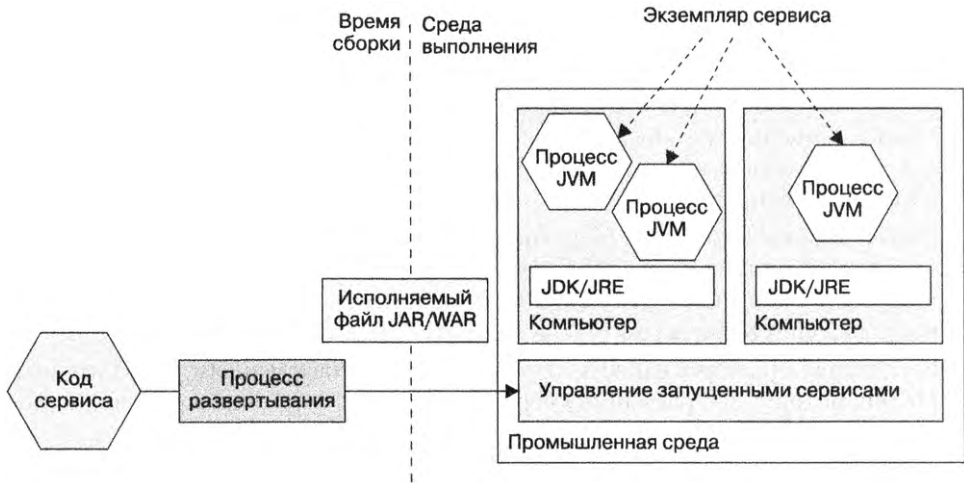


Рис. 12.3. Процесс развертывания собирает исполняемый JAR-файл и доставляет его в промышленную среду. В промышленной среде каждый экземпляр сервиса представлен JVM-машиной, запущенной на компьютере с установленным пакетом `JDK` или `JRE`

Экземпляр сервиса обычно (но не всегда) представлен единственным процессом. Например, в `Java` это процесс, в котором выполняется `JVM`. Сервис на основе `NodeJS` может породить несколько рабочих процессов для параллельной обработки запросов. Некоторые языки поддерживают развертывание множества экземпляров сервиса в рамках одного процесса.

Иногда компьютер обслуживает лишь один экземпляр сервиса, но при этом вы можете развернуть на нем дополнительные экземпляры. Например, вы можете запустить на одном сервере несколько JVM-машин (рис. 12.4). Каждая JVM-машина выполняет отдельный экземпляр сервиса.

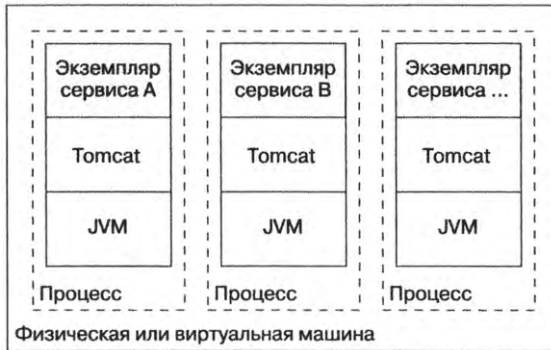


Рис. 12.4. Развертывание нескольких экземпляров сервиса на одном компьютере. Эти экземпляры могут принадлежать одному и тому же или разным сервисам. Ресурсы операционной системы распределяются между экземплярами. Каждый экземпляр представлен отдельным процессом, что обеспечивает некоторую их изоляцию

Некоторые языки позволяют запускать несколько экземпляров сервиса в одном процессе. Например, вы можете запустить несколько Java-сервисов в одном контейнере Apache Tomcat (рис. 12.5).



Рис. 12.5. Развертывание нескольких экземпляров сервиса в одном и том же веб-контейнере или сервере приложений. Эти экземпляры могут принадлежать одному и тому же или разным сервисам. Ресурсы операционной системы распределяются между экземплярами. Но поскольку они запущены в одном процессе, то никак не изолированы

Этот подход обычно используют при развертывании кода в традиционных дорогих и тяжеловесных серверах приложений, таких как WebLogic и WebSphere. Сервисы можно также упаковывать в формате OSGI и запускать по несколько их экземпляров в каждом OSGI-контейнере.

Применение пакетов, характерных для тех или иных языков, имеет положительные и отрицательные последствия. Начнем с положительных.

12.1.1. Преимущества использования пакетов для конкретных языков

Задействование пакетов для конкретных языков имеет несколько преимуществ:

- ☐ быстрое развертывание;
- ☐ эффективное задействование ресурсов, особенно при запуске нескольких экземпляров на одном компьютере или внутри одного процесса.

Рассмотрим их.

Быстрое развертывание

Одно из важнейших преимуществ этого подхода состоит в относительно высокой скорости развертывания экземпляров сервиса. Если сервис написан на Java, вы можете просто скопировать файл JAR или WAR. При использовании других языков, таких как NodeJS или Ruby, нужно скопировать исходный код. В любом случае объем информации, копируемой по сети, получается довольно небольшим.

Кроме того, запуск сервисов обычно не занимает много времени. Если сервис находится в отдельном процессе, вы можете его просто запустить. Если же это один из нескольких экземпляров, размещенных в процессе одного контейнера, у вас есть два варианта: развернуть его динамически или перезапустить весь контейнер. Благодаря отсутствию накладных расходов перезапуск сервиса обычно происходит очень быстро.

Эффективное использование ресурсов

Еще одно преимущество данного подхода связано с довольно эффективным потреблением ресурсов. Множество экземпляров сервиса находятся на одном компьютере и работают в одной ОС. Эффективность может возрасти еще сильнее, если несколько экземпляров сервиса запустить в одном процессе. Например, разные веб-приложения могут задействовать один и тот же сервер Apache Tomcat и JVM.

12.1.2. Недостатки применения пакетов для конкретных языков

Несмотря на свою привлекательность, использование пакетов для конкретных языков имеет несколько существенных недостатков:

- ☐ отсутствие инкапсуляции стека технологий;
- ☐ невозможность ограничения ресурсов, потребляемых экземпляром сервиса;

- ❑ недостаточная изоляция при запуске нескольких экземпляров сервиса на одном компьютере;
- ❑ трудности автоматического определения места размещения экземпляров сервиса.

Рассмотрим каждый из этих пунктов.

Отсутствие инкапсуляции стека технологий

Команде системных администраторов должны быть известны подробности развертывания каждого отдельного сервиса, включая конкретную версию среды выполнения. Например, веб-приложению, написанному на Java, нужны определенные версии Apache Tomcat и JDK. Администраторы должны позаботиться об установке подходящих программных пакетов.

Но что еще хуже, сервисы могут быть написаны на разных языках и с применением разных фреймворков. Также они могут использовать разные версии одних и тех же языков и библиотек. Поэтому команда разработчиков должна сообщать администраторам множество подробностей. Ведь, например, на компьютере может быть установлена не та версия среды выполнения.

Невозможность ограничения ресурсов, потребляемых экземпляром сервиса

Еще один недостаток состоит в том, что вы не можете ограничить ресурсы, потребляемые экземпляром сервиса. Процесс потенциально может занять все процессорное время или память, отнимая ресурсы у других экземпляров сервиса и операционной системы. Это может произойти, к примеру, из-за ошибки в коде.

Недостаточная изоляция при запуске нескольких экземпляров сервиса на одном компьютере

Проблема усугубляется, когда несколько экземпляров выполняются на одном компьютере. Недостаточная изоляция означает, что один вышедший из строя экземпляр может повлиять на работу других. В итоге приложение может стать ненадежным, особенно при запуске нескольких экземпляров сервиса на одном компьютере.

Трудности автоматического определения места размещения экземпляров сервиса

Еще одно затруднение при выполнении нескольких экземпляров сервиса на одном компьютере связано с определением того, где их разместить. Объем процессорного времени, памяти и т. д. у каждого сервера ограничен, и каждому экземпляру сервиса нужна какая-то часть этих ресурсов. Экземпляры сервисов следует размещать так, чтобы они эффективно использовали оборудование и не вызывали перегрузок. Чуть позже вы узнаете, что облачные платформы на основе ВМ и фреймворки

оркестрации контейнеров делают это автоматически. При развертывании сервисов без каких-либо абстракций вам, скорее всего, придется самостоятельно управлять размещением.

Как видите, привычное применение пакетов для конкретных языков все же имеет существенные недостатки. Вам следует избегать этого подхода, за исключением тех случаев, когда эффективность перевешивает все остальное.

Теперь рассмотрим современные способы развертывания сервисов, у которых нет этих проблем.

12.2. Развертывание сервисов в виде виртуальных машин

Опять-таки представьте, что вам нужно развернуть сервис **Restaurant**, но на этот раз в AWS EC2. Для этого вы можете создать и настроить сервер EC2 и скопировать на него исполняемый файл или архив WAR. В этом случае вы бы получили определенные преимущества от использования облака, но у этого подхода те же недостатки, что были описаны в предыдущем разделе. Более современным решением будет упаковать сервис в формате AMI (Amazon Machine Image) (рис. 12.6). Каждый экземпляр сервиса будет представлен сервером EC2, созданным из этого AMI-пакета. Серверами EC2 обычно управляет группа автомасштабирования AWS, которая пытается поддерживать желаемое количество работоспособных экземпляров.

Шаблон «Развертывание сервиса в виде ВМ»

Развертывает в промышленной среде сервисы, упакованные в виде образов для виртуальной машины. Каждый экземпляр сервиса является отдельной ВМ. См. microservices.io/patterns/deployment/service-per-vm.html.

Образ виртуальной машины собирается в процессе развертывания сервиса. Процесс развертывания запускает сборщик образов ВМ, чтобы создать образ с кодом сервиса и тем программным обеспечением, которое ему нужно для работы (см. рис. 12.6). Например, в случае с проектом FTGO сборщик ВМ устанавливает JDK и исполняемый JAR-файл сервиса. Он также настраивает виртуальную машину для запуска приложения с помощью системы инициализации Linux, такой как Upstart.

Существует множество инструментов, с помощью которых процесс развертывания может собрать образ ВМ. Одна из первых утилит для создания образов EC2 AMI называлась Aminator, она была создана компанией Netflix для развертывания видеовещательного сервиса на AWS (<https://github.com/Netflix/aminator>). Более современный сборщик образов ВМ — Packer. В отличие от Aminator он поддерживает

различные технологии виртуализации, включая EC2, Digital Ocean, Virtual Box и VMware (www.packer.io). Для его использования нужно написать конфигурационный файл, в котором указать базовый образ и набор средств подготовки для установки и конфигурации AMI.

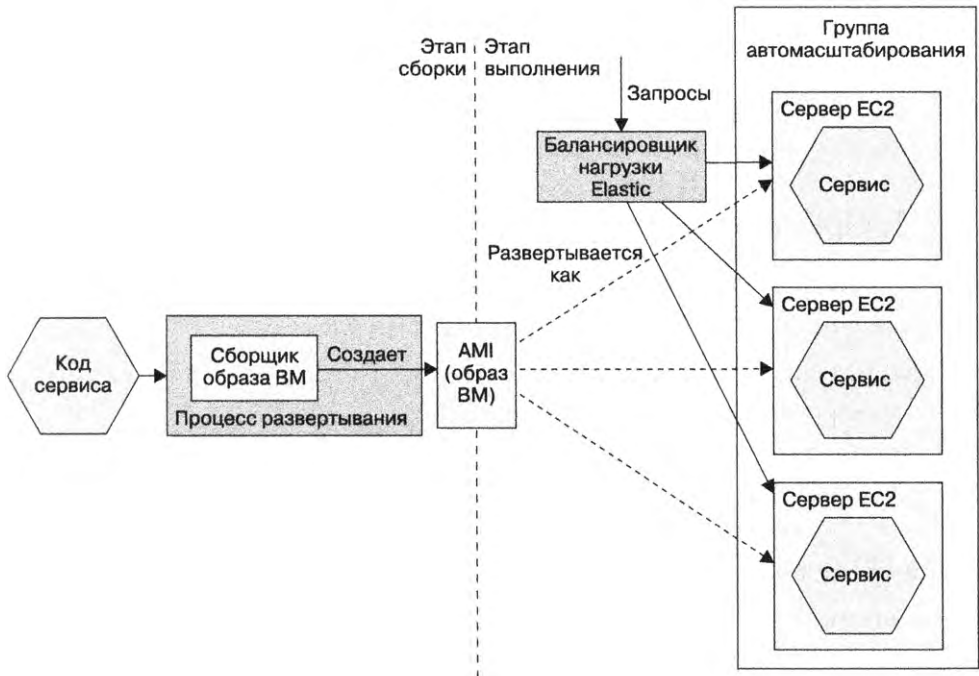


Рис. 12.6. Процесс развертывания упаковывает сервис в виде образа VM, такого как EC2 AMI, который содержит все необходимое для его работы, включая среду выполнения языка. На этапе выполнения каждый экземпляр сервиса представлен VM, такой как сервер EC2, основанной на этом образе. Балансировщик нагрузки Elastic направляет запросы к этим экземплярам

06 Elastic Beanstalk

Elastic Beanstalk — это простое решение для развертывания сервисов с помощью VM, которое входит в состав AWS. Вы загружаете свой код, например WAR-файл, а Elastic Beanstalk развертывает его в виде одного или нескольких управляемых серверов EC2 с поддержкой балансирования нагрузки. Возможно, этот инструмент не такой модный, как, скажем, Kubernetes, но он позволяет легко развертывать микросервисные приложения в EC2.

Интересно, что Elastic Beanstalk сочетает в себе элементы трех шаблонов развертывания, описываемых в этой главе. Он поддерживает пакеты нескольких форматов для языков Java, Ruby и .NET. Он развертывает приложения в виде виртуальных машин,

но вместо сборки AMI использует базовый образ, который устанавливает приложение во время запуска.

Elastic Beanstalk также умеет развертывать контейнеры Docker. Каждый сервер EC2 выполняет набор из одного или нескольких контейнеров. В отличие от фреймворков оркестрации Docker, с которыми мы познакомимся позже, единицей масштабирования здесь является сервер EC2, а не контейнер.

Теперь рассмотрим положительные и отрицательные стороны этого подхода.

12.2.1. Преимущества развертывания сервисов в виде VM

Развертывание сервисов в виде VM имеет ряд преимуществ.

- ❑ Образ VM инкапсулирует стек технологий.
- ❑ Экземпляры сервиса изолированы.
- ❑ Используется зрелая облачная инфраструктура.

Давайте их рассмотрим.

Образ VM инкапсулирует стек технологий

Важное преимущество этого шаблона состоит в том, что образ VM содержит сервис вместе со всеми его зависимостями. Это избавляет от необходимости устанавливать и конфигурировать ПО, нужное для работы сервиса, что может спасти от потенциальных ошибок. После упаковки в образ VM сервис становится своеобразным черным ящиком, который инкапсулирует свой стек технологий. Такой образ можно развертывать где угодно, не внося изменений. API для развертывания сервиса становится интерфейсом для управления VM, а сам процесс существенно упрощается и становится более надежным.

Экземпляры сервиса изолированы

Существенным преимуществом виртуальных машин является то, что экземпляры сервисов выполняются в полной изоляции. Это, в конце концов, основная цель данной технологии. Каждая VM имеет фиксированное количество процессорного времени и памяти, что не позволяет ей занимать ресурсы других сервисов.

Использование зрелой облачной инфраструктуры

Еще одна положительная сторона развертывания микросервисов в виде виртуальных машин — возможность задействования зрелой, высокоавтоматизированной облачной инфраструктуры. Публичные облака, например AWS, пытаются запланировать

работу ВМ на физических серверах так, чтобы избежать перегрузок. Они также предоставляют полезные возможности — автомасштабирование и балансирование трафика между ВМ.

12.2.2. Недостатки развертывания сервисов в виде ВМ

Развертывание сервисов в виде ВМ имеет и недостатки.

- ❑ Менее эффективно используются ресурсы.
- ❑ Развертывание протекает довольно медленно.
- ❑ Требуются дополнительные расходы на системное администрирование.

Рассмотрим их.

Менее эффективное использование ресурсов

Каждый экземпляр сервиса тянет за собой целую виртуальную машину, включая ее операционную систему. Более того, публичные платформы IaaS обычно предлагают ограниченный набор размеров ВМ, поэтому ваши ресурсы, скорее всего, будут использоваться не на полную мощность. Это в меньшей мере относится к сервисам, основанным на Java, поскольку они относительно тяжеловесны. Но развертывание таким образом легковесных сервисов, написанных на NodeJS или GoLang, может оказаться неэффективным.

Довольно медленное развертывание

Сборка образа ВМ обычно исчисляется минутами из-за размера виртуальной машины. Для этого по сети нужно передать довольно много данных. Создание экземпляра ВМ из образа тоже требует некоторого времени — опять-таки из-за количества данных, перемещаемых по сети. К тому же операционная система, выполняемая внутри ВМ, загружается не сразу, хотя понятие «медленно» является относительным. Процесс может растянуться на минуты, но это все равно быстрее, чем традиционное развертывание. Вместе с тем он значительно уступает по скорости более легковесным шаблонам, с которыми вы вскоре познакомитесь.

Дополнительные расходы на системное администрирование

Вы сами отвечаете за обновление операционной системы и среды выполнения. Системное администрирование может показаться неотъемлемой частью развертывания ПО, но в разделе 12.5 я опишу бессерверное развертывание, в котором этот аспект полностью искоренен.

Теперь поговорим об альтернативных способах развертывания микросервисов, которые, несмотря на свою легковесность, обладают многими преимуществами ВМ.

12.3. Развертывание сервисов в виде контейнеров

Контейнеры — это более легковесный современный механизм развертывания. Они используют механизм виртуализации на уровне операционной системы. Контейнер обычно состоит из одного процесса (хотя их может быть несколько), запущенного в среде, изолированной от других контейнеров (рис. 12.7). Например, контейнер с Java-сервисом, как правило, состоит из процесса JVM.



Рис. 12.7. Контейнер состоит из одного или нескольких процессов, запущенных в изолированной среде. На одном компьютере обычно запускается несколько контейнеров, использующих общую операционную систему

Процесс, запущенный внутри контейнера, ведет себя так, словно ему отведен целый отдельный сервер. Обычно он имеет собственный IP-адрес, что позволяет избежать конфликтов с портами, — все Java-процессы, к примеру, могут прослушивать порт 8080. У каждого контейнера есть своя корневая файловая система. Для изоляции контейнеров друг от друга их среда выполнения использует механизмы операционной системы. Наиболее популярная среда выполнения контейнеров — Docker, есть и альтернативы, такие как Solaris Zones.

Шаблон «Развертывание сервиса в виде контейнера»

Развертывает в среде выполнения сервис, упакованный в виде образа контейнера. Каждый экземпляр сервиса является отдельным контейнером. См. microservices.io/patterns/deployment/service-per-container.html.

При создании контейнера можно указать его процессор, объем памяти и в зависимости от реализации ресурсы ввода/вывода. Среда выполнения контейнеров следит за соблюдением этих ограничений и не дает им занять все ресурсы компьютера. Это особенно важно при использовании фреймворков оркестрации Docker, таких как Kubernetes, так как эти лимиты учитываются при выборе серверов для запуска контейнеров, что позволяет избежать их перегрузки.

На рис. 12.8 показан процесс развертывания сервиса в виде контейнера. На этапе сборки применяется специальный инструмент, который считывает код сервиса и описание его образа, чтобы создать образ контейнера и сохранить его в реестр. Во время выполнения этот образ извлекается из реестра и используется для создания контейнеров.

Давайте подробнее рассмотрим этапы сборки и выполнения.

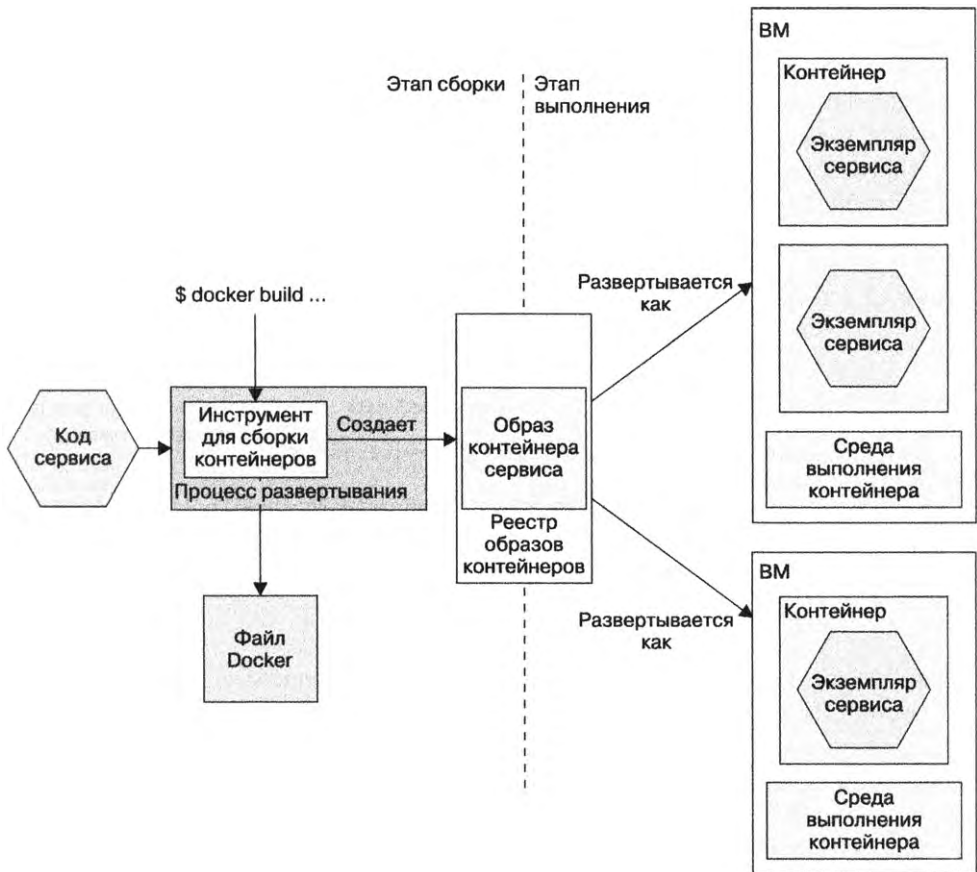


Рис. 12.8. Сервис упаковывается в виде образа контейнера, который хранится в реестре. Во время выполнения сервис состоит из нескольких контейнеров, созданных из этого образа. Контейнеры обычно запускаются в виртуальных машинах. Одна ВМ, как правило, выполняет несколько контейнеров

12.3.1. Развертывание сервисов с помощью Docker

Чтобы развернуть сервис в виде контейнера, его необходимо упаковать в образ. *Образ контейнера* — это слепок файловой системы, состоящий из приложения и любого ПО, которое требуется для его работы. Часто для этого берут полноценную корневую файловую систему Linux, хотя встречаются и более легкие образы. Например, чтобы развернуть сервис на основе Spring Boot, нужно собрать образ контейнера, в который входят исполняемый JAR-файл сервиса и подходящая версия JDK. Точно так же для развертывания веб-приложения, написанного на Java, собирается образ контейнера с WAR-файлом, Apache Tomcat и JDK.

Сборка образа Docker

Первым шагом при сборке образа становится создание файла `Dockerfile`, который описывает то, как Docker будет собирать этот образ. В нем указываются базовый образ контейнера, последовательность инструкций для установки ПО и конфигурации контейнера и команда оболочки, выполняемая при создании контейнера. В листинге 12.1 показан `Dockerfile`, который собирает образ для сервиса `Restaurant`. Результат будет содержать исполняемый JAR-файл этого сервиса. Контейнер конфигурируется для выполнения при запуске команды `java -jar`.

Листинг 12.1. `Dockerfile` для сборки сервиса `Restaurant`

Базовый образ

Устанавливаем curl для проверки работоспособности

Настраиваем Docker для выполнения java -jar .. при запуске контейнера

Копирует JAR из каталога сборки Gradle в образ

Делаем так, чтобы Docker обратился к конечной точке для проверки работоспособности

```
FROM openjdk:8u171-jre-alpine
RUN apk --no-cache add curl
CMD java ${JAVA_OPTS} -jar ftgo-restaurant-service.jar
HEALTHCHECK --start-period=30s --interval=5s CMD curl http://localhost:8080/actuator/health || exit 1
COPY build/libs/ftgo-restaurant-service.jar .
```

`openjdk:8u171-jre-alpine` — это минимальный образ Linux с содержащимся в нем JRE. `Dockerfile` копирует JAR-архив сервиса в этот образ и конфигурирует контейнер для выполнения этого архива во время запуска. Он также настраивает Docker для периодического обращения к конечной точке с данными о работоспособности (см. главу 11). Директива `HEALTHCHECK` приводит к вызову этой конечной точки раз в 5 с после начальной 30-секундной задержки, которая дает сервису время на запуск.

Написав `Dockerfile`, вы можете собрать образ. В листинге 12.2 показаны команды оболочки, которые собирают образ для сервиса `Restaurant`. Этот скрипт компилирует JAR-файл с содержащимся в нем сервисом и выполняет команду `docker build` для создания образа.

Листинг 12.2. Командные оболочки, которые собирают образ для сервиса Restaurant

```
cd ftgo-restaurant-service
../gradlew assemble
docker build -t ftgo-restaurant-service .
```

Переходим в каталог сервиса

Компилируем JAR сервиса

Собираем образ

У команды `docker build` есть два аргумента: ключ `-t`, определяющий имя образа, и `.` — то, что в терминологии Docker называется контекстом. *Контекст* в данном случае является текущим каталогом и состоит из `Dockerfile` и файлов для сборки образа. Команда `docker build` передает контекст демону Docker, который выполняет сборку.

Загрузка образа Docker в реестр

Последний шаг в процессе сборки — загрузка свеже созданного образа Docker в так называемый реестр. *Реестр* Docker — это эквивалент репозитория Maven для библиотек Java или реестра NPM для пакетов NodeJS. Docker Hub служит примером публичного реестра Docker и эквивалентом Maven Central и NpmJS.org. Но в своих проектах вы, скорее всего, будете использовать приватный реестр, предоставляемый такими сервисами, как Docker Cloud Registry или AWS EC2 Container Registry.

Для загрузки образа в реестр необходимо выполнить две команды. Первая команда, `docker tag`, присваивает образу название с сетевым именем в качестве префикса, а также указывает порт реестра, если это необходимо. Название образа содержит также суффикс в виде номера версии — это будет важно при обновлении сервиса. Например, если сетевое имя реестра равно `registry.acme.com`, для маркирования образа нужно использовать следующую команду:

```
docker tag ftgo-restaurant-service registry.acme.com/ftgo-restaurant-  
service:1.0.0.RELEASE
```

Затем выполняется команда `docker push`, которая загружает промаркированный образ в реестр:

```
docker push registry.acme.com/ftgo-restaurant-service:1.0.0.RELEASE
```

Эта команда обычно занимает намного меньше времени, чем можно было бы ожидать. Дело в том, что образы Docker имеют так называемую *многослойную файловую систему*, что позволяет передавать по сети только их часть. Операционная система образа, среда выполнения Java и само приложение находятся в разных слоях. Docker нужно передать только те слои, которых не существует в реестре. В итоге, если передается только слой приложения, который обычно занимает лишь небольшую часть образа, загрузка происходит довольно быстро.

Разобравшись с загрузкой образа, мы можем перейти к созданию контейнера.

Запуск контейнера Docker

Упаковав свой сервис в виде образа, можете его запустить. Инфраструктура Docker загрузит образ из реестра на рабочий сервер и создаст из него один или несколько контейнеров. Каждый контейнер будет служить экземпляром вашего сервиса.

Как можно было бы ожидать, Docker предоставляет команду `docker run`, которая создает и запускает контейнер. Использование этой команды на примере сервиса `Restaurant` продемонстрировано в листинге 12.3. Она имеет несколько аргументов, включая название образа и переменные окружения, которые будут заданы в среде выполнения контейнера. Последние применяются для передачи внешней конфигурации, такой как сетевое размещение базы данных и пр.

Листинг 12.3. Использование команды `docker run` для запуска контейнера сервиса

```
docker run \
  -d \
  --name ftgo-restaurant-service \
  -p 8082:8080 \
  -e SPRING_DATASOURCE_URL=... -e SPRING_DATASOURCE_USERNAME=... \
  -e SPRING_DATASOURCE_PASSWORD=... \
  registry.acme.com/ftgo-restaurant-service:1.0.0.RELEASE
```

Запускает его в виде фонового демона

Название контейнера

Привязывает порт контейнера 8080 к порту 8082 родительского узла

Переменные окружения

Запускаемый образ

Команда `docker run` достает образ из реестра, если это необходимо. Затем она создает и запускает контейнер, который выполняет команду `java -jar`, указанную в `Dockerfile`.

На первый взгляд, команда `docker run` выглядит довольно просто, но у нее есть несколько проблем. Прежде всего, она не обеспечивает надежное развертывание сервиса, поскольку созданный ею контейнер запускается лишь на одном компьютере. Ядро Docker предоставляет некоторые базовые возможности управления, такие как автоматический перезапуск контейнера, если тот вышел из строя (или в случае перезагрузки сервера). Но при этом не учитывается риск отказа родительской системы.

Еще одна проблема связана с тем, что сервисы обычно не существуют сами по себе. Им нужны другие компоненты, такие как база данных и брокер сообщений. Было бы неплохо развертывать и удалять сервис как единое целое со всеми его зависимостями.

Лучшим решением, особенно на этапе разработки, является использование Docker Compose. Docker Compose — это инструмент для декларативного описания набора контейнеров в виде файла YAML с возможностью их последующего группового запуска и остановки. Кроме того, в файле YAML удобно указывать многочисленные свойства внешней конфигурации. Для более тесного знакомства с Docker Compose я рекомендую прочитать книгу Джеффа Николоффа (Jeff Nickoloff) *Docker in Action* (Manning, 2016) и просмотреть файл `docker-compose.yml` в примере кода.

Однако у Docker Compose есть одна проблема — он ограничен одним компьютером. Для надежного развертывания сервисов необходимо задействовать фреймворк оркестрации Docker, такой как Kubernetes, который превращает набор серверов в пул ресурсов. Я покажу, как использовать Kubernetes, в разделе 12.4. А сначала рассмотрим положительные и отрицательные стороны контейнеризации.

12.3.2. Преимущества развертывания сервисов в виде контейнеров

Развертывание сервисов в виде контейнеров имеет несколько преимуществ, которые в основном свойственны и виртуальным машинам.

- ❑ Инкапсуляция стека технологий, благодаря которой API для управления сервисом превращается в API контейнера.
- ❑ Экземпляры сервиса изолированы.
- ❑ Ресурсы экземпляров сервиса ограничены.

Но, в отличие от виртуальных машин, контейнеры — это легковесная технология. Сборка их образов обычно протекает быстро. Например, на упаковку приложения на основе Spring Boot в виде образа контейнера мой ноутбук тратит всего 5 с. Перемещение образов по сети, например в реестр и из него, тоже происходит довольно быстро — в основном благодаря тому, что передаются лишь некоторые его слои. Еще контейнеры очень быстро запускаются, минуя длительный процесс загрузки ОС. Запуску подлежит лишь сам сервис.

12.3.3. Недостатки развертывания сервисов в виде контейнеров

Существенный недостаток контейнеров состоит в том, что вы должны постоянно заниматься управлением образами контейнеров и обновлением операционной системы вместе со средой выполнения. Кроме того, если вы не применяете облачные контейнерные решения наподобие Google Container Engine или AWS ECS, на вас ложится администрирование контейнерной инфраструктуры и, возможно, инфраструктуры виртуальных машин, поверх которой она работает.

12.4. Развертывание приложения FTGO с помощью Kubernetes

Итак, мы рассмотрели контейнеры и взвесили их плюсы и минусы. Теперь поговорим о том, как развернуть сервис **Restaurant** приложения FTGO с помощью Kubernetes. Утилита **Docker Compose**, описанная в разделе 12.3.1, отлично подходит для разработки и тестирования. Но для надежного запуска контейнеризированных сервисов в промышленных условиях нужно более продвинутое решение, такое как Kubernetes. Kubernetes — это фреймворк оркестрации **Docker**, программный слой поверх **Docker**, который объединяет набор серверов в единый пул ресурсов для запуска сервисов. Он постоянно пытается поддерживать желаемое количество запущенных экземпляров сервиса, даже в случае возникновения неполадок в этих экземплярах или серверах. Легкость контейнеров и богатые возможности Kubernetes выводят развертывание сервисов на новый уровень.

Я начну этот раздел с обзора платформы Kubernetes, ее возможностей и архитектуры. Вслед за этим покажу, как с ее помощью развертывать сервисы. Kubernetes — это сложная система, и ее исчерпывающий анализ выходит за рамки книги. Я лишь покажу, как ее могут применять разработчики. Больше информации можно найти в книге Марко Луксы (Marko Luksa) *Kubernetes in Action* (Manning, 2018).

12.4.1. Обзор Kubernetes

Kubernetes — это *фреймворк оркестрации Docker*, который обращается с набором серверов под управлением Docker, как с пулом ресурсов. Вы лишь указываете, сколько экземпляров сервиса нужно запустить, а фреймворк делает все остальное. Архитектура фреймворка оркестрации Docker представлена на рис. 12.9.

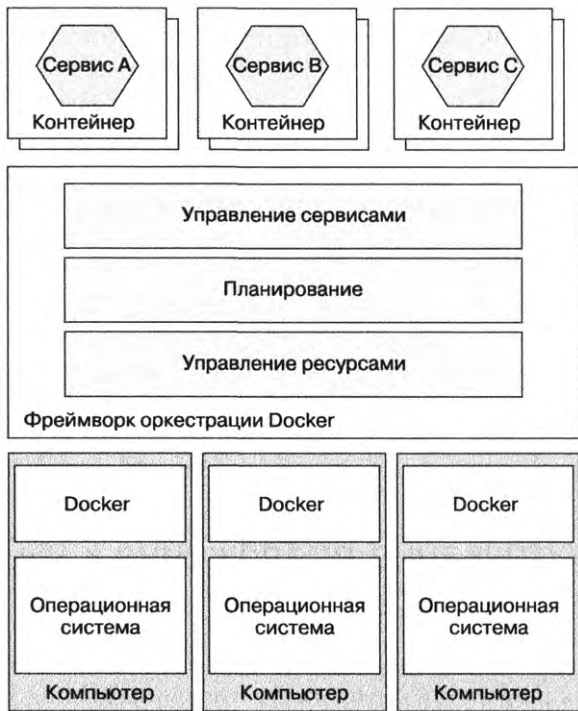


Рис. 12.9. Фреймворк оркестрации Docker превращает набор серверов под управлением Docker в кластер ресурсов. Он назначает контейнеры определенным серверам и постоянно пытается поддерживать желаемое количество работоспособных контейнеров

Фреймворк оркестрации Docker, такой как Kubernetes, имеет три основные функции.

- ❑ *Управление ресурсами.* Обращается с кластером серверов как с пулом процессов, памяти и томов хранилища, объединяя их в единый абстрактный компьютер.

- ❑ *Планирование*. Выбирает сервер для выполнения вашего контейнера. По умолчанию планировщик учитывает требования к ресурсам, предъявляемые контейнером, и свободные ресурсы на каждом узле. Он также поддерживает принципы *принадлежности* (affinity) и *непринадлежности* (anti-affinity), которые позволяют размещать контейнеры на одном и том же или на разных узлах.
- ❑ *Управление сервисом*. Реализует концепцию именованных и версионных сервисов, которые накладываются непосредственно на сервисы в микросервисной архитектуре. Фреймворк оркестрации постоянно поддерживает желаемое количество работоспособных экземпляров и распределяет между ними запросы. Он также выполняет плавающие обновления сервиса и позволяет откатиться к предыдущей версии.

Фреймворки оркестрации Docker все чаще применяются для развертывания приложений. Фреймворк Docker Swarm является частью ядра Docker, поэтому он прост в настройке и использовании. Платформу Kubernetes намного сложнее конфигурировать и администрировать, но у нее значительно больше возможностей. На момент написания книги она переживает взрывной рост популярности и имеет огромное сообщество. Посмотрим, как она работает.

Архитектура Kubernetes

Kubernetes выполняется на кластере компьютеров. Архитектура кластера Kubernetes показана на рис. 12.10. Каждый компьютер в этом кластере является либо ведущим, либо служебным узлом. Обычно ведущих узлов очень мало (иногда один), а рабочих — много. *Ведущий* компьютер отвечает за управление кластером. Рабочий узел выполняет одну или несколько *pod-оболочек*. *Pod-оболочка* — это единица развертывания в Kubernetes, состоящая из набора контейнеров.

Ведущий узел содержит несколько компонентов, включая следующие:

- ❑ *API-сервер* — интерфейс REST API для развертывания и администрирования сервисов. Используется, к примеру, утилитой командной строки `kubectl`;
- ❑ *Etc*d — база данных NoSQL типа «ключ — значение», хранящая данные кластера;
- ❑ *планировщик* — выбирает узел для запуска *pod-оболочки*;
- ❑ *диспетчер контроллеров* — запускает контроллеры, которые следят за тем, чтобы кластер находился в нужном состоянии. Например, контроллер *репликации* (это лишь один из типов контроллеров) обеспечивает выполнение желаемого количества экземпляров сервиса, отвечая за их запуск и удаление.

Рабочий узел тоже выполняет несколько компонентов, включая следующие:

- ❑ *Kubelet* — создает *pod-оболочки* и управляет их выполнением на рабочем узле;
- ❑ *Kube-proxy* — управляет сетевыми функциями, включая балансирование нагрузки между *pod-оболочками*;
- ❑ *pod-оболочки* — сервисы приложения.

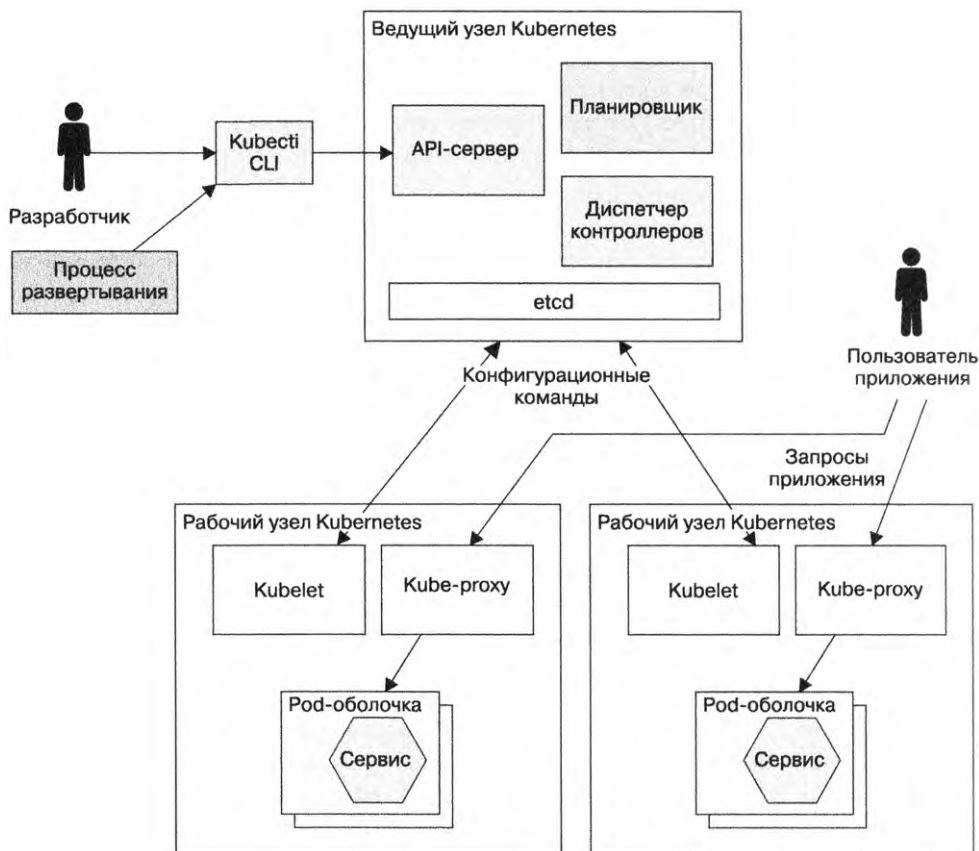


Рис. 12.10. Кластер Kubernetes состоит из ведущих и рабочих узлов, первые отвечают за управление кластером, а последние выполняют сервисы. Разработчики и процесс развертывания взаимодействуют с Kubernetes через API-сервер, который вместе с другим управляющим ПО находится на ведущем узле. Контейнеры приложения выполняются на рабочих узлах. Каждый рабочий узел запускает утилиты Kubelet и Kube-proxy, первая управляет контейнером приложения, а вторая направляет пользовательские запросы к pod-оболочкам — либо напрямую в качестве прокси, либо опосредованно, конфигурируя правила маршрутизации брандмауэра iptables, встроенного в ядро Linux

Теперь рассмотрим ключевые концепции, которыми вам нужно овладеть для того, чтобы развертывать сервисы в Kubernetes.

Ключевые концепции Kubernetes

Как упоминалось во введении к этому разделу, Kubernetes — довольно сложная система. Однако для ее продуктивного использования достаточно овладеть несколькими ключевыми концепциями, которые называются *объектами*. Kubernetes поддерживает объекты многих типов. С точки зрения разработчика самые важные из них следующие.

- ❑ *Pod-оболочка*. Это базовая единица развертывания в Kubernetes. Она состоит из одного или нескольких контейнеров с общими IP-адресом и томами хранения. Pod-оболочка экземпляра сервиса часто содержит лишь один контейнер, который, к примеру, выполняет JVM. Но в некоторых случаях в ее состав может входить несколько *дополнительных* контейнеров, реализующих вспомогательные функции. Например, у сервера NGINX может быть дополнительный контейнер, который периодически выполняет команду `git pull`, загружая последнюю версию веб-сайта. Pod-оболочка является временной, поскольку ее контейнер или узел, на котором она выполняется, могут выйти из строя.
- ❑ *Развертывание*. Декларативная спецификация pod-оболочки. Это контроллер, который постоянно обеспечивает нужное количество запущенных экземпляров сервиса (pod-оболочки). Для поддержки версионирования он использует плавающие обновления и откаты. В подразделе 12.4.2 вы увидите, что в терминологии Kubernetes каждый сервис в микросервисной архитектуре является развертыванием.
- ❑ *Сервис*. Предоставляет клиентам сервиса статический/стабильный сетевой адрес. Это разновидность механизма обнаружения сервисов на уровне инфраструктуры, описанного в главе 3. Сервис имеет IP-адрес и DNS-имя, которое на него указывает, TCP- и UDP-трафик распределяются между несколькими pod-оболочками, если их больше одной. IP-адрес и DNS-имя доступны только внутри Kubernetes. Позже я покажу, как сконфигурировать сервисы таким образом, чтобы они были доступны из-за пределов кластера.
- ❑ *ConfigMap*. Именованный набор пар «имя — значение», который описывает внешнюю конфигурацию для одного или нескольких сервисов приложения (краткий обзор вынесения конфигурации вовне сделан в главе 11). Определение контейнера pod-оболочки может ссылаться на ConfigMap для перечисления переменных окружения. Оно может использовать ConfigMap также для создания конфигурационных файлов внутри контейнера. Вы можете хранить конфиденциальную информацию, например пароли, в варианте ConfigMap под названием Secret.

На этом мы заканчиваем обзор ключевых концепций Kubernetes. Теперь применим их на практике, развернув сервис приложения на этой платформе.

12.4.2. Развертывание сервиса Restaurant в Kubernetes

Как упоминалось ранее, для запуска сервиса в Kubernetes необходимо определить его развертывание. Самый простой способ создания объекта Kubernetes, такого как развертывание, состоит в написании файла в формате YAML. В листинге 12.4 показан YAML-файл, описывающий развертывание для сервиса Restaurant. Оно должно запустить две реплики (копии) pod-оболочки. Pod-оболочка содержит лишь один контейнер. В определении контейнера указаны запускаемый образ Docker и другие атрибуты, такие как значения переменных окружения. Переменные окружения контейнера играют роль внешней конфигурации сервиса. Они считаются фреймворком Spring Boot и становятся доступными в виде свойств в контексте приложения.

Листинг 12.4. Развертывание Kubernetes для ftgo-restaurant-service

```

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ftgo-restaurant-service
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: ftgo-restaurant-service
    spec:
      containers:
        - name: ftgo-restaurant-service
          image: msapatterns/ftgo-restaurant-service:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
            name: httpport
          env:
            - name: JAVA_OPTS
              value: "-Dsun.net.inetaddr.ttl=30"
            - name: SPRING_DATASOURCE_URL
              value: jdbc:mysql://ftgo-mysql/eventuate
            - name: SPRING_DATASOURCE_USERNAME
              valueFrom:
                secretKeyRef:
                  name: ftgo-db-secret
                  key: username
            - name: SPRING_DATASOURCE_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: ftgo-db-secret
                  key: password
            - name: SPRING_DATASOURCE_DRIVER_CLASS_NAME
              value: com.mysql.jdbc.Driver
            - name: EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS
              value: ftgo-kafka:9092
            - name: EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING
              value: ftgo-zookeeper:2181
          livenessProbe:
            httpGet:
              path: /actuator/health
              port: 8080
            initialDelaySeconds: 60
            periodSeconds: 20
          readinessProbe:
            httpGet:
              path: /actuator/health
              port: 8080
            initialDelaySeconds: 60
            periodSeconds: 20

```

Указывает на то, что это объект типа Deployment

Название развертывания

Количество реплик pod-оболочки

Назначает каждой pod-оболочке метку с названием app и значением ftgo-restaurant-service

Спецификация pod-оболочки, которая определяет лишь один контейнер

Порт контейнера

Переменные окружения контейнера, которые читает Spring Boot

Требующие особого отношения значения, которые извлекаются из объекта Secret под названием ftgo-db-secret

Конфигурируем Kubernetes, чтобы вызывать конечную точку для проверки работоспособности

Определение этого развертывания конфигурирует платформу Kubernetes так, чтобы она вызывала конечную точку для проверки работоспособности, принадлежащую сервису **Restaurant**. Как говорилось в главе 11, эта конечная точка позволяет Kubernetes определить состояние экземпляра сервиса. Kubernetes поддерживает два вида проверок. Первая называется **readinessProbe**, она определяет, нужно ли направлять трафик к экземпляру сервиса. В этом примере Kubernetes каждые 20 с вызывает по HTTP конечную точку `/actuator/health`, но только после начальной 30-секундной задержки, которая дает сервису время на инициализацию. Если какое-то количество последовательных проверок типа **readinessProbe** (по умолчанию одна) заканчивается успешно, Kubernetes считает сервис готовым к работе, при получении подряд определенного количества отказов (по умолчанию три) сервис считается неготовым. Kubernetes направляет трафик к сервису только в том случае, если проверка **readinessProbe** указывает на его готовность.

Вторая проверка называется **livenessProbe**. Она настраивается так же, как и **readinessProbe**. Но вместо определения того, следует ли направлять трафик к экземпляру сервиса, она подсказывает Kubernetes, нужно ли этот экземпляр удалить и запустить заново. Если какое-то количество последовательных проверок **livenessProbe** завершается неудачно, Kubernetes перезапускает сервис.

Написав YAML-файл, вы можете создать или обновить развертывание с помощью команды **kubectl apply**:

```
kubectl apply -f ftgo-restaurant-service/src/deployment/kubernetes/ftgo-restaurant-service.yml
```

Эта команда делает запрос к API-серверу Kubernetes, результатом которого будет создание развертывания и **pod**-оболочек.

Но перед этим вы должны создать объект **Secret** **ftgo-db-secret**. Это можно сделать быстрым и не совсем безопасным способом:

```
kubectl create secret generic ftgo-db-secret \
  --from-literal=username=mysqluser --from-literal=password=mysqlpw
```

Данная команда создает объект **Secret** с идентификатором и паролем пользователя базы данных, указанными в командной строке. Другие, более безопасные способы описаны в документации Kubernetes (kubernetes.io/docs/concepts/configuration/secret/#creating-your-own-secrets).

Создание сервиса в Kubernetes

На этом этапе **pod**-оболочки уже запущены, и развертывание Kubernetes будет стараться поддерживать их в рабочем состоянии. Но проблема в том, что IP-адреса **pod**-оболочек были назначены динамически и по этой причине клиент не может ими воспользоваться для выполнения HTTP-запросов. Решение, описанное в главе 3, состоит в применении механизма обнаружения.

Обнаружение можно выполнять на клиентской стороне, установив реестр сервисов, такой как Netflix OSS Eureka. К счастью, чтобы этого избежать, достаточно определить сервис Kubernetes и использовать встроенный механизм обнаружения.

Сервис — это объект Kubernetes, который предоставляет клиентам одну или несколько pod-оболочек со стабильной конечной точкой. Он имеет IP-адрес и DNS-имя, которое на него ссылается. Сервис распределяет между pod-оболочками трафик, приходящий на этот IP-адрес. В листинге 12.5 показан сервис Kubernetes для RestaurantService. Он направляет трафик из `http://ftgo-restaurant-service:8080` к pod-оболочкам, определенным в развертывании, представленном в листинге.

Листинг 12.5. Определение сервиса Kubernetes для ftgo-restaurant-service в формате YAML

```
apiVersion: v1
kind: Service
metadata:
  name: ftgo-restaurant-service
spec:
  ports:
    - port: 8080
      targetPort: 8080
  selector:
    app: ftgo-restaurant-service
---
```

Ключевая часть определения сервиса — раздел `selector`, который выбирает целевые pod-оболочки, то есть те, у которых есть метка `app` со значением `ftgo-restaurant-service`. Если внимательно присмотреться, можно увидеть, что у контейнера, описанного в листинге 12.4, есть такая метка.

Написав YAML-файл, вы можете создать сервис с помощью следующей команды:

```
kubectl apply -f ftgo-restaurant-service-service.yml
```

Благодаря тому что мы создали сервис Kubernetes, любые клиенты RestaurantService, работающие внутри кластера Kubernetes, могут обращаться к его интерфейсу REST API по адресу `http://ftgo-restaurant-service:8080`. Позже я покажу, как обновлять запущенные сервисы, но сначала посмотрим, как сделать наш сервис доступным из-за пределов кластера Kubernetes.

12.4.3. Развертывание API-шлюза

Сервис Kubernetes для RestaurantService, показанный в листинге 12.5, доступен только внутри кластера. Для самого сервиса это не проблема, но как насчет API-шлюза? Его роль состоит в направлении внешнего трафика к внутренним компонентам. Поэтому он должен быть доступен извне. К счастью, Kubernetes поддерживает и этот сценарий использования. Сервис, рассмотренный ранее, имеет тип ClusterIP, применяемый по умолчанию, но в нашем распоряжении есть два других типа: NodePort и LoadBalancer.

Сервисы типа NodePort доступны на общекластерном порте на всех рабочих узлах кластера. Любой трафик, направленный на этот порт на любом из рабочих узлов кластера, распределяется между внутренними pod-оболочками. Номер порта должен

относиться к диапазону 30 000–32 767. Например, в листинге 12.6 показан сервис, который направляет трафик на порт 30 000 сервиса `Consumer`.

Листинг 12.6. YAML-определение сервиса типа `NodePort`, который направляет трафик на порт 30 000 сервиса `Consumer`

```
apiVersion: v1
kind: Service
metadata:
  name: ftgo-api-gateway
spec:
  type: NodePort  ← Задает тип NodePort
  ports:
    - nodePort: 30000  ← Общекластерный порт
      port: 80
      targetPort: 8080
  selector:
    app: ftgo-api-gateway
---
```

API-шлюз доступен внутри кластера по адресу `http://ftgo-api-gateway`, для обращения к нему извне используется URL `http://<node-ip-address>:30000/`, где `node-ip-address` — это IP-адрес одного из рабочих узлов. Сконфигурировав сервис типа `NodePort`, вы можете, к примеру, настроить AWS Elastic Load Balancer (ELB), чтобы распределять внешний трафик между узлами. Ключевое преимущество этого подхода заключается в том, что балансировщик ELB находится под вашим полным контролем и обеспечивает максимальную гибкость при настройке.

Однако сервисы типа `NodePort` — это не единственный вариант. Вы можете использовать также тип `LoadBalancer`, который автоматически конфигурирует балансировщик нагрузки для конкретной облачной платформы. В случае с AWS это будет ELB. Одно из преимуществ сервисов этого типа состоит в том, что вам больше не нужно настраивать собственный балансировщик. Но есть и обратная сторона: несмотря на то что Kubernetes предоставляет несколько параметров для настройки ELB, таких как SSL-сертификат, конфигурация балансировщика теряет свою гибкость.

12.4.4. Развертывание без простоя

Представьте, что вы обновили сервис `Restaurant` и хотите развернуть эти изменения в промышленной среде. Kubernetes превращает обновление запущенного сервиса в простой процесс, состоящий из трех шагов.

1. Сборка и загрузка в реестр нового контейнера. Здесь используется описанный ранее процесс. Разница лишь в том, что образ получит метку с другой версией, например `ftgo-restaurant-service:1.1.0.RELEASE`.
2. Редактирование YAML-файла с развертыванием сервиса таким образом, чтобы оно ссылалось на новый образ.
3. Обновление развертывания с помощью команды `kubectl apply -f`.

После этого платформа Kubernetes выполнит плавающее обновление pod-оболочек. Она шаг за шагом создаст pod-оболочки версии `1.1.0.RELEASE` и удалит запущенные экземпляры версии `1.0.0.RELEASE`. Замечательной особенностью платформы Kubernetes является то, что она начинает удалять старые pod-оболочки только тогда, когда их замены уже готовы к работе. Готовность определяется с помощью механизма проверки работоспособности `readinessProbe`, описанного ранее в этом разделе. Благодаря этому у вас всегда будут pod-оболочки, готовые к обработке запросов. В итоге, если запуск pod-оболочек пройдет успешно, развертывание перейдет на новую версию.

Но что, если в результате какой-то проблемы pod-оболочки версии `1.1.0.RELEASE` не запустятся? Это может быть вызвано ошибкой в коде, такой как опечатка в имени образа, или отсутствием переменной окружения для нового свойства конфигурации. В этом случае развертывание застопорится. У вас останется два варианта: либо исправить YAML-файл и повторить обновление с помощью команды `kubectl apply -f`, либо откатить развертывание.

Развертывание хранит историю так называемых *выкатываний* (rollout). Каждый раз, когда вы его обновляете, оно создает новое выкатывание. Благодаря этому вы можете легко откатить развертывание до предыдущей версии, выполнив команду:

```
kubectl rollout undo deployment ftgo-restaurant-service
```

Это приведет к тому, что Kubernetes заменит pod-оболочки версии `1.1.0.RELEASE` pod-оболочками версии `1.0.0.RELEASE`.

Развертывания в Kubernetes позволяют доставлять сервисы без простоя. Но что если ошибка проявится уже после того, как pod-оболочка запустится и начнет принимать промышленный трафик? В этом случае Kubernetes продолжит выкатывать новые версии, что будет отражаться на все большем числе пользователей. И хотя ваша система мониторинга, будем надеяться, обнаружит проблему и быстро откатит развертывание, некоторые пользователи все же пострадают. Чтобы этого избежать и сделать новую версию сервиса более надежной, необходимо отделить *развертывание* (запуск сервиса в промышленной среде) от *выпуска* сервиса, в результате которого тот может начать обрабатывать промышленный трафик. Посмотрим, как этого добиться с помощью сети сервисов.

12.4.5. Использование сети сервисов для отделения развертывания от выпуска

Традиционно перед выкатыванием новая версия сервиса тестируется в предпромышленных условиях. Пройдя этот этап, она выкатывается в промышленную среду посредством плавающего обновления, заменяя собой старые экземпляры сервиса. С одной стороны, вы уже могли убедиться в том, что Kubernetes максимально упрощает выполнение плавающих обновлений. Но с другой — этот подход основан на допущении, что новая версия сервиса, пройдя проверку в предпромышленных условиях, будет работать в промышленной среде. К сожалению, так происходит не всегда.

Чаще всего предпромышленные условия не идентичны промышленным. Это связано как минимум с тем, что промышленная среда, скорее всего, имеет куда больший масштаб и обрабатывает намного больше трафика. К тому же на синхронизацию двух сред затрачивается много времени. Из-за этих расхождений некоторые ошибки могут проявиться только в реальных условиях. Но, даже если расхождений нет, вы не можете гарантировать, что тестирование выявит все ошибки.

Выкатывание новых версий можно сделать намного более надежным, отделив развертывание сервиса от его выпуска:

- ❑ *развертывание* — выполнение в промышленной среде;
- ❑ *выпуск сервиса* — открытие доступа к нему конечным пользователям.

Развертывание сервиса в промышленной среде состоит из следующих шагов.

1. Развертывание новой версии в промышленной среде без перенаправления к ней запросов конечных пользователей.
2. Тестирование ее в реальных условиях.
3. Выпуск сервиса для небольшого количества пользователей.
4. Постепенный выпуск сервиса для все более широкой аудитории, пока он не станет обрабатывать весь промышленный трафик.
5. Если на каком-либо этапе появится проблема, можно откатиться к старой версии. Если же вы уверены в том, что все работает как следует, старую версию можно удалить.

В идеале эти шаги следует выполнять в процессе полностью автоматизированного развертывания, который тщательно следит за возникновением ошибок в свежеразвернутом сервисе.

Раньше подобное разделение развертываний и выпусков было затруднительным, так как для его реализации требовалось много усилий. Но при наличии сети сервисов использовать этот стиль развертывания становится намного проще. Как говорилось в главе 11, *сеть сервисов* — это сетевая инфраструктура, через которую сервис общается с другими сервисами и внешними приложениями. Помимо того что она берет на себя некоторые из обязанностей фреймворка микросервисного шасси, сеть сервисов предоставляет балансирование нагрузки и маршрутизацию трафика на основе правил, что позволяет безопасно запускать сразу несколько версий одного и того же сервиса. Позже в этом разделе вы увидите, что тестовых пользователей можно направлять к одной версии, а настоящих — к другой.

Как сообщалось в главе 11, вам доступно несколько реализаций сети сервисов. В этом разделе я покажу, как использовать Istio — популярную сеть сервисов с открытым исходным кодом, созданную компаниями Google, IBM и Lyft. Вначале я сделаю краткий обзор этого проекта и некоторых из его многочисленных возможностей. Затем покажу, как развернуть приложение с помощью Istio. После этого вы увидите, как с помощью механизма маршрутизации этой сети наладить развертывание и выпуск обновлений сервиса.

Краткий обзор сети сервисов Istio

На официальном веб-сайте Istio описывается как «открытая платформа для объединения, администрирования и защиты микросервисов» (<https://istio.io>). Это сетевой слой, через который проходит весь трафик ваших сервисов. У Istio богатый набор возможностей, которые делятся на четыре основные категории:

- ❑ *управление трафиком* — включает в себя обнаружение сервисов, балансирование нагрузки, правила маршрутизации и предохранители;
- ❑ *безопасность* — безопасное межсервисное взаимодействие на основе протокола защиты транспортного уровня (Transport Layer Security, TLS);
- ❑ *телеметрия* — собирает сведения о сетевом трафике и реализует распределенную трассировку;
- ❑ *соблюдение политики* — обеспечивает соблюдение квот и лимитов на частоту запросов.

В этом разделе мы сосредоточимся на возможностях Istio, относящихся к управлению трафиком.

Архитектура Istio (рис. 12.11) состоит из уровня управления и уровня данных. Управляющий уровень реализует такие функции администрирования, как конфигурация уровня данных для маршрутизации трафика. Уровень данных состоит из прокси-серверов Envoy, по одному для каждого экземпляра сервиса.

Два основных компонента управляющего уровня — Pilot и Mixer. *Pilot* извлекает из внутренней инфраструктуры информацию о развернутых сервисах. Например, работая в рамках Kubernetes, этот компонент получает сведения о сервисах и работоспособных pod-оболочках. Он настраивает прокси Envoy для перенаправления трафика согласно заданным правилам маршрутизации. *Mixer* собирает телеметрию из прокси Envoy и обеспечивает соблюдение политики.

Прокси Istio Envoy является модифицированной версией Envoy (www.envoyproxy.io). Это высокопроизводительный прокси-сервер с поддержкой разнообразных протоколов, включая TCP, высокоуровневые и низкоуровневые протоколы, такие как HTTP и HTTPS. Он также понимает протоколы MongoDB, Redis и DynamoDB. Кроме того, Envoy поддерживает надежное межсервисное взаимодействие с такими функциями, как предохранители, ограничение частоты запросов и автоматическое повторение вызовов. Он может защитить коммуникации внутри приложения, задействуя TLS для общения между прокси-серверами Envoy.

Istio использует Envoy в качестве подключаемого модуля (шаблон Sidecar) — процесса или контейнера, который работает в связке с экземпляром сервиса и реализует общую функциональность. В Kubernetes прокси Envoy запускается в виде контейнера внутри pod-оболочки сервиса. В других средах, в которых нет понятия pod-оболочки, Envoy работает в одном контейнере с сервисом. Прокси-сервер Envoy пропускает через себя весь входящий и исходящий трафик сервиса и направляет его согласно правилам маршрутизации, которые предоставляются управляющим уровнем. Например, непосредственное взаимодействие типа «сервис → сервис» приобретает вид «сервис → исходный Envoy → конечный Envoy → сервис».

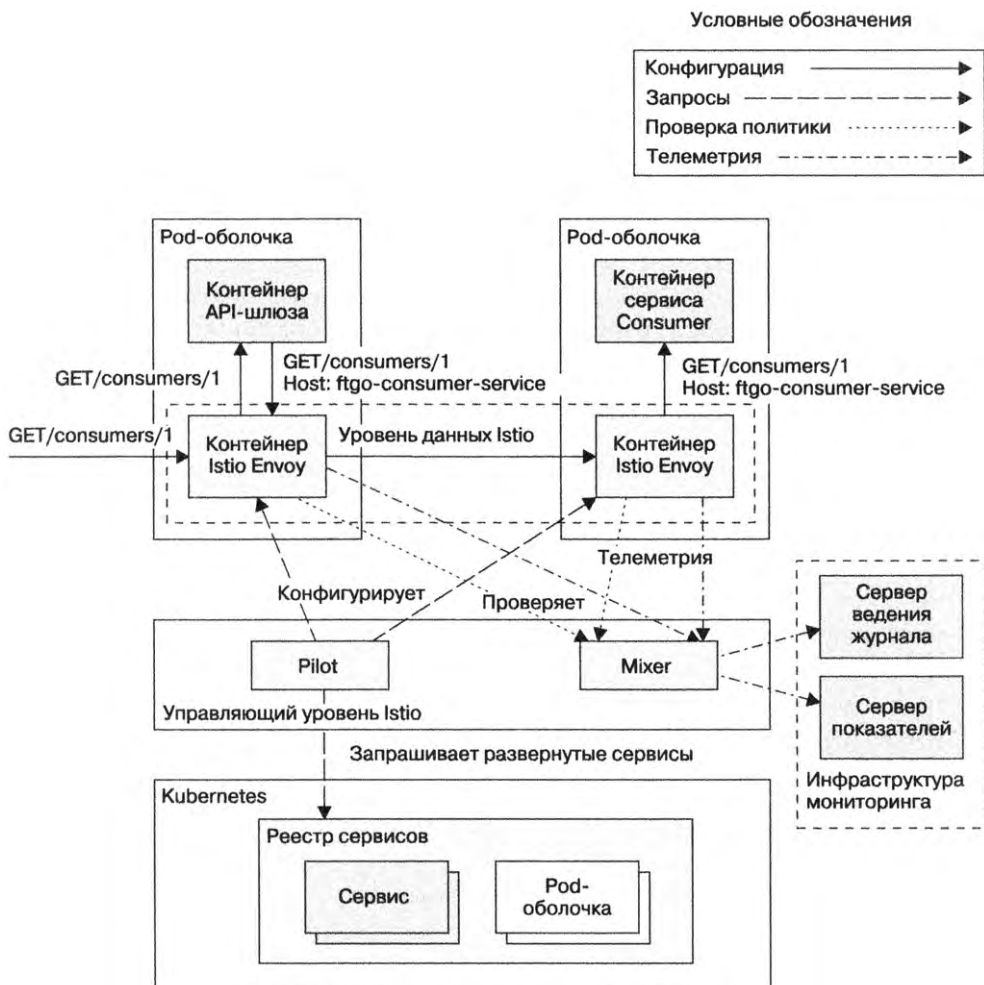


Рис. 12.11. Istio состоит из управляющего уровня, в число компонентов которого входят Pilot и Mixer, и уровня данных, состоящего из прокси-серверов Envoy. Pilot извлекает из внутренней инфраструктуры сведения о развернутых сервисах и конфигурирует уровень данных. Mixer обеспечивает соблюдение политик, таких как квоты, собирает телеметрию и передает ее серверам инфраструктуры мониторинга. Прокси-серверы Envoy маршрутизируют входящий и исходящий трафик сервисов. У каждого экземпляра сервиса есть отдельный прокси-сервер Envoy

Шаблон Sidecar

Реализует общие функции в подключаемом процессе или контейнере, который находится рядом с экземпляром сервиса. См. microservices.io/patterns/deployment/sidecar.html.

Istio настраивается с помощью конфигурационных YAML-файлов в стиле Kubernetes. У этой системы есть утилита командной строки `istioctl`, похожая на `kubectl`. Она используется для создания, обновления и удаления правил и политик. Работая с Istio в Kubernetes, можно задействовать также `kubectl`.

Посмотрим, как развернуть сервис с помощью Istio.

Развертывание сервиса с помощью Istio

Процесс развертывания сервиса в Istio выглядит довольно просто. Для каждого сервиса приложения необходимо определить объекты `Service` и `Deployment`. Пример таких определений в контексте сервиса `Consumer` приведен в листинге 12.7. Они почти полностью совпадают с определениями, которые я показывал ранее, за исключением нескольких деталей. Istio предъявляет определенные требования к сервисам и pod-оболочкам Kubernetes.

- ❑ Порт сервиса Kubernetes должен использовать соглашение об именовании, принятое в Istio и имеющее вид `<протокол>[<суффикс>]`. В качестве протокола можно указать `http`, `http2`, `grpc`, `mongo` или `redis`. Если имя не указано, Istio считает, что это порт TCP, и не применяет правила маршрутизации.
- ❑ Pod-оболочка должна иметь метку `app`, такую как `app: ftgo-consumer-service`, которая идентифицирует сервис. Это требуется для поддержки распределенной трассировки в Istio.
- ❑ Чтобы иметь возможность запускать сразу несколько версий сервиса, название развертывания Kubernetes должно включать в себя версию, например, `ftgo-consumerservice-v1`, `ftgo-consumer-service-v2` и т. д. У pod-оболочки развертывания должна быть метка `version`, такая как `version: v1`. Это позволяет Istio направлять трафик к конкретной версии сервиса.

Листинг 12.7. Развертывание сервиса `Consumer` с помощью Istio

```
apiVersion: v1
kind: Service
metadata:
  name: ftgo-consumer-service
spec:
  ports:
    - name: http ← Именованный порт
      port: 8080
      targetPort: 8080
  selector:
    app: ftgo-consumer-service
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ftgo-consumer-service-v2 ← Версионное развертывание
spec:
  replicas: 1
  template:
    metadata:
```

```

labels:
  app: ftgo-consumer-service ← Рекомендуемые метки
  version: v2
spec:
  containers:
    - image: image: ftgo-consumer-service:v2 ← Версия образа
...

```

Вам, наверное, уже интересно, как запустить прокси-контейнер Envoy в pod-оболочке сервиса. К счастью, Istio делает эту задачу на удивление простой, автоматизируя добавление записи о прокси Envoy в определение pod-оболочки. Это можно сделать двумя способами. Первый способ состоит в *ручном внедрении контейнера* с последующим выполнением команды `istioctl kube-inject`:

```
istioctl kube-inject -f ftgo-consumer-service/src/deployment/kubernetes/ftgo-consumer-service.yml | kubectl apply -f -
```

Эта команда считывает YAML-файл Kubernetes и возвращает модифицированную конфигурацию, содержащую прокси Envoy. Затем полученный результат передается по каналу команде `kubectl apply`.

Второй способ подключения контейнера Envoy к pod-оболочке заключается в *автоматическом внедрении*. Когда эта функция включена, развертывание сервиса производится с помощью команды `kubectl apply`. Kubernetes автоматически вызовет Istio для включения записи о прокси Envoy в определение pod-оболочки.

Если открыть определение pod-оболочки, можно увидеть, что оно не ограничивается контейнером вашего сервиса:

```
$ kubectl describe po ftgo-consumer-service-7db65b6f97-q9jpr
```

```

Name:          ftgo-consumer-service-7db65b6f97-q9jpr
Namespace:     default
...
Init Containers:
  istio-init:   ← Инициализирует pod-оболочку
    Image:      docker.io/istio/proxy_init:0.8.0
    ...
Containers:
  ftgo-consumer-service: ← Контейнер сервиса
    Image:      msapatterns/ftgo-consumer-service:latest
    ...
  istio-proxy:
    Image:      docker.io/istio/proxyv2:0.8.0 ← Контейнер Envoy
...

```

Итак, мы развернули сервис. Теперь поговорим о том, как описать правила маршрутизации.

Создание правил маршрутизации для направления трафика к версии v1

Представьте, что вы развернули объект `ftgo-consumer-service-v2`. В случае отсутствия правил маршрутизации Istio распределяет запросы между всеми версиями сервиса. Таким образом, нагрузка будет ложиться на версии 1 и 2 сервиса

`ftgo-consumer-service`, из-за чего теряется весь смысл использования Istio. Чтобы безопасно выкатить новую версию, вы должны определить правило маршрутизации, которое направляет весь трафик к текущей версии `v1`.

На рис. 12.12 показано правило маршрутизации для сервиса `Consumer`, которое направляет весь трафик к `v1`. Оно состоит из двух объектов Istio — `VirtualService` и `DestinationRule`.

`VirtualService` определяет маршрутизацию запросов к одному или нескольким сетевым узлам. В этом примере указаны маршруты для одного узла — `ftgo-consumer-service`. Далее приводится определение `VirtualService` для сервиса `Consumer`:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ftgo-consumer-service ← Применяется к сервису Consumer
spec:
  hosts:
  - ftgo-consumer-service
  http:
  - route:
    - destination:
        host: ftgo-consumer-service ← Направляет к сервису Consumer
        subset: v1 ← Подмножество v1
```

Этот объект направляет все запросы к подмножеству `v1` pod-оболочек сервиса `Consumer`. Позже я покажу более сложный пример с маршрутизацией на основе HTTP-запросов и балансированием нагрузки между несколькими взвешенными адресатами.

Помимо `VirtualService`, вы должны определить объект `DestinationRule`, который описывает одно или несколько подмножеств pod-оболочек. Подмножеством обычно служит версия сервиса. Также `DestinationRule` может определить политики управления трафиком, такие как алгоритм балансирования нагрузки. Далее приведен пример этого объекта для сервиса `Consumer`:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: ftgo-consumer-service
spec:
  host: ftgo-consumer-service
  subsets:
  - name: v1 ← Имя подмножества
    labels:
      version: v1 ← Селектор для pod-оболочек подмножества
  - name: v2
    labels:
      version: v2
```

`DestinationRule` определяет два подмножества pod-оболочек: `v1` и `v2`. Подмножество `v1` охватывает pod-оболочки с меткой `version: v1`. Pod-оболочки, помеченные как `version: v2`, входят в подмножество `v2`.

После определения этих правил Istio будет направлять трафик только к pod-оболочкам с меткой `version: v1`, благодаря чему мы сможем безопасно развернуть `v1`.

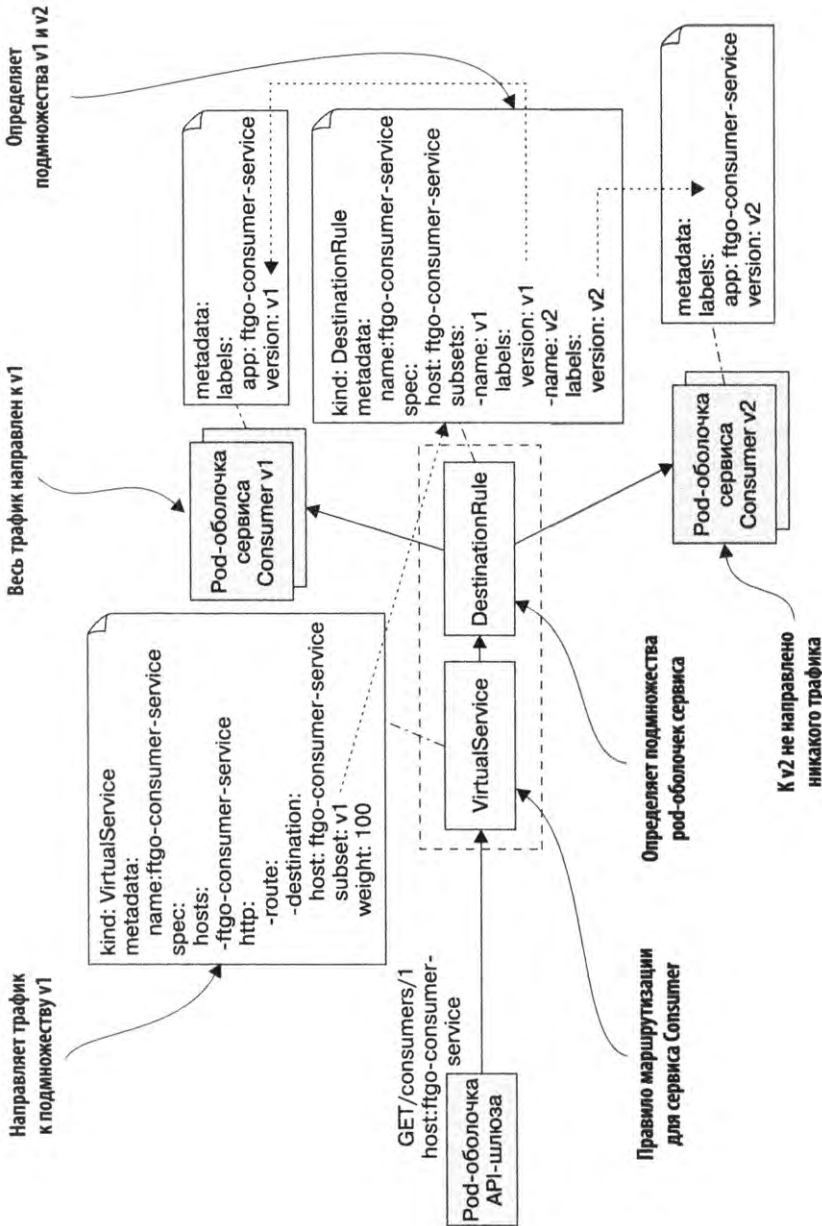


Рис. 12.12. Правило маршрутизации для сервиса Consumer, которое направляет весь трафик к pod-оболочкам версии v1. Оно состоит из объектов VirtualService и DestinationRule, первый направляет трафик к подмножеству v1, а второй включает в подмножество v2 pod-оболочки с меткой version: v1. После определения этого правила вы можете безопасно развернуть новую версию так, чтобы изначально к ней не направлялся никакой трафик

Развертывание версии v2 сервиса Consumer

Далее представлен фрагмент развертывания версии 2 для сервиса Consumer:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ftgo-consumer-service-v2 ← Версия 2
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: ftgo-consumer-service
        version: v2 ← Pod-оболочка с меткой версии
...

```

Это развертывание называется `ftgo-consumer-service-v2`. Оно маркирует свои pod-оболочки `version: v2`. После его создания мы получим две рабочие версии сервиса `ftgo-consumer-service`. Но благодаря правилам маршрутизации Istio не станет направлять трафик к v2. Это позволит вам направить к этой версии тестовые запросы.

Направление тестовых запросов к версии v2

Следующий шаг после развертывания новой версии сервиса — ее тестирование. Предположим, что тестовые запросы от тестовых пользователей содержат заголовок `testuser`. Мы можем сделать так, чтобы сервис `VirtualService` из состава `ftgo-consumer-service` направлял их к экземплярам версии v2. Для этого нужно внести такое изменение:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ftgo-consumer-service
spec:
  hosts:
  - ftgo-consumer-service
  http:
  - match:
    - headers:
        testuser:
          regex: "^.+ $" ← Ищет непустой заголовок testuser
    route:
    - destination:
        host: ftgo-consumer-service
        subset: v2 ← Направляет тестовых пользователей к v2
    - route:
        destination:
          host: ftgo-consumer-service
          subset: v1 ← Направляет всех остальных к v1

```

Помимо исходного маршрута, указанного по умолчанию, `VirtualService` содержит правило маршрутизации, которое направляет запросы с заголовком `testuser` к подмножеству `v2`. После обновления этих правил можно протестировать сервис `Consumer`. Затем, убедившись в том, что версия `v2` готова к работе, вы можете направить к ней часть промышленного трафика. Посмотрим, как это сделать.

Направление промышленного трафика к версии `v2`

Следующим шагом после тестирования свежеразвернутого сервиса будет направление к нему промышленного трафика. Начинать лучше с небольших объемов. Здесь, например, показано правило, которое направляет 95 % запросов к `v1`, а 5 % — к `v2`:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ftgo-consumer-service
spec:
  hosts:
    - ftgo-consumer-service
  http:
    - route:
        - destination:
            host: ftgo-consumer-service
            subset: v1
            weight: 95
        - destination:
            host: ftgo-consumer-service
            subset: v2
            weight: 5
```

Убедившись в том, что сервис способен обрабатывать промышленный трафик, увеличивайте количество запросов, направленных к `pod`-оболочкам версии 2, до тех пор, пока оно не достигнет 100 %. В итоге Istio перестанет слать запросы `pod`-оболочкам версии 1. Можете дать им возможность поработать некоторое время, прежде чем удалять развертывание `v1`.

Позволяя легко разделять развертывание и выпуск сервисов, Istio делает выкатывание новых версий намного более надежным. Но это лишь небольшая часть возможностей, которыми обладает Istio. На момент написания этих строк текущей версией проекта была 0.8. Мне очень нравится наблюдать за тем, как эта и другие сети сервисов становятся стандартным элементом промышленной среды.

12.5. Бессерверное развертывание сервисов

Пакеты, рассчитанные на определенные языки (см. раздел 12.1), виртуальные машины (см. раздел 12.2) и контейнеры (см. раздел 12.3) довольно сильно различаются, но у всех них есть и общее. Во-первых, все эти шаблоны развертывания должны заранее выделять определенные вычислительные ресурсы — физические серверы,

виртуальные машины или контейнеры. Некоторые платформы развертывания поддерживают автомасштабирование, динамически регулируя количество ВМ или контейнеров в зависимости от нагрузки. Тем не менее вам постоянно нужно платить за какие-то ресурсы, даже если они простаивают.

Еще одной общей чертой является то, что ответственность за системное администрирование ложится на вас. С каким бы сервером вы ни работали, его операционную систему нужно обновлять. Если это физические серверы, их необходимо распределять по стойкам. Вы также отвечаете за обслуживание среды выполнения языка программирования. Это то, что компания Amazon называет неразделяемой рутинной работой. С первых дней возникновения компьютерной индустрии системное администрирование было тем, без чего нельзя обойтись. Но, как оказывается, у этого правила есть исключение — бессерверные платформы.

12.5.1. Обзор бессерверного развертывания с помощью AWS Lambda

На конференции AWS Re:Invent 2014 Вернер Вогельс (Werner Vogels), технический директор Amazon, в ходе представления AWS Lambda произнес потрясающую фразу: «На пересечении функций, событий и данных происходит магия». Как можно догадаться из этих слов, платформа AWS Lambda изначально предназначалась для развертывания сервисов с событийной моделью, а «магической» ее делает то, что это пример технологии бессерверного развертывания.

Технологии бессерверного развертывания

Все основные публичные облака предоставляют возможность бессерверного развертывания, а самым продвинутым вариантом является AWS Lambda. В Google Cloud есть Google Cloud Functions, на момент написания книги этот сервис находится на стадии beta-тестирования (cloud.google.com/functions/). В Microsoft Azure есть Azure Functions (azure.microsoft.com/en-us/services/functions/).

Существуют также бессерверные фреймворки с открытым исходным кодом, такие как Apache Openwhisk (openwhisk.apache.org) и Fission for Kubernetes (fission.io), которые можно использовать в собственной инфраструктуре. Однако я не совсем уверен в том, что они окажутся полезными. Вам все равно придется управлять инфраструктурой, на которой эти фреймворки выполняются, — это едва ли можно назвать *бессерверным* подходом. Более того, как вы увидите позже в этом разделе, в обмен на минимизацию системного администрирования бессерверные платформы ограничивают модель программирования. То есть, если вам самим приходится администрировать инфраструктуру, эти ограничения ничем не компенсируются.

AWS Lambda поддерживает Java, NodeJS, C#, GoLang и Python. *Лямбда-функция* — это сервис, лишенный состояния. Для обработки запросов она обычно обращается к сервисам AWS. Например, лямбда-функция, которая срабатывает при

загрузке изображения в облако S3, вставляет элемент в таблицу **IMAGES** DynamoDB и публикует сообщение в Kinesis, чтобы инициировать обработку этого изображения. Лямбда-функция также может вызывать сторонние веб-сервисы.

Чтобы развернуть сервис, нужно упаковать его в файл ZIP или JAR, загрузить в AWS Lambda и указать имя функции, которая будет вызываться для обработки запроса (который еще называют *событием*). AWS Lambda автоматически следит за тем, чтобы экземпляров вашего микросервиса было достаточно для обработки входящих запросов. Вы платите за каждый запрос с учетом потраченного времени и потребленной памяти. Конечно, дьявол кроется в деталях, и позже вы увидите, что AWS Lambda имеет ограничения. Но само то, что ни вы, как разработчик, ни кто-нибудь другой в вашей организации не должны волноваться о каких-либо аспектах серверов, виртуальных машин или контейнеров, чрезвычайно интересно.

Шаблон «Бессерверное развертывание»

Развертывает сервисы с помощью бессерверного механизма, предоставляемого публичным облаком. См. microservices.io/patterns/deployment/serverless-deployment.html.

12.5.2. Написание лямбда-функции

В отличие от предыдущих трех шаблонов лямбда-функции требуют использования особой модели разработки. Их код и формат упаковывания зависят от языка программирования. В Java лямбда-функция представляет собой класс, реализующий обобщенный интерфейс **RequestHandler**, который определен в основной библиотеке AWS Lambda Java (листинг 12.8). Этот интерфейс принимает параметры двух типов: **I** — тип ввода и **O** — тип вывода. Они зависят от того, какого рода запросы обрабатывает лямбда-функция.

Листинг 12.8. В Java лямбда-функция является классом, который реализует интерфейс **RequestHandler**

```
public interface RequestHandler<I, O> {  
    public O handleRequest(I input, Context context);  
}
```

Интерфейс **RequestHandler** определяет единственный метод — **handleRequest()**. У него есть два параметра — входящий объект и контекст, который предоставляет доступ к среде выполнения Lambda, например к ID запроса. В качестве результата возвращается исходящий объект. У лямбда-функций, которые обрабатывают HTTP-запросы, проходящие через API-шлюз AWS, в качестве **I** и **O** используются типы **APIGatewayProxyRequestEvent** и **APIGatewayProxyResponseEvent** соответственно. Вскоре вы увидите, что функции обработки очень напоминают старые добрые сервлеты из Java EE.

В Java лямбда-функции упаковываются в файлы ZIP или JAR. Последние имеют формат uber JAR («толстый JAR») и создаются такими инструментами, как дополнение Maven Shade. ZIP-файлы хранят классы в корневом каталоге, а JAR-зависимости — в папке `lib`. Позже я продемонстрирую создание ZIP-файлов в проекте Gradle. Но сначала рассмотрим разные способы вызова лямбда-функций.

12.5.3. Вызов лямбда-функций

Лямбда-функцию можно вызвать четырьмя способами:

- ☐ с помощью HTTP-запросов;
- ☐ посредством событий, генерируемых сервисами AWS;
- ☐ как запланированные вызовы;
- ☐ напрямую с помощью API-вызовов.

Обработка HTTP-запросов

Вы можете сконфигурировать API-шлюз AWS таким образом, чтобы он направлял HTTP-запросы к вашей лямбда-функции. Она будет доступна по протоколу HTTPS в виде конечной точки. API-шлюз играет роль HTTP-прокси, он передает лямбда-функции объект внутри HTTP-запроса и ожидает получения от нее HTTP-ответа. Использование API-шлюза в сочетании с AWS Lambda позволяет, к примеру, развертывать RESTful-сервисы в виде лямбда-функций.

Обработка событий, сгенерированных сервисами AWS

Лямбда-функцию можно сконфигурировать для обработки событий, генерируемых сервисом AWS. Примеры событий, которые могут привести к срабатыванию лямбда-функции:

- ☐ в бакете S3 создан объект;
- ☐ в таблице DynamoDB создан, обновлен или удален элемент;
- ☐ в потоке Kinesis появилось сообщение, доступное для чтения;
- ☐ с помощью Simple Email Service получено электронное письмо.

Благодаря такой интеграции с сервисами AWS лямбда-функции широко применяются.

Объявление запланированных лямбда-функций

Для вызова лямбда-функции можно также использовать механизм планирования Linux в стиле `cron`. Вы можете сконфигурировать ее для периодических вызовов, например, раз в минуту, три часа или семь дней. Для этого также предусмотрены выражения в формате `cron`, которые определяют, когда платформа AWS должна

вызвать вашу лямбда-функцию. Эти выражения чрезвычайно гибки. Например, вы можете сделать так, чтобы лямбда-функция вызывалась каждый день с понедельника по пятницу в 14:15.

Вызов лямбда-функций с помощью запросов веб-сервисов

Четвертый способ вызова лямбда-функций заключается в использовании веб-сервисов вашего приложения. Веб-сервис указывает в своем запросе имя лямбда-функции и данные входящего события. Ваше приложение может делать синхронные и асинхронные вызовы. В первом случае ответ лямбда-функции будет содержаться в HTTP-ответе веб-сервиса. Если же вызов сделан асинхронно, ответ веб-сервиса сигнализирует о том, был ли успешным запуск лямбда-функции.

12.5.4. Преимущества использования лямбда-функций

Развертывание сервисов в виде лямбда-функций имеет несколько преимуществ.

- ❑ *Интеграция со многими сервисами AWS.* Вы можете с невероятной легкостью писать лямбда-функции, которые потребляют события, сгенерированные такими сервисами AWS, как DynamoDB и Kinesis, и обрабатывают HTTP-запросы через API-шлюз AWS.
- ❑ *Избавление от многих задач системного администрирования.* Вы больше не отвечаете за низкоуровневое системное администрирование. У вас нет операционных систем и сред выполнения, которые нужно обновлять. Благодаря этому вы можете сосредоточиться на развертывании своего приложения.
- ❑ *Эластичность.* AWS Lambda запускает экземпляры вашего приложения в количестве, которого достаточно, чтобы справиться с нагрузкой. Вам не нужно предсказывать необходимую пропускную способность или волноваться о недостаточном или чрезмерном выделении ВМ или контейнеров.
- ❑ *Тарифы, основанные на потреблении.* В отличие от типичных облаков IaaS, в которых вы платите за каждую минуту или час работы ваших ВМ или контейнеров (даже когда они простаивают), AWS Lambda берет плату только за ресурсы, потраченные на обработку каждого запроса.

12.5.5. Недостатки использования лямбда-функций

Как видите, AWS Lambda — чрезвычайно удобный способ развертывания сервисов. Но эта технология не лишена некоторых существенных недостатков и ограничений.

- ❑ *Периодически возникает высокая латентность.* Поскольку в AWS Lambda ваш код выполняется динамически, некоторые запросы будут демонстрировать высокую латентность. Это связано с тем, что AWS нужно время на создание экземпляра приложения и его запуск. Особенно остра эта проблема в сервисах, написанных на Java, поскольку они обычно запускаются как минимум несколько

секунд. Например, лямбда-функция, представленная в следующем разделе, запускается довольно медленно. В связи с этим AWS Lambda может оказаться не лучшим выбором для сервисов, требующих низкой латентности.

- ❑ *Ограниченная модель программирования, основанная на событиях/запросах.* Технология AWS Lambda не предназначена для развертывания длительные время работающих сервисов, которые, к примеру, принимают сообщения от стороннего брокера.

Из-за этих недостатков и ограничений AWS Lambda не всегда является хорошим выбором. И прежде, чем рассматривать альтернативы, я советую проверить, совместимо ли бессерверное развертывание с требованиями вашего сервиса.

12.6. Развертывание RESTful-сервиса с помощью AWS Lambda и AWS Gateway

Давайте посмотрим, как развернуть сервис `Restaurant` с помощью AWS Lambda. У этого сервиса есть REST API для создания и управления ресторанами. Он не использует долгоживущих соединений с Apache Kafka, что делает его хорошим кандидатом для запуска в AWS Lambda. Процесс его развертывания показан на рис. 12.13. Сервис состоит из нескольких лямбда-функций, по одной для каждой конечной точки REST. За направление запросов к этим функциям отвечает API-шлюз AWS.

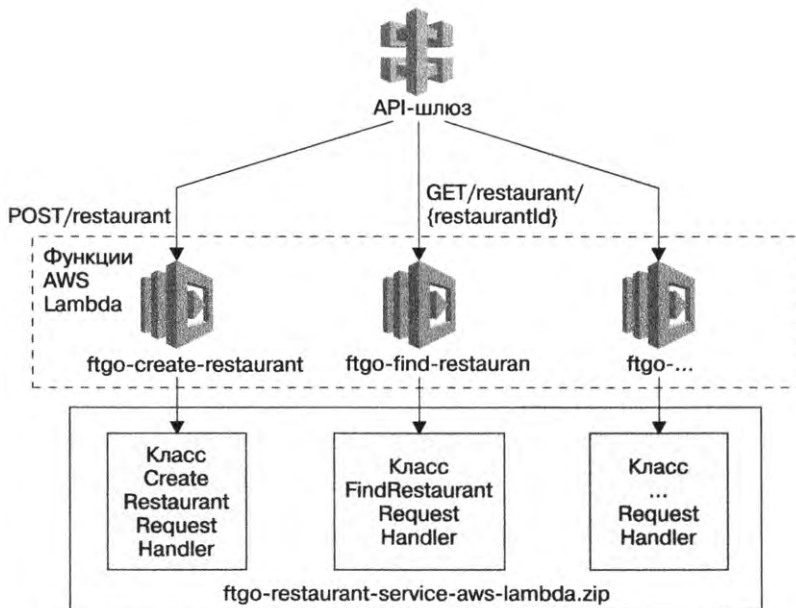


Рис. 12.13. Развертывание сервиса `Restaurant` в виде функций AWS Lambda. API-шлюз AWS направляет HTTP-запросы к лямбда-функциям, которые реализованы классами-обработчиками, определенными в `RestaurantService`

Каждая лямбда-функция содержит класс для обработки запросов. Функция `ftgo-create-restaurant` вызывает класс `CreateRestaurantRequestHandler`, а `ftgo-find-restaurant` — класс `FindRestaurantRequestHandler`. Поскольку эти классы реализуют тесно связанные аспекты одного и того же сервиса, они упаковываются в один ZIP-файл `restaurant-service-aws-lambda.zip`. Рассмотрим архитектуру этого сервиса, включая его классы-обработчики.

12.6.1. Архитектура сервиса Restaurant на основе AWS Lambda

Архитектура, представленная на рис. 12.14, довольно сильно напоминает традиционный сервис. Основное отличие в том, что вместо контроллеров Spring MVC используются классы для обработки запросов из AWS Lambda. Остальная бизнес-логика осталась неизменной.

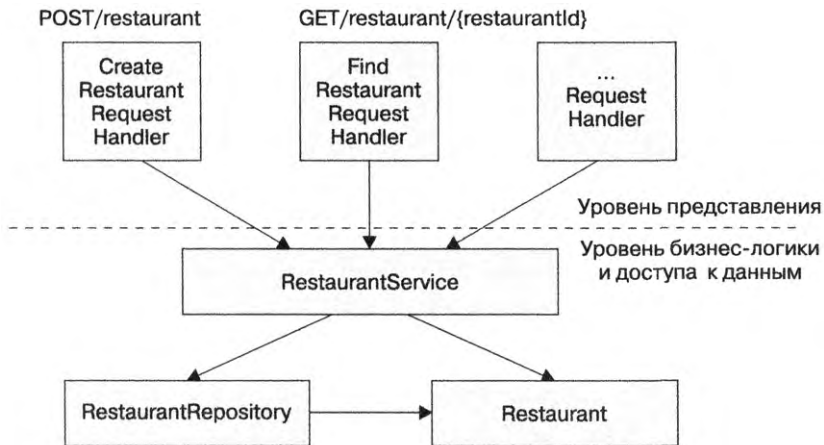


Рис. 12.14. Архитектура сервиса Restaurant, основанного на AWS Lambda. Уровень представления состоит из классов-обработчиков, реализующих лямбда-функции. Они обращаются к уровню бизнес-логики, который написан в традиционном стиле и содержит класс сервиса, сущность и репозиторий

Сервис состоит из уровня представления, в который входят классы-обработчики, вызываемые платформой AWS Lambda для обработки HTTP-запросов, и традиционного уровня бизнес-логики. Последний содержит JPA-сущность `RestaurantService` и слой абстракции для базы данных `RestaurantRepository`.

Рассмотрим класс `FindRestaurantRequestHandler`.

Класс `FindRestaurantRequestHandler`

Класс `FindRestaurantRequestHandler` реализует конечную точку `GET /restaurant/{restaurantId}`. Этот и другие классы-обработчики являются листьями иерархии классов (рис. 12.15). Корнем иерархии служит класс `RequestHandler`, входящий

в состав AWS SDK. Его абстрактные классы обрабатывают ошибки и внедряют зависимости.

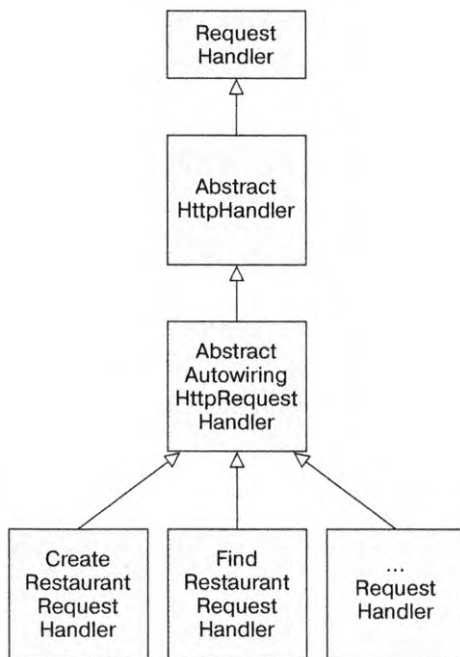


Рис. 12.15. Иерархия классов для обработки запросов. Абстрактные родительские классы реализуют внедрение зависимостей и обработку ошибок

`AbstractHttpHandler` — это базовый абстрактный класс для обработчиков HTTP-запросов. Он перехватывает необработанные исключения, сгенерированные во время обработки запроса, и возвращает ответ вида `500 internal server error`. Класс `AbstractAutowiringHttpRequestHandler` реализует внедрение зависимостей для обработчиков запросов. Я опишу эти абстрактные родительские классы чуть позже, а сначала исследую код `FindRestaurantRequestHandler`.

Код класса `FindRestaurantRequestHandler` показан в листинге 12.9. Он содержит метод `handleHttpRequest()`, который принимает в качестве параметра объект `APIGatewayProxyRequestEvent`, представляющий HTTP-запрос. Он вызывает `RestaurantService`, чтобы найти ресторан, и возвращает `APIGatewayProxyResponseEvent` с описанием HTTP-ответа.

Как видите, это сильно напоминает сервлет, только вместо метода `service()`, который принимает `HttpServletRequest` и возвращает `HttpServletResponse`, этот класс использует метод `handleHttpRequest()`, принимающий `APIGatewayProxyRequestEvent` и возвращающий `APIGatewayProxyResponseEvent`.

Теперь посмотрим на его родительский класс, который реализует внедрение зависимостей.

Листинг 12.9. Класс-обработчик для GET /restaurant/{restaurantId}

```

public class FindRestaurantRequestHandler
    extends AbstractAutowiringHttpRequestHandler {

    @Autowired
    private RestaurantService restaurantService;

    @Override
    protected Class<?> getApplicationContextClass() {
        return CreateRestaurantRequestHandler.class;
    }

    @Override
    protected APIGatewayProxyResponseEvent
        handleHttpRequest(APIGatewayProxyRequestEvent request, Context context) {
        long restaurantId;
        try {
            restaurantId = Long.parseLong(request.getPathParameters()
                .get("restaurantId"));
        } catch (NumberFormatException e) {
            return makeBadRequestResponse(context);
        }

        Optional<Restaurant> possibleRestaurant =
            restaurantService.findById(restaurantId);

        return possibleRestaurant
            .map(this::makeGetRestaurantResponse)
            .orElseGet(() -> makeRestaurantNotFoundResponse(context,
                restaurantId));
    }

    private APIGatewayProxyResponseEvent makeBadRequestResponse(Context context) {
        ...
    }

    private APIGatewayProxyResponseEvent
        makeRestaurantNotFoundResponse(Context context, long restaurantId) { ... }

    private APIGatewayProxyResponseEvent
        makeGetRestaurantResponse(Restaurant restaurant) { ... }
}

```

Конфигурационный класс из состава Spring, который будет играть роль контекста приложения

Если параметр restaurantId отсутствует или недействителен, возвращает 400 — bad request response

Возвращает либо ресторан, либо 404 not found

Внедрение зависимостей с помощью класса AbstractAutowiringHttpRequestHandler

Функции AWS Lambda не являются ни веб-приложениями, ни программами с методом `main()`. Однако было бы досадно, если бы мы не могли воспользоваться возможностями Spring Boot, к которым так привыкли. Класс `AbstractAutowiringHttpRequestHandler`, представленный в листинге 12.10, реализует внедрение зависимостей

для обработчиков запросов. Он создает контекст `ApplicationContext` с помощью `SpringApplication.run()` и автоматически пробрасывает зависимости еще до обработки первого запроса. Дочерние классы, такие как `FindRestaurantRequestHandler`, должны реализовывать метод `getApplicationContextClass()`.

Листинг 12.10. Абстрактный класс `RequestHandler`, реализующий внедрение зависимостей

```
public abstract class AbstractAutowiringHttpRequestHandler
    extends AbstractHttpHandler {

    private static ConfigurableApplicationContext ctx;
    private ReentrantReadWriteLock ctxLock = new ReentrantReadWriteLock();
    private boolean autowired = false;

    protected synchronized ApplicationContext getAppCtx() {
        ctxLock.writeLock().lock();
        try {
            if (ctx == null) {
                ctx = SpringApplication.run(getApplicationContextClass());
            }
            return ctx;
        } finally {
            ctxLock.writeLock().unlock();
        }
    }

    @Override
    protected void
        beforeHandling(APIGatewayProxyRequestEvent request, Context context) {
        super.beforeHandling(request, context);
        if (!autowired) {
            getAppCtx().getAutowireCapableBeanFactory().autowireBean(this);
            autowired = true;
        }
    }

    protected abstract Class<?> getApplicationContextClass();
}
```

Один раз создает контекст приложения Spring Boot

Прежде чем обрабатывать первый запрос, внедряет зависимости в обработчик, используя автопробрасывание

Возвращает класс `@Configuration`, с помощью которого создается `ApplicationContext`

Этот класс переопределяет метод `beforeHandling()` из `AbstractHttpHandler`. Перед обработкой первого запроса его метод `beforeHandling()` внедряет зависимости, используя автоматическое пробрасывание.

Класс `AbstractHttpHandler`

Обработчики запросов для сервиса `Restaurant`, по сути, наследуют класс `AbstractHttpHandler`, показанный в листинге 12.11. Этот класс реализует интерфейсы `RequestHandler<APIGatewayProxyRequestEvent` и `APIGatewayProxyResponseEvent>`. Его основные обязанности — перехват исключений, возникающих во время обработки запросов, и возвращение кода ошибки 500.

Листинг 12.11. Абстрактный класс `RequestHandler`, который перехватывает исключения и возвращает HTTP-ответ с кодом 500

```
public abstract class AbstractHttpHandler implements
    RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    private Logger log = LoggerFactory.getLogger(this.getClass());

    @Override
    public APIGatewayProxyResponseEvent handleRequest(
        APIGatewayProxyRequestEvent input, Context context) {
        log.debug("Got request: {}", input);
        try {
            beforeHandling(input, context);
            return handleHttpRequest(input, context);
        } catch (Exception e) {
            log.error("Error handling request id: {}", context.getAwsRequestId(), e);
            return buildErrorResponse(new AwsLambdaError(
                "Internal Server Error",
                "500",
                context.getAwsRequestId(),
                "Error handling request: " + context.getAwsRequestId() + " "
                + input.toString()));
        }
    }

    protected void beforeHandling(APIGatewayProxyRequestEvent request,
        Context context) {
        // ничего не делать
    }

    protected abstract APIGatewayProxyResponseEvent handleHttpRequest(
        APIGatewayProxyRequestEvent request, Context context);
}
```

12.6.2. Упаковывание сервиса в виде ZIP-файла

Прежде чем развертывать сервис, мы должны упаковать его в ZIP-файл. Легко сделать это с помощью следующего задания для Gradle:

```
task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
        from configurations.runtime
    }
}
```

Это задание собирает архив ZIP с классами и ресурсами на верхнем уровне и JAR-зависимостями в каталоге `lib`.

Имя ZIP-файл, мы можем приступить к развертыванию лямбда-функции.

12.6.3. Развертывание лямбда-функций с помощью бессерверного фреймворка

Развертывание лямбда-функций и настройка API-шлюза могут оказаться довольно утомительными, если ограничиваться лишь инструментами, входящими в состав AWS. К счастью, этот процесс можно существенно упростить, если воспользоваться проектом с открытым исходным кодом под названием Serverless. Вам достаточно написать простой файл `serverless.yml` со списком своих лямбда-функций и их конечных точек RESTful, а Serverless автоматически их развернет и создаст, а также сконфигурирует API-шлюз, чтобы тот направлял к ним запросы.

В листинге 12.12 показан фрагмент файла `serverless.yml`, который развертывает сервис Restaurant в виде лямбда-функций.

Листинг 12.12. `serverless.yml` развертывает сервис Restaurant

```
service: ftgo-application-lambda

provider:
  name: aws
  runtime: java8
  timeout: 35
  region: ${env:AWS_REGION}
  stage: dev
  environment:
    SPRING_DATASOURCE_DRIVER_CLASS_NAME: com.mysql.jdbc.Driver
    SPRING_DATASOURCE_URL: ...
    SPRING_DATASOURCE_USERNAME: ...
    SPRING_DATASOURCE_PASSWORD: ...

package:
  artifact: ftgo-restaurant-service-aws-lambda/build/distributions/
    ftgo-restaurant-service-aws-lambda.zip

functions:
  create-restaurant:
    handler: net.chrisrichardson.ftgo.restaurant.service.lambda
      .CreateRestaurantRequestHandler
    events:
      - http:
          path: restaurants
          method: post
  find-restaurant:
    handler: net.chrisrichardson.ftgo.restaurant.service.lambda
      .FindRestaurantRequestHandler
    events:
      - http:
          path: restaurants/{restaurantId}
          method: get
```

Говорит Serverless,
что развертывать нужно в AWS

Предоставляет сервису
внешнюю конфигурацию
в виде переменных окружения

ZIP-файл
с лямбда-функциями

Определения лямбда-функций,
состоящие из функций
обработки и конечных точек

Вслед за этим можно воспользоваться командой `serverless deploy`, которая считывает файл `serverless.yml`, разворачивает лямбда-функции и конфигурирует API-шлюз AWS. После непродолжительного ожидания ваш сервис станет доступен через URL-адрес конечной точки API-шлюза. Количество экземпляров лямбда-функций сервиса `Restaurant` будет достаточно для того, чтобы справиться с нагрузкой. В случае изменения кода вы легко обновите свои лямбда-функции, пересобрав ZIP-файл и заново выполнив команду `serverless deploy`. И все это без каких-либо серверов!

Инфраструктура развивается впечатляющими темпами. Не так давно мы вручную разворачивали приложения на физических серверах. Сегодня же высокоавтоматизированные публичные облака предоставляют целый ряд вариантов виртуального развертывания. Вы можете запустить свои сервисы в виде виртуальных машин или, что еще лучше, упаковать их в контейнеры и развернуть с помощью многофункциональных фреймворков оркестрации `Docker`, таких как `Kubernetes`. В некоторых случаях сервисы можно развернуть в виде легковесных временных лямбда-функций, полностью забыв об инфраструктуре.

Резюме

- ❑ Вы должны выбрать наиболее легковесный шаблон развертывания, который удовлетворяет требованиям вашего приложения. Варианты следует рассматривать в таком порядке: бессерверные платформы, контейнеры, виртуальные машины и пакеты, рассчитанные на определенные языки.
- ❑ Из-за периодически возникающей высокой латентности и программной модели на основе событий/запросов бессерверное развертывание подходит не для всех сервисов. Но в подходящих сценариях оно становится очень достойным решением, которое позволяет забыть об администрировании операционных систем и сред выполнения. Оно также поддерживает эластичное выделение ресурсов и тарификацию отдельных запросов.
- ❑ Контейнеры `Docker` — это легковесная технология виртуализации на уровне ОС. По сравнению с бессерверным развертыванием они обеспечивают большую гибкость и более предсказуемы в отношении латентности. Для работы с ними лучше всего использовать фреймворк оркестрации `Docker` наподобие `Kubernetes`, который управляет контейнерами в кластере серверов. Недостаток контейнеров заключается в том, что вам придется заниматься администрированием операционных систем и сред выполнения, а также, вероятно, фреймворка оркестрации `Docker` и виртуальных машин, в которых он выполняется.
- ❑ Третий вариант — развертывание сервиса в виде виртуальной машины. С одной стороны, это тяжеловесное решение, поэтому по сравнению со вторым вариантом оно будет более медленным и, скорее всего, ресурсоемким. С другой стороны, современные облака, такие как `Amazon EC2`, высокоавтоматизированы

и предоставляют богатый набор возможностей. Поэтому иногда проще развернуть небольшое приложение с помощью виртуальной машины, чем настраивать фреймворк оркестрации Docker.

- ❑ Обычно стоит избегать развертывания сервисов в виде пакетов, предназначенных для конкретных языков. Исключение представляют собой случаи, когда вы имеете дело с небольшим количеством сервисов. Например, как говорится в главе 13, при переходе на микросервисы вы, скорее всего, будете использовать тот же механизм, что и в монолитной версии приложения (и чаще всего это данный вариант). О подготовке развитой инфраструктуры развертывания следует задуматься только после того, как у вас будут готовы несколько сервисов.
- ❑ Одно из многих преимуществ сети сервисов (сетевого слоя, который пропускает через себя весь входящий и исходящий трафик сервиса) — возможность протестировать свой код в промышленных условиях, прежде чем направлять к нему реальный трафик. Разделение развертывания и выпуска сервисов делает «выкатывание» их новых версий более надежным.

Процесс перехода на микросервисы

В этой главе

- Когда следует переводить монолитное приложение на микросервисную архитектуру.
- Почему перевод монолитного приложения на микросервисы важно выполнять поэтапно.
- Реализация новых возможностей в виде сервисов.
- Извлечение сервисов из монолита.
- Интеграция сервисов и монолита.

Надеюсь, эта книга позволила вам хорошо разобраться в микросервисной архитектуре — ее преимуществах, недостатках и сценариях использования. Тем не менее вы, скорее всего, работаете над большим и сложным монолитным приложением и ежедневно сталкиваетесь с медленным и мучительным процессом разработки и развертывания. А микросервисы при этом кажутся несбыточной мечтой, хотя для вашего приложения они отлично подошли бы. Подобно Мэри с командой разработчиков FTGO, вы, наверное, задаетесь вопросом: как же, черт возьми, перейти на микросервисную архитектуру?

К счастью, существуют стратегии, которые помогут вам выбраться из монолитного ада, не переписывая весь свой код с нуля. Вы постепенно превратите свой монолит в микросервисы, разрабатывая так называемое удушающее приложение (strangler application). Идея этого подхода навеяна фикусом-душителем, растущим в тропических лесах, который оплетает и иногда убивает деревья. *Удушающее*

- ❑ *Предшественник* — предшествующий шаблон, который обосновывает потребность в данном шаблоне. Например, микросервисная архитектура — это предшественник всех остальных шаблонов в языке шаблонов, кроме монолитной архитектуры.
- ❑ *Преемник* — шаблон, который решает проблемы, порожденные данным шаблоном. Например, при использовании микросервисной архитектуры необходимо применить целый ряд шаблонов-преемников, включая обнаружение сервисов и шаблон «Предохранитель».
- ❑ *Альтернатива* — альтернативное решение по отношению к данному шаблону. Например, монолитная и микросервисная архитектуры — это альтернативные способы проектирования приложения. Нужно выбрать одну из них.
- ❑ *Обобщение* — обобщенное решение проблемы. Например, в главе 12 представлены разные реализации шаблона «Один сервис — один сервер».
- ❑ *Специализация* — специализированная разновидность шаблона. Например, в главе 12 вы узнаете, что развертывание сервиса в виде контейнера — это частный случай шаблона «Один сервис — один сервер».

Кроме того, можно группировать шаблоны по областям, в которых их применяют. Явное описание родственных шаблонов помогает получить представление о том, как эффективно решить ту или иную проблему.

Пример визуального представления связей между шаблонами приведен на рис. 1.9.

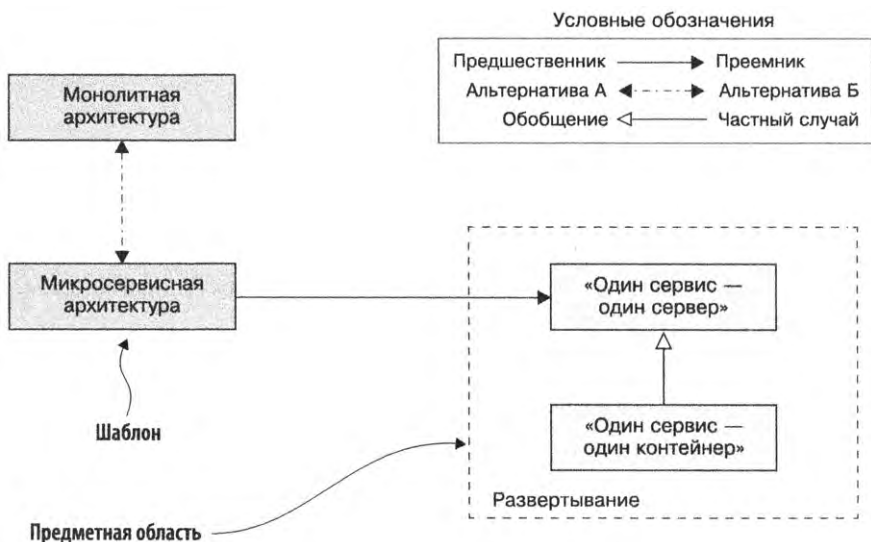


Рис. 1.9. Визуальное представление разного вида связей между шаблонами

намного проще развивать свой стек технологий. Однако переход с монолита на микросервисы — задача не из легких. Он потребует ресурсов, необходимых для разработки новых функций. В итоге руководство, скорее всего, одобрит такой переход только в случае, если это поможет решить существенные бизнес-проблемы.

Если вы попали в монолитный ад, у вас уже наверняка есть как минимум одна проблема на уровне бизнеса. Далее приводятся примеры бизнес-проблем, присущих монолитам.

- ❑ *Медленная доставка.* В приложении сложно разобраться, затрудняются его обслуживание и тестирование, что понижает продуктивность разработчиков. В итоге организация не может эффективно функционировать и рискует уступить конкурентам.
- ❑ *Обновления с ошибками.* Из-за плохой тестируемости новые выпуски ПО часто содержат ошибки. Это может привести к потере недовольных клиентов и снижению прибыли.
- ❑ *Плохая масштабируемость.* Трудность масштабирования монолитного приложения связана с тем, что оно сочетает в одном исполняемом компоненте модули с кардинально разными требованиями к ресурсам. Это означает, что с какого-то момента масштабирование приложения становится либо непомерно дорогим, либо невозможным. В итоге оно не в состоянии удовлетворить текущие или запланированные потребности бизнеса.

Важно убедиться в том, что эти проблемы вызваны незрелой архитектурой, ведь причиной медленного развертывания и низкокачественных обновлений часто становится плохая организация процесса разработки. Например, если вы все еще применяете ручное тестирование, одна лишь его автоматизация может существенно ускорить темп доставки кода. Точно так же проблемы со стабильностью иногда можно решить без изменения архитектуры. Вначале следует попробовать простые решения. И только если они не дали эффекта и вы все равно испытываете трудности с развертыванием программного обеспечения, имеет смысл мигрировать на микросервисную архитектуру. Давайте посмотрим, как это делается.

13.1.2. «Удушье» монолита

Преобразование монолитного приложения в микросервисы — это разновидность модернизации ПО (en.wikipedia.org/wiki/Software_modernization). *Модернизация ПО* — это процесс перевода устаревшего кода на новые архитектуру и стек технологий. Разработчики десятилетиями модернизируют свои приложения. Опыт, накопленный за это время, можно применить для миграции на микросервисную архитектуру. Самый важный урок состоит в том, что не стоит переписывать проект с нуля.

Начать с чистого листа и оставить старую кодовую базу в прошлом — звучит заманчиво. Но это чрезвычайно рискованный подход, который, скорее всего,

закончится неудачей. На дублирование существующей функциональности уйдут месяцы, возможно, годы, а бизнес-нужды требуют реализации новых возможностей уже сегодня! К тому же вам все равно придется заниматься разработкой старого приложения, что будет отвлекать вас от переписывания, из-за этого сроки завершения работы будут постоянно сдвигаться. Что еще хуже, вы вполне можете потратить время на реализацию функций, которые больше не нужны. Мартину Фаулеру приписывают такое высказывание: «Переписывание с нуля гарантирует лишь одно — ноль!» (www.randyshop.com/evolutionary-architecture).

Вместо того чтобы начинать с чистого листа, вы должны постепенно трансформировать свой монолитный проект (рис. 13.1), построив новое *удушающее* приложение. Оно состоит из микросервисов, работающих в связке с монолитным кодом. Со временем монолитное приложение будет реализовывать все меньше и меньше функций, пока полностью не исчезнет или не превратится в еще один микросервис. Эта стратегия похожа на то, как если бы вы пытались ремонтировать свою машину прямо на ходу. Это непросто, но куда менее рискованно, чем попытка переписывания с нуля.

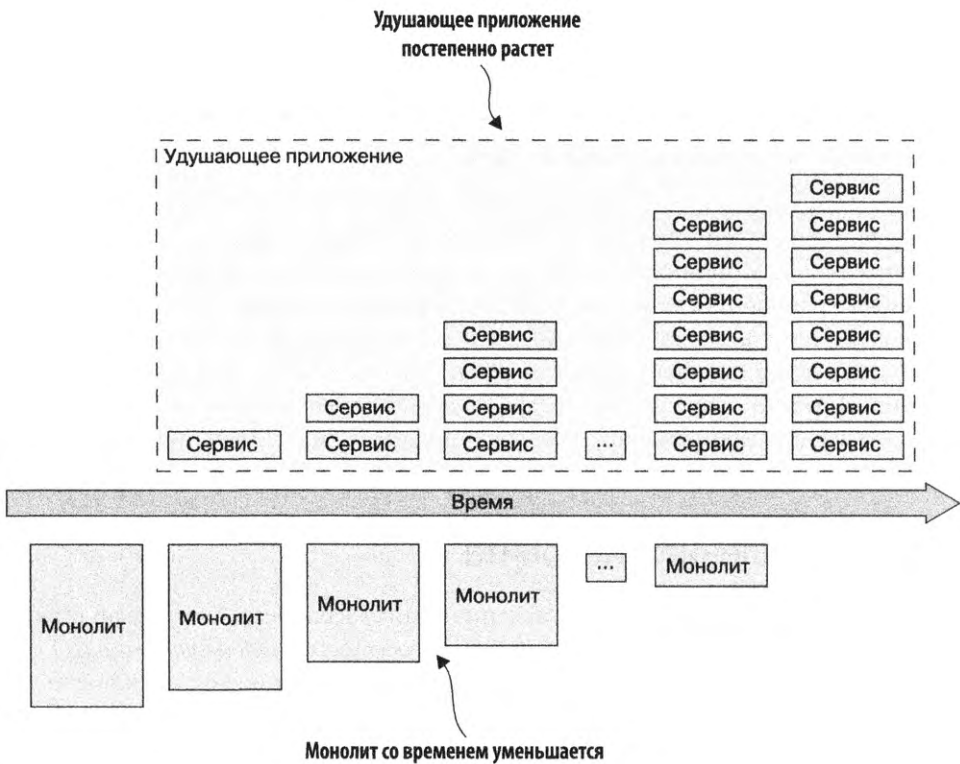


Рис. 13.1. Монолит постепенно заменяется удушающим приложением, состоящим из сервисов. В конце концов монолит полностью исчезает или становится одним из микросервисов

Мартин Фаулер называет эту стратегию модернизации шаблоном удушающего приложения (strangler application, см. www.martinfowler.com/bliki/StranglerApplication.html). Это название навеяно фикусом-душителем (см. ru.wikipedia.org/wiki/Фикусы-душители), обитающим в тропических лесах. Он оплетает дерево, стремясь подняться над пологом леса и достичь солнечных лучей. Дерево часто умирает либо из-за старости, либо в процессе удушения, оставляя после себя древовидный фикус.

Шаблон «Удушающее приложение»

Выполняет модернизацию постепенно возводя новое (удушающее) приложение вокруг старого кода. См. microservices.io/patterns/refactoring/strangler-application.html.

Процесс рефакторинга обычно занимает месяцы и даже годы. Например, если верить Стиву Йерге (Steve Yegge), компания Amazon потратила несколько лет на рефакторинг своего монолита. Если ваша система очень большая, трансформация может так никогда и не завершиться. Например, может настать момент, когда разбиение монолита перестанет быть самой важной задачей и вы переключитесь, скажем, на реализацию функций, которые приносят прибыль. Если монолит не препятствует текущей разработке, возможно, не стоит его трогать.

Демонстрируйте пользу от перехода как можно раньше и чаще

Важное преимущество постепенного перехода на микросервисную архитектуру состоит в том, что вы сразу же видите плоды своей работы. Это сильно отличается от переписывания с нуля, о пользе которого можно судить только по его окончании. При постепенном рефакторинге монолита каждый новый сервис можно писать с помощью нового стека технологий и современного высокоскоростного процесса разработки и развертывания. Благодаря этому ваша команда будет доставлять свой код все быстрее и быстрее.

Кроме того, вы можете начать миграцию с самых значимых участков приложения. Представьте, к примеру, что вы работаете над проектом FTGO и руководство считает алгоритм планирования доставки ключевым конкурентным преимуществом. В этом случае управление доставкой, скорее всего, будет постоянно находиться в процессе разработки. Выделив его в отдельный сервис, вы сможете прикрепить к нему команду, которая будет действовать независимо от своих коллег, что существенно повысит темп разработки. Эта команда сможет чаще выпускать новые версии алгоритма и оценивать их эффективность.

Ранняя демонстрация результатов также помогает заручиться поддержкой руководства в процессе перехода. Это крайне важно, поскольку рефакторинг отнимает ресурсы, которые могли бы быть направлены на разработку новых возможностей.

Некоторые организации испытывают трудности с устранением технической задолженности из-за слишком амбициозных и, как оказывается впоследствии, мало полезных планов. В итоге руководство сложно убедить в необходимости переписывания проекта. Но благодаря тому, что переход на микросервисы проходит постепенно, команда разработки может демонстрировать пользу этой идеи на ранних этапах и с высокой частотой.

Минимизация изменений, вносимых в монолит

В этой главе мы постоянно будем возвращаться к тому, что при переходе на микросервисную архитектуру не стоит вносить масштабные изменения в монолит. Естественно, в процессе миграции вам неизбежно придется что-то менять. В подразделе 13.3.2 говорится о том, что монолит часто нужно модифицировать для участия в повествованиях, которые обеспечивают согласованность данных между ним и сервисами. Такие масштабные изменения отнимают много времени, довольно рискованны и дорогостоящи. В конце концов, это, наверное, основная причина, почему вы решили перейти на микросервисы.

К счастью, существуют стратегии, с помощью которых масштаб необходимых изменений можно уменьшить. Например, в подразделе 13.2.3 показана стратегия копирования информации из извлеченного сервиса обратно в базу данных монолита. А в подразделе 13.3.2 я покажу, как тщательно спланировать извлечение сервиса, чтобы уменьшить его воздействие на монолит. Применяя эти стратегии, вы сможете снизить объем работы, необходимой для рефакторинга монолитного приложения.

Инфраструктура развертывания: не все сразу

В этой книге приводится множество новых блестящих технологий, включая платформы развертывания вроде Kubernetes и AWS Lambda, а также механизмы обнаружения сервисов. У вас может появиться соблазн начать переход на микросервисы с выбора этих технологий и построения инфраструктуры. Возможно даже, что руководство и дружественный поставщик услуг PaaS будут подталкивать вас к покупке подобного рода решений.

Но как бы соблазнительно ни выглядела подготовка инфраструктуры, советую делать как можно меньше предварительных инвестиций в ее построение. Единственное, без чего нельзя обойтись, — это процесс развертывания с автоматическим тестированием. Например, если у вас всего несколько сервисов, вам не нужны развитые инструменты для развертывания и обеспечения наблюдаемости. В самом начале для обнаружения сервисов можно использовать конфигурацию, встроенную в исходный код. Я рекомендую отложить любые решения о технической стороне инфраструктуры, требующие существенных инвестиций, до тех пор, пока вы не накопите реальный опыт работы с микросервисной архитектурой. Только после запуска нескольких сервисов вы сможете выбрать подходящие технологии, хорошо понимая, что делаете.

Теперь рассмотрим стратегии, которые можно применять для миграции на микросервисы.

13.2. Стратегии перехода с монолита на микросервисы

Существует три основные стратегии для удушения монолита и постепенной замены его микросервисами.

1. Реализация новых возможностей в виде сервисов.
2. Разделение уровня представления и внутренних компонентов.
3. Разбиение монолита путем оформления функциональности в виде сервисов.

Первая стратегия предотвращает дальнейшее развитие монолита. Она позволяет быстро продемонстрировать выгоду от использования микросервисов, помогая заручиться поддержкой руководства. Две другие стратегии разбивают монолит на части. Второй подход может стать полезным в процессе рефакторинга, а без третьего вы точно не обойдетесь, поскольку именно так функциональность переносится из монолита в удушающее приложение.

Рассмотрим каждую из этих стратегий, начиная с реализации новых возможностей в виде сервисов.

13.2.1. Реализация новых возможностей в виде сервисов

Закон ямы гласит: «Если вы оказались в яме, перестаньте копать» (en.m.wikipedia.org/wiki/Law_of_holes). Это отличный совет на случай, когда монолитное приложение становится сложно поддерживать. Иными словами, если у вас есть большой и сложный монолитный проект, прекратите добавлять в него новые возможности, иначе он станет еще более крупным и неуправляемым. Вместо этого новые функции следует реализовывать в виде сервисов.

Это отличный способ начать перевод монолитного приложения на микросервисную архитектуру, снизив темпы его развития. Данный подход ускоряет реализацию новых функций, поскольку разработка происходит в совершенно новой кодовой базе. Он также позволяет быстро продемонстрировать выгоды от перехода на микросервисы.

Интеграция нового сервиса с монолитом

Архитектура приложения после реализации новой возможности в виде сервиса показана на рис. 13.2. Помимо нового сервиса и монолита, она содержит два других компонента, которые интегрируют новый код в приложение:

- *API-шлюз* — направляет запросы новой функциональности к новым сервисам, а старые запросы — к монолиту;

- *интеграционный связующий код* — интегрирует сервисы в монолит. Позволяет сервису обращаться к данным и функциям, принадлежащим монолиту.

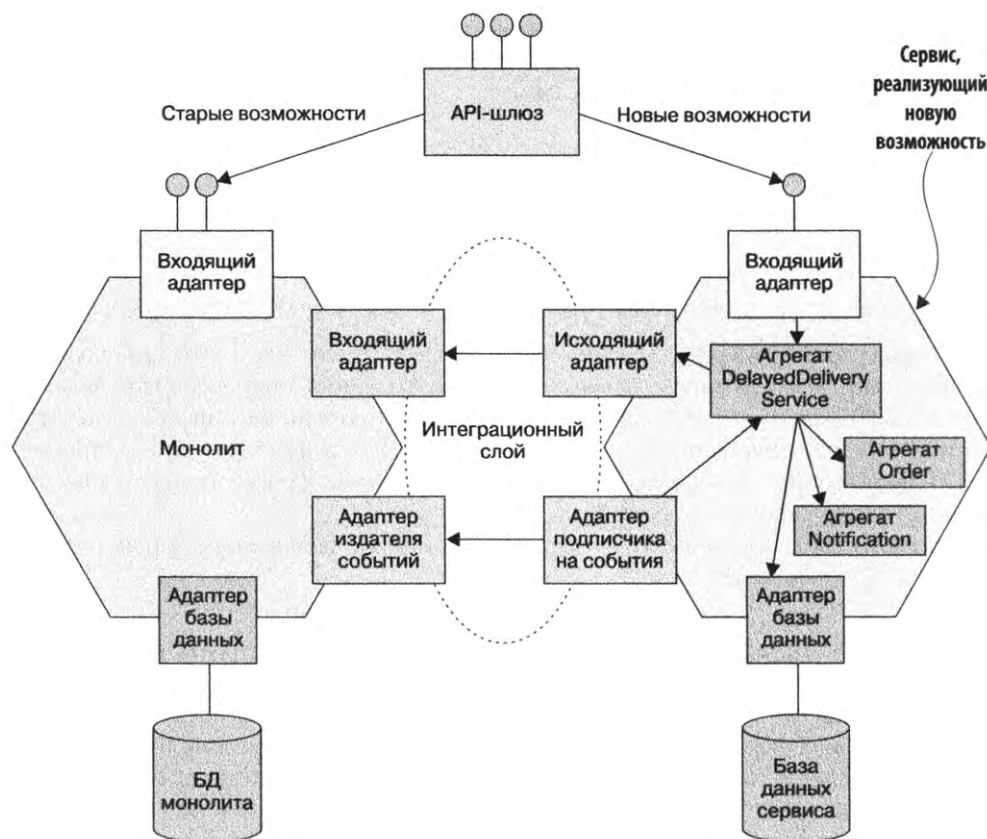


Рис. 13.2. Новая возможность реализуется в виде сервиса, входящего в удушающее приложение. Интеграционный слой связывает сервис с монолитом и состоит из адаптеров, которые реализуют синхронные и асинхронные API. API-шлюз направляет к сервису запросы, которые обращаются к новым функциям

Интеграционный код не является самостоятельным компонентом. Он состоит из адаптеров к монолиту и сервисов, которые применяют один или несколько механизмов межпроцессного взаимодействия. Например, связующий слой для сервиса **Delayed Delivery**, описанного в подразделе 13.4.1, использует как REST, так и доменные события. Сервис извлекает из монолита информацию о контракте клиента, обращаясь к REST API. Монолит публикует доменные события **Order**, чтобы сервис **Delayed Delivery** мог отслеживать состояние заказов и реагировать на несвоевременную доставку. В подразделе 13.3.1 интеграционный код описывается подробнее.

Когда новую функцию следует реализовывать в виде сервиса

В идеале каждая новая возможность должна быть реализована в удушающем приложении, а не в монолите. Для этого создается новый или дополняется уже существующий сервис. Таким образом вы сможете избежать дальнейшей модификации монолитного кода. К сожалению, это не всегда возможно.

Дело в том, что микросервисная архитектура, в сущности, представляет собой набор слабо связанных сервисов, организованных вокруг бизнес-возможностей. Возможность может оказаться слишком мелкой для того, чтобы быть значимым сервисом, и вы просто добавите несколько полей и методов в существующий класс. Или же новая функциональность слишком тесно связана с кодом монолита. Если вы попытаетесь реализовать ее в виде сервиса, это может вылиться в излишнее меж-процессное взаимодействие и, как следствие, ухудшение производительности. У вас могут возникнуть проблемы с согласованностью данных. Возможность, для которой нельзя выделить отдельный сервис, обычно вначале реализуют в монолите. Позже ее можно будет извлечь вместе с другими связанными функциями.

Реализация новых возможностей в виде сервисов ускоряет их разработку. Это хороший способ быстро продемонстрировать преимущества микросервисной архитектуры. К тому же это замедляет темпы развития монолита. Но в конечном итоге вам необходимо разбить монолит на части, используя две другие стратегии. Вы должны перенести функциональность в удушающее приложение, извлекая ее из монолита в сервисы. Вы также можете ускорить темп разработки, разбивая монолит горизонтально. Посмотрим, как это делается.

13.2.2. Разделение уровня представления и внутренних компонентов

Одна из стратегий сокращения монолитного приложения заключается в отделении уровня представления от бизнес-логики и слоя доступа к данным. Типичное промышленное приложение включает следующие слои.

- ❑ *Логика представления* — состоит из модулей, которые обрабатывают HTTP-запросы и генерируют HTML-страницы с пользовательским веб-интерфейсом. В приложениях, обладающих сложным пользовательским интерфейсом, слой представления занимает существенную часть кода.
- ❑ *Бизнес-логика* — состоит из модулей, реализующих бизнес-правила. В промышленных приложениях может быть довольно сложной.
- ❑ *Логика доступа к данным* — состоит из модулей для доступа к инфраструктурным сервисам, таким как базы данных и брокеры сообщений.

Уровень представления обычно четко отделен от бизнес-логики и прослойки для доступа к данным. Бизнес-логика обладает обобщенным API с одним или

несколькими фасадами, которые ее инкапсулируют. Этот API представляет собой естественный шов, вдоль которого монолит можно разделить на два более мелких приложения (рис. 13.3).

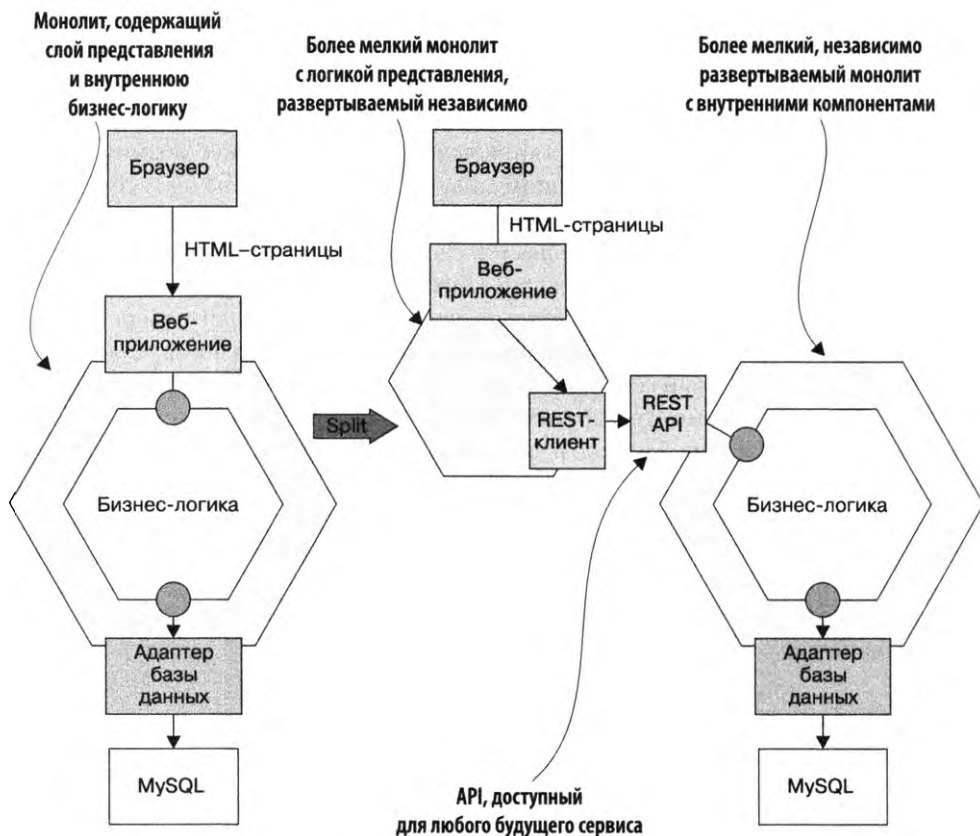


Рис. 13.3. Разделение клиентских и внутренних компонентов сервиса позволяет развертывать их отдельно друг от друга и открывает API для сервисов

Одно приложение может содержать слой представления, а другое — бизнес-логику и прослойку для доступа к данным. После разделения логика представления и бизнес-логика будут общаться посредством удаленных вызовов.

Разделение монолита подобным образом обеспечивает два основных преимущества. Оно позволяет разрабатывать, развертывать и масштабировать два приложения независимо друг от друга. В частности, разработчики слоя представления могут ускорить темпы развития пользовательского интерфейса и, например, легко проводить А/В-тестирование, не развертывая внутреннюю часть. Еще одна положительная сторона этого подхода — открытие доступа к удаленному API, который смогут вызывать будущие микросервисы.

Но это лишь часть решения. Как минимум одно из этих приложений (или оба сразу) почти наверняка будет таким же неуправляемым монолитом. Чтобы заменить монолит сервисами, необходимо применить третью стратегию.

13.2.3. Извлечение бизнес-возможностей в сервисы

Реализация новых возможностей в виде сервисов и отделение клиентского веб-приложения от внутренних компонентов — это лишь полдела. Вам все равно придется активно работать с кодовой базой монолита. Если вы хотите существенно улучшить архитектуру своего приложения и ускорить темпы разработки, разбейте монолит на части, постепенно перенося его бизнес-возможности в сервисы. Например, в разделе 13.5 показано, как вынести управление доставкой из монолита FTGO в новый сервис **Delivery**. При использовании этой стратегии количество бизнес-возможностей, реализованных в виде сервисов, будет постепенно увеличиваться, а монолитный код — понемногу сокращаться.

Для извлечения функций в сервисы необходимо брать вертикальный срез монолита, который состоит из следующих компонентов:

- ☐ входящих адаптеров, реализующих конечные точки API;
- ☐ доменной логики;
- ☐ исходящих адаптеров, таких как логика доступа к БД;
- ☐ схемы базы данных монолита.

Как показано на рис. 13.4, этот код извлекается из монолита и помещается в отдельный сервис. API-шлюз направляет вызовы извлеченных бизнес-возможностей к новому сервису, а остальные запросы оставляет на откуп монолиту. Монолит и сервис взаимодействуют через интеграционный связующий код. Как описывается в подразделе 13.3.1, этот код состоит из размещенных по обе стороны адаптеров, которые используют один или несколько механизмов межпроцессного взаимодействия (IPC).

Извлечение сервисов — непростая задача. Вам нужно решить, как разбить доменную модель монолита на отдельные модели, одна из которых будет принадлежать сервису. Для вынесения функциональности придется разорвать зависимости, такие как объектные ссылки, и, возможно, разделить существующие классы. А также модифицировать структуру базы данных.

Извлечение сервисов обычно занимает много времени, особенно если кодовая база монолита запутанная, что далеко не редкость. В связи с этим вам следует тщательно подумать над тем, какие сервисы нужно извлечь. Необходимо сосредоточиться на рефакторинге тех частей приложения, которые приносят большую пользу. Но прежде всего нужно спросить себя, какую выгоду вы от этого получите.

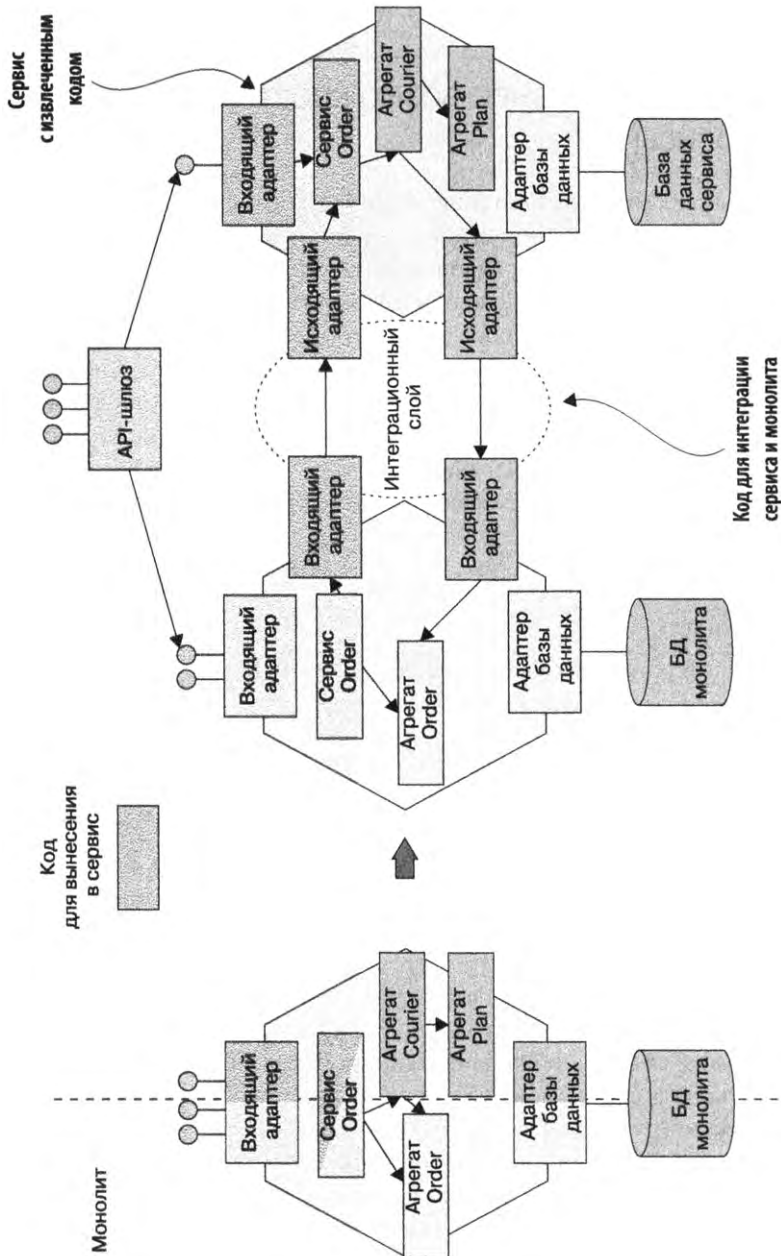


Рис. 13.4. Разбиение монолита путем извлечения сервисов. Вам нужно определить срез функциональности, состоящий из бизнес-логики и адаптеров, и вынести его в сервис. Свежеизвлеченный сервис и монолит взаимодействуют через API, предоставленные связующим слоем

Например, имеет смысл извлечь сервис, функции которого постоянно эволюционируют и имеют критическое значение с точки зрения бизнеса. Если извлечение не приносит существенной выгоды, не стоит тратить на него время. Позже в этом разделе я опишу стратегии для определения того, что и когда следует извлекать. Но сначала подробнее рассмотрим некоторые трудности, с которыми вы можете столкнуться при вынесении кода в сервисы, и то, как с ними справиться.

В ходе извлечения сервисов вы столкнетесь с двумя непростыми задачами:

- ❑ разделением доменной модели;
- ❑ рефакторингом базы данных.

Рассмотрим их.

Разделение доменной модели

Чтобы извлечь сервис, необходимо вычленить его доменную модель из доменной модели монолита. Разделение доменных моделей требует хирургической точности. Одна из проблем, с которой вы столкнетесь, состоит в устранении объектных ссылок, которые могут выйти за границы сервиса. Вполне возможно, что класс, оставшийся в монолите, ссылается на классы, вынесенные в сервис, и наоборот. Представьте, к примеру, ситуацию, изображенную на рис. 13.5: в результате извлечения сервиса `Order` одноименный класс ссылается на класс `Restaurant`, находящийся в монолите. Поскольку сервис обычно работает в отдельном процессе, сохранять объектные ссылки, выходящие за его пределы, попросту нельзя. От этой ссылки нужно как-то избавиться.

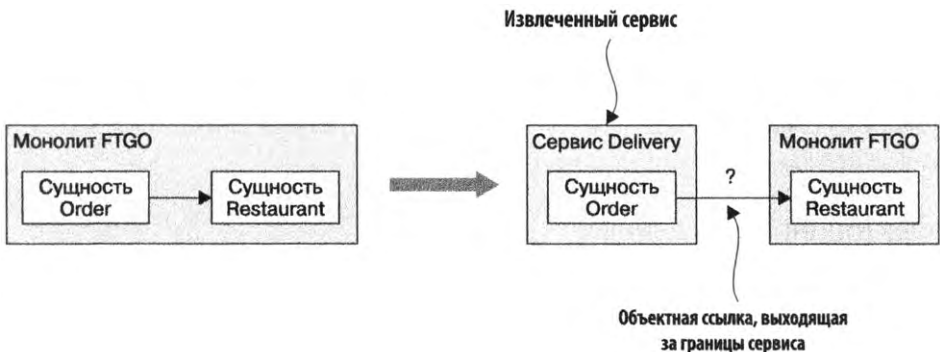


Рис. 13.5. Доменный класс `Order` содержит ссылку на класс `Restaurant`. Если выделить `Order` в отдельный сервис, с этой ссылкой нужно что-то сделать, поскольку между процессами ее быть не должно

Эту проблему можно рассматривать с точки зрения агрегатов DDD, описанных в главе 5. *Агрегаты* ссылаются друг на друга с помощью первичных ключей,

а не объектных ссылок. Представим классы `Order` и `Restaurant` в виде агрегатов (рис. 13.6) и заменим ссылку на `Restaurant` внутри `Order` полем `restaurantId`, которое будет хранить значение первичного ключа.

Одна из проблем с заменой объектных ссылок первичными ключами состоит в том, что, несмотря на свою незначительность в контексте класса, она может серьезно повлиять на его клиентов, которые по-прежнему работают со ссылками. Позже в этом разделе я покажу, как ограничить область изменения путем репликации данных между сервисом и монолитом. Сервис `Delivery`, к примеру, может определить класс `Restaurant`, который будет копией одноименного класса монолита.



Рис. 13.6. Объектную ссылку на `Restaurant` внутри класса `Order` заменяют первичным ключом, чтобы избавиться от объекта, выходящего за пределы процесса

Извлечение сервиса обычно намного сложнее, чем простое вынесение класса целиком. Куда более проблемным аспектом разделения доменной модели является извлечение функциональности из класса, у которого есть другие обязанности. Эта ситуация обычно возникает с так называемыми божественными классами (см. главу 2), которым делегирована слишком большая ответственность. Пример таких классов в приложении FTGO — `Order`. Он реализует несколько бизнес-возможностей, включая управление заказами, доставкой и т. д. В разделе 13.5 вы увидите, каким образом вынесение управления доставкой в отдельный сервис связано с извлечением класса `Delivery` из `Order`. Сущность `Delivery` реализует функции управления доставкой, которые прежде были встроены в класс `Order`.

Рефакторинг базы данных

Разделение доменной модели не ограничивается модификацией кода. Многие ее классы хранят свое содержимое на постоянной основе. Их поля накладываются на схему базы данных. Следовательно, вместе с сервисом из монолита извлекаются еще и данные. Вам придется переместить таблицы из БД монолита в БД сервиса.

Кроме того, при разделении сущности нужно также разделить соответствующую таблицу базы данных и переместить ее часть в сервис. Например, когда управление доставкой выделяется в отдельный сервис, разделяется сущность `Order` и извлекается сущность `Delivery`. На уровне базы данных разделяется таблица `ORDERS`, часть которой оформляется в виде новой таблицы `DELIVERY`. Затем `DELIVERY` перемещается в сервис.

В книге Скотта Амблера (Scott W. Ambler) и Прамодкумара Дж. Садаладжа (Pramodkumar J. Sadalage) *Refactoring Databases* (Addison Wesley, 2011)¹ описывается ряд методик рефакторинга схемы базы данных. Например, там можно найти рефакторинг типа «разделение таблицы» (split table), который разбивает одну таблицу на две или больше. Многие методики, рассмотренные в этой книге, могут пригодиться при извлечении сервисов из монолита. В качестве примера можно привести идею репликации данных для постепенного перевода клиентов БД на новую схему. Мы можем применить этот подход для ограничения области изменений, которые необходимо внести в монолит в ходе извлечения сервисов.

Репликация данных для ограничения масштаба изменений

Как уже упоминалось, извлечение сервиса требует изменения доменной модели монолита. Например, вам нужно заменить объектные ссылки первичными ключами и разделить классы. Подобного рода модификации могут пронизывать кодовую базу, требуя внесения широкомасштабных изменений в монолит. Скажем, если вы разделите сущность **Order** и извлечете сущность **Delivery**, вам придется исправлять каждый участок кода, который ссылается на перемещенные поля. Подобного рода правки могут занять чрезвычайно много времени и стать существенной преградой на пути разбиения монолита.

Такие затратные изменения можно отложить и даже сделать их необязательными, если воспользоваться методикой, аналог которой описан в упомянутой книге. Рефакторинг базы данных существенно затрудняется из-за того, что всех ее клиентов следует перевести на новую схему. Предложенное решение состоит в том, чтобы сохранить исходную схему на время переходного периода и использовать триггеры для ее синхронизации с новыми схемами. При этом клиенты постепенно мигрируют со старой схемы на новую.

Аналогичный подход можно применить для извлечения сервисов из монолита. Например, при извлечении сущности **Delivery** сущность **Order** временно остается почти неизменной. Поля, относящиеся к доставке, остаются доступными только для чтения и поддерживаются в актуальном состоянии за счет копирования данных из сервиса **Delivery** обратно в монолит (рис. 13.7). В итоге нужно лишь найти в коде монолита участки, которые обновляют эти поля, и сделать так, чтобы они обращались к новому сервису **Delivery**.

Сохранение структуры сущности **Order** путем репликации данных из сервиса **Delivery** существенно уменьшает объем начальных работ. Со временем мы можем перенести в сервис **Delivery** код, который использует поля сущности **Order** или столбцы таблицы **ORDERS**, относящиеся к доставке. Более того, вполне возможно, что нам никогда не придется вносить эти изменения в монолит. Другие сервисы могут обращаться к коду, извлеченному в сервис **Delivery**.

¹ Амблер С., Садаладж П. Рефакторинг баз данных. — М.: Вильямс, 2016.

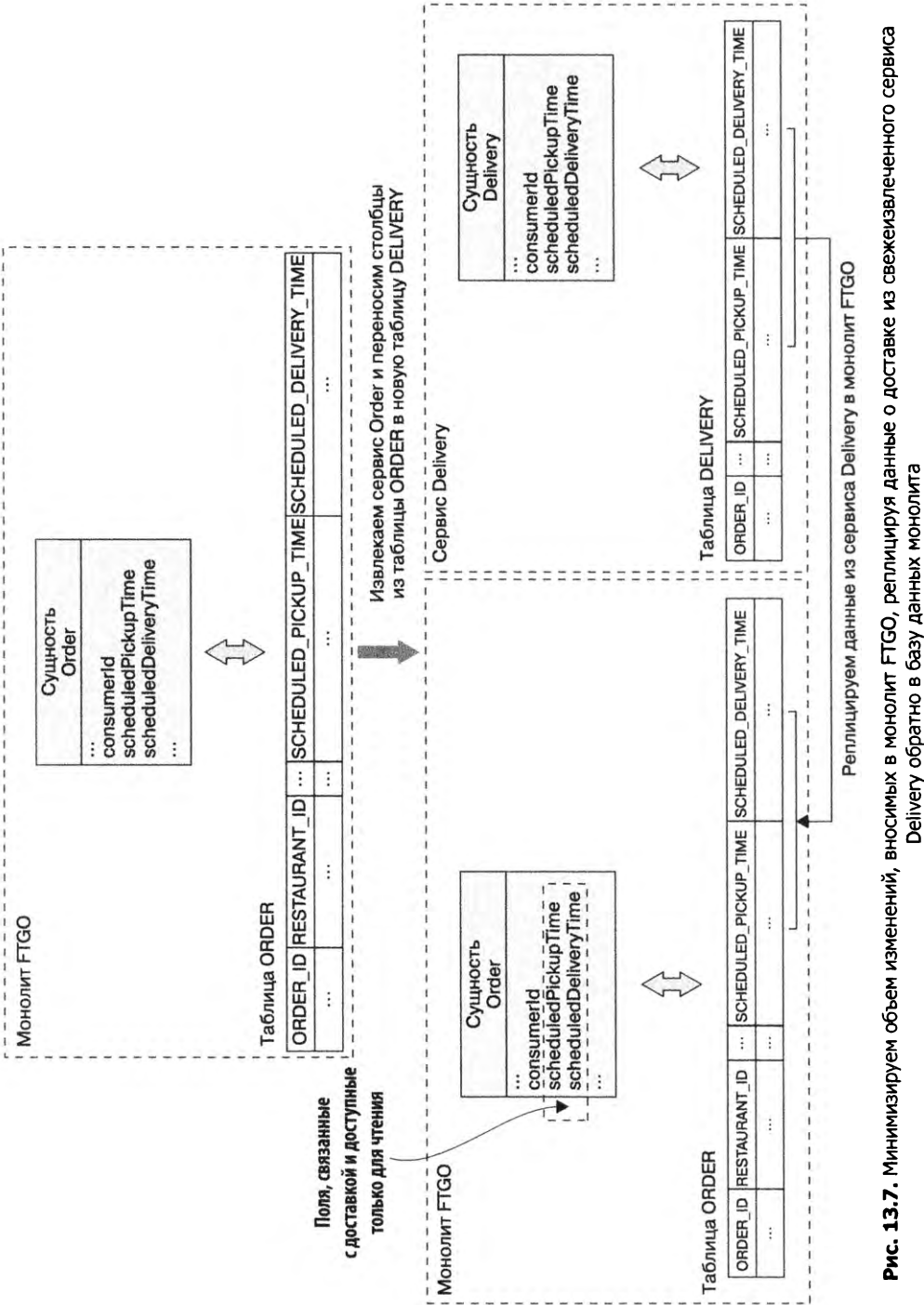


Рис. 13.7. Минимизируем объем изменений, вносимых в монолит FTGO, реплицируя данные о доставке из свежеекзвлеченного сервиса Delivery обратно в базу данных монолита

Какие сервисы и в какой момент нужно извлекать

Как я уже упоминал, разбиение монолита занимает много времени и отвлекает от реализации новых возможностей. В связи с этим следует тщательно продумать последовательность извлечения сервисов. Приоритет должны иметь сервисы, которые приносят наибольшую выгоду. Кроме того, вы постоянно должны демонстрировать руководству пользу, которую приносит миграция на микросервисную архитектуру.

В этом деле важно знать, куда мы движемся. Переход на микросервисы стоит начать с проектирования. Вы должны собраться и на протяжении короткого времени — скажем, двух недель — направить все свои усилия на создание идеальной архитектуры и определение набора сервисов. Это даст вам цель, к которой можно стремиться, но помните, что архитектура не высекается в камне. По мере разбиения монолита и приобретения опыта вы должны пересматривать то, что спроектировали, с учетом приобретенных знаний.

Вслед за определением цели можно приступить к разбиению монолита на части. Существует несколько стратегий, с помощью которых можно определить последовательность извлечения сервисов.

Первая стратегия состоит в фактической заморозке работы над монолитом и извлечении сервисов по мере необходимости. Вместо того чтобы реализовывать новые возможности или исправлять ошибки в монолите, вы извлекаете нужный (-ые) сервис (-ы) и делаете это в нем (них). Одно из преимуществ этого подхода — то, что он заставляет вас разбивать монолит. Его недостаток заключается в том, что мотивацией для извлечения сервисов служат краткосрочные требования, а не долгосрочные потребности. Например, сервисы извлекаются даже при внесении малейшего изменения в относительно стабильную часть системы. В итоге вы рискуете потратить много усилий с минимальной отдачей.

Альтернативная стратегия больше заботится о планировании: модулям приложения назначается рейтинг в соответствии с тем, какую пользу вы ожидаете получить от их извлечения. Есть три причины, почему извлечение сервиса может быть полезным.

- ❑ *Ускорение разработки.* Если согласно плану какая-то часть вашего приложения будет активно развиваться на протяжении следующего года, разработку можно ускорить путем извлечения ее в сервис.
- ❑ *Решение проблем с производительностью, масштабируемостью и надежностью.* Если определенная часть вашего приложения ненадежна или имеет проблемы с производительностью или масштабируемостью, будет полезно преобразовать ее в сервис.
- ❑ *Возможность извлечь какие-то другие сервисы.* Иногда из-за зависимостей между модулями извлечение одного сервиса упрощает извлечение другого.

Вы можете использовать эти критерии для определения приоритета задач рефакторинга согласно предполагаемой пользе. Преимущество данного подхода заключается в его стратегичности и более тесной связи с потребностями бизнеса. В ходе интенсивного планирования следует решить, что более выгодно — реализовывать новые возможности или извлекать сервисы.

13.3. Проектирование взаимодействия между сервисом и монолитом

Сервисы редко являются автономными. Обычно им приходится взаимодействовать с монолитом. Иногда сервису нужно обратиться к данным монолита или вызвать его операции. Например, сервис *Delayed Delivery*, описанный в разделе 13.4.1, должен получить доступ к информации о заказах и клиентах, принадлежащей монолиту. Монолиту тоже могут понадобиться данные или операции сервиса. Например, в разделе 13.5, где обсуждается извлечение управления заказами в сервис, вы увидите, что монолиту необходимо обратиться к сервису *Delivery*.

Одним из важных моментов является поддержание согласованности данных между сервисом и монолитом. В частности, при извлечении сервиса вы разделяете то, что прежде было ACID-транзакциями. Вы должны внимательно следить за сохранением согласованности. Как будет показано позже в этом разделе, для этого иногда применяются повествования.

Ранее уже упоминалось, что взаимодействие между сервисом и монолитом осуществляется с помощью связующего кода. Структура такой интеграционной прослойки показана на рис. 13.8. Она состоит из адаптеров, размещенных в сервисе и монолите, которые общаются между собой с использованием некоего механизма IPC. В зависимости от требований этот механизм может быть основан на REST или обмене сообщениями. Кроме того, механизмов IPC может быть несколько.

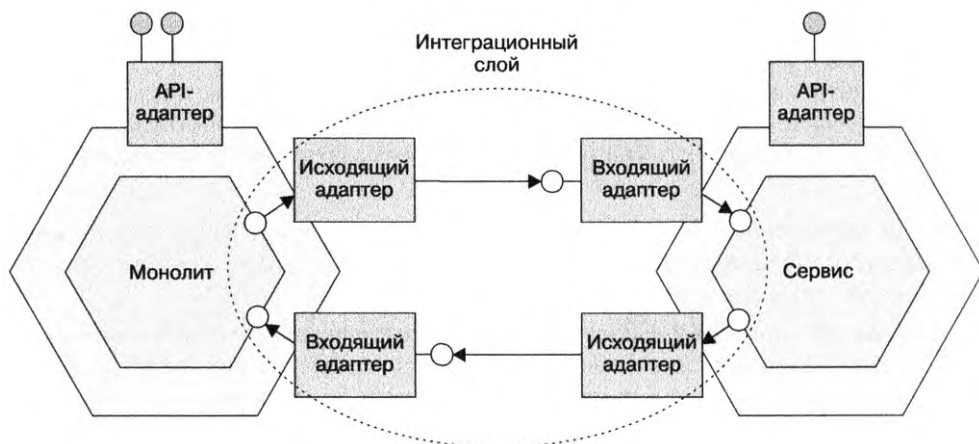


Рис. 13.8. При переходе с монолита на микросервисы обеим сторонам, сервисам и монолиту, часто нужно обращаться к данным друг друга. Это взаимодействие выполняется через интеграционный слой, который состоит из адаптеров, реализующих API. Некоторые интерфейсы основаны на обмене сообщениями, другие — на RPI

Например, сервис *Delayed Delivery* использует как REST, так и доменные события. Контактную информацию о клиентах он извлекает из монолита с помощью

REST, а для отслеживания состояния заказов подписывается на доменные события, публикуемые монолитом.

Я начну этот раздел с описания структуры интеграционного слоя. Мы поговорим о проблемах, которые он решает, и рассмотрим разные варианты его реализации. После этого будут представлены стратегии управления транзакциями, включая использование повествований. Вы увидите, что иногда требование к согласованности данных влияет на порядок извлечения сервисов.

Итак, рассмотрим структуру интеграционного слоя.

13.3.1. Проектирование интеграционного слоя

При реализации новой или извлечении существующей функции в виде сервиса вы должны разработать интеграционный слой, который позволит этому сервису взаимодействовать с монолитом. Его код будет находиться как в сервисе, так и в монолите и использовать некий механизм IPC. Тип выбранного механизма определяет структуру интеграционного слоя. Если, к примеру, сервис обращается к монолиту с помощью REST, этот слой состоит из REST-клиента в сервисе и веб-контроллеров в монолите. Если же монолит подписывается на доменные события, публикуемые сервисом, то интеграционный слой представляет собой адаптер, публикующий события (внутри сервиса), и обработчики событий (внутри монолита).

Проектирование API интеграционного слоя

Проектирование интеграционного слоя нужно начинать с определения того, какие API он будет предоставлять доменной логике. Существует несколько стилей интерфейсов, их выбор зависит от того, что вы делаете с данными — запрашиваете или обновляете. Представьте, что вы работаете над сервисом **Delayed Delivery**, которому необходимо извлекать из монолита контактную информацию клиентов. Бизнес-логике сервиса не нужно знать о механизме IPC, с помощью которого монолит получает эту информацию. Таким образом, механизм следует инкапсулировать в виде интерфейса. Поскольку сервис **Delayed Delivery** запрашивает данные, имеет смысл определить интерфейс **CustomerContactInfoRepository**:

```
interface CustomerContactInfoRepository {  
    CustomerContactInfo findCustomerContactInfo(long customerId)  
}
```

Бизнес-логика сервиса может обращаться к API, не зная при этом, как интеграционный слой извлекает данные.

Теперь рассмотрим другой сервис. Представьте, что вы извлекаете управление доставкой из монолита **FTGO**. Для планирования, переноса и отмены доставки монолиту теперь нужно обращаться к сервису **Delivery**. И снова бизнес-логике неважны подробности реализации внутреннего механизма IPC, поэтому их лучше инкапсулировать в виде интерфейса. В этом сценарии монолит должен вызывать

операции сервиса, поэтому использование репозитория здесь не подходит. Интерфейс сервиса лучше определить следующим образом:

```
interface DeliveryService {
    void scheduleDelivery(...);
    void rescheduleDelivery(...);
    void cancelDelivery(...);
}
```

Бизнес-логика обращается к этому интерфейсу, не зная, каким образом он реализован в интеграционном слое.

Разобравшись с проектированием интерфейсов, можем поговорить о стилях взаимодействия и механизмах IPC.

Выбор стиля взаимодействия и механизма IPC

Важное архитектурное решение при проектировании интеграционного слоя — выбор стилей взаимодействия и механизмов IPC, которые позволят сервису и монолиту общаться. Как говорилось в главе 3, в вашем распоряжении несколько таких стилей и механизмов. Какие из них использовать, зависит от того, что именно требуется одной *стороне* (сервису или монолиту) для запрашивания или обновления другой.

Если одной стороне необходимо запросить данные, принадлежащие другой, у вас есть несколько вариантов. Первый вариант (рис. 13.9) состоит в том, чтобы адаптер, реализующий интерфейс репозитория, обращался к API провайдера данных. Этот API обычно основан на взаимодействии вида «запрос/ответ», таком как REST или gRPC. Например, сервис **Delayed Delivery** может извлекать контактную информацию клиентов через интерфейс REST API, реализованный монолитом FTGO.

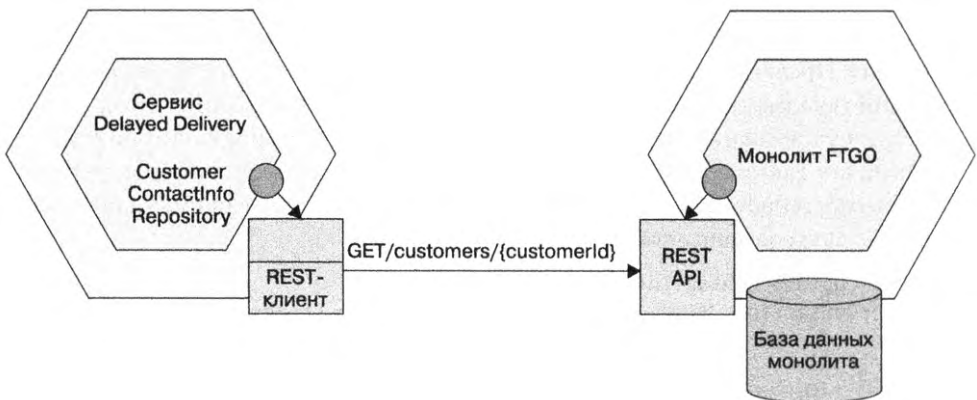


Рис. 13.9. Адаптер, реализующий интерфейс `CustomerContactInfoRepository`, обращается к REST API монолита, чтобы извлечь информацию о клиенте

В этом примере доменная логика сервиса **Delayed Delivery** извлекает контактную информацию клиента, обращаясь к интерфейсу `CustomerContactInfoRepository`. Реализация этого интерфейса вызывает REST API монолита.

Важным преимуществом запрашивания данных через API является его простота, а основным недостатком — потенциальная неэффективность. Потребителю, возможно, понадобится выполнить большое количество запросов, а провайдер может вернуть большой объем данных. Еще одна проблема связана с ухудшением доступности, поскольку мы используем синхронный механизм IPC. В итоге API для выполнения запросов может оказаться нецелесообразным.

Альтернативный подход состоит в том, чтобы потребитель хранил у себя реплику данных (рис. 13.10). Эта реплика, в сущности, является CQRS-представлением. Для поддержания ее в актуальном состоянии потребитель данных подписывается на доменные события, публикуемые провайдером.

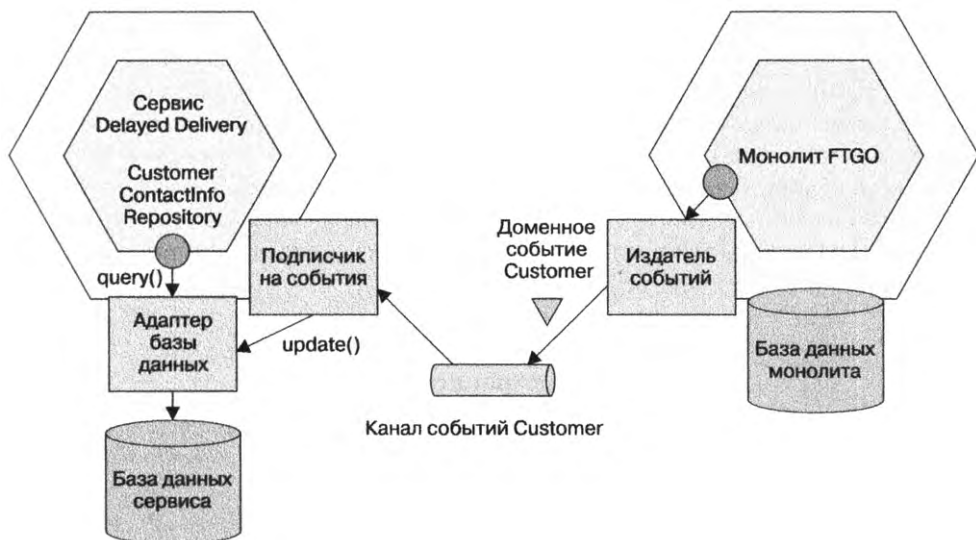


Рис. 13.10. Интеграционный слой реплицирует данные из монолита в сервис. Монолит публикует доменные события, а их обработчик, реализованный в сервисе, обновляет базу данных с репликой

Использование реплики обеспечивает несколько преимуществ. Нам больше не нужно раз за разом запрашивать данные у провайдера. Вместо этого, как было показано при описании CQRS-представлений, реплику можно спроектировать для поддержки эффективных запросов. Один из недостатков этого подхода — сложность обслуживания реплики. Есть также потенциальная трудность, связанная с необходимостью изменения монолита для публикации доменных событий (об этом чуть позже).

Итак, мы обсудили, как делать запросы. Теперь рассмотрим обновления. Трудность выполнения обновлений состоит в том, что необходимо поддерживать согласованность данных между сервисом и монолитом. Запрашивающая сторона уже обновила или должна обновить свою базу данных. Поэтому важно сделать так, чтобы были выполнены оба обновления. Для этого сервис и монолит могут общаться с помощью механизма транзакционных сообщений, реализованного таким фреймворком,

как Eventuate Tram. В простых случаях запрашивающая сторона может послать уведомление или опубликовать событие, чтобы инициировать обновление. Но в более сложных сценариях, чтобы добиться согласованности данных, она должна использовать повествования. Последствия применения повествований описываются в подразделе 13.3.2.

Реализация предохранительного слоя

Представьте, что вы реализуете новую возможность в виде совершенно нового сервиса. Вы не ограничены кодовой базой монолита, поэтому можете использовать современные методики разработки, такие как DDD, и создать новую доменную модель, свободную от прежних недостатков. К тому же определение проблемной области монолита FTGO нечеткое и слегка устаревшее, поэтому вы, скорее всего, смоделируете все немного иначе. В итоге в доменной модели вашего сервиса будут другие имена классов, названия полей и их значения. Например, сервис **Delayed Delivery** содержит сущность **Delivery** с узким набором обязанностей, тогда как у монолита FTGO есть сущность **Order**, на которую возложена чрезмерная ответственность. Из-за такой разницы в доменных моделях вы должны реализовать то, что в терминологии DDD называется предохранительным слоем (anti-corruption layer, ACL). Это позволит сервису общаться с монолитом.

Шаблон «Предохранительный слой»

Программный слой, выступающий посредником между двумя доменными моделями, не давая им засорить друг друга своими концепциями. См. microservices.io/patterns/refactoring/anti-corruption-layer.html.

Цель ACL состоит в том, чтобы не дать устаревшей доменной модели монолита засорить доменную модель сервиса. Это слой кода, который посредничает между разными доменными моделями. Например, у сервиса **Delayed Delivery** есть интерфейс **CustomerContactInfoRepository** с методом **findCustomerContactInfo()**, который возвращает **CustomerContactInfo** (рис. 13.11). Класс, реализующий этот интерфейс, должен понимать языки для общения монолита FTGO с сервисом **Delayed Delivery**.

Реализация метода **findCustomerContactInfo()** обращается к монолиту, чтобы получить контактную информацию клиента, и преобразует ответ в объект **CustomerContactInfo**. В этом примере преобразование получилось довольно простым, но в других ситуациях оно может оказаться сложнее и включать в себя связывание значений, таких как коды состояния.

Подписчик, потребляющий доменные события, тоже содержит слой ACL. Доменные события являются частью доменной модели издателя. Обработчик должен привести их к доменной модели подписчика. Например, монолит FTGO публикует доменные события **Order**, а сервис **Delivery** содержит обработчик, который на них подписывается (рис. 13.12).

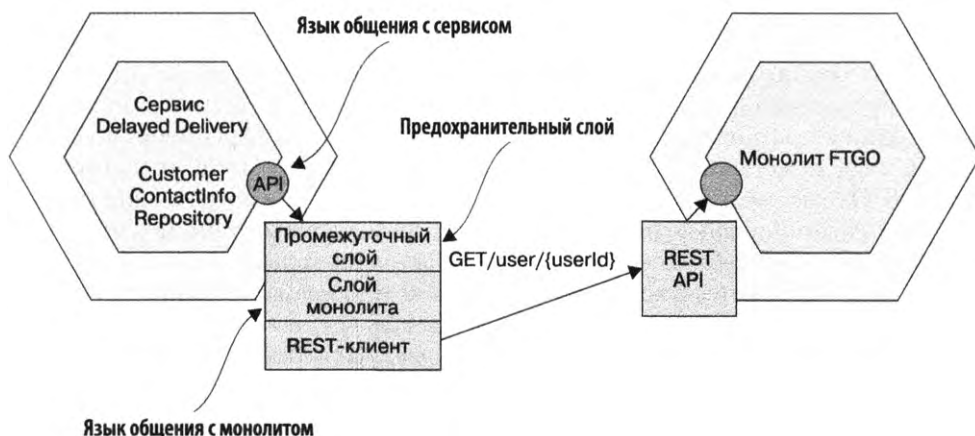


Рис. 13.11. Адаптер сервиса, который обращается к монолиту, должен выполнять преобразования между доменными моделями сервиса и монолита

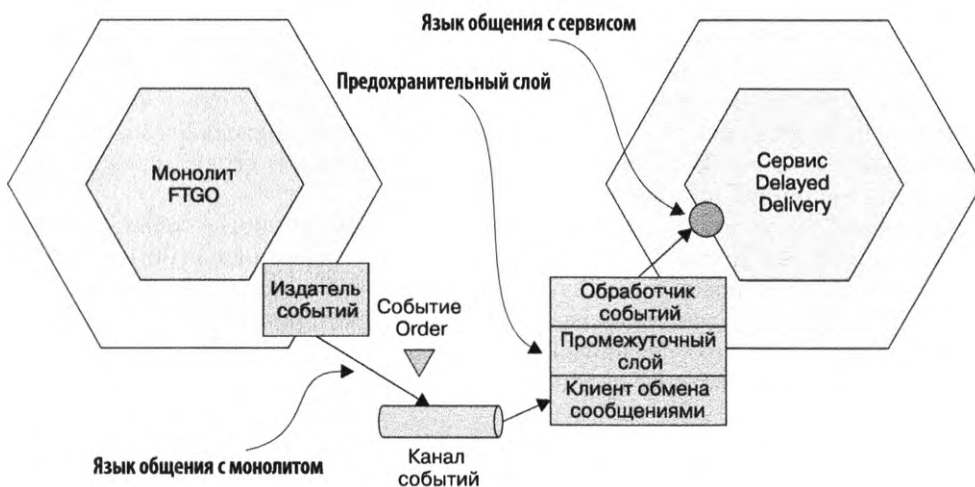


Рис. 13.12. Обработчик событий должен привести доменную модель издателя к доменной модели подписчика

Обработчик должен выполнять преобразования между доменными событиями монолита и сервиса **Delivery**, которые иногда включают в себя связывание классов, имен атрибутов и их значений.

Предохранительный слой применяется не только в сервисах. Монолит тоже использует ACL, обращаясь к сервису и подписываясь на доменные события, которые тот публикует. Например, монолит FTGO планирует доставку, отправляя уведомительное сообщение сервису **Delivery**. Для этого он вызывает метод из интерфейса **DeliveryService**. Реализация класса преобразует его параметры так, чтобы сервис **Delivery** мог их понять.

Как монолит публикует и подписывается на события

Доменные события — важный механизм взаимодействия. В свежеразработанном сервисе их публикация и потребление происходят довольно просто. Вы можете использовать один из механизмов, описанных в главе 3, таких как фреймворк Eventuate Tram. Сервис может даже применить методику порождения событий, рассмотренную в главе 6. Однако модификация монолита для публикации и потребления событий оказывается потенциально непростой задачей. Давайте посмотрим почему.

Монолит может публиковать доменные события несколькими способами. Один из подходов заключается в использовании того же механизма, который применяется в сервисах. Для этого нужно найти все участки кода, где изменяется определенная сущность, и вставить туда вызов API, публикующего события. Однако модификация монолита не всегда проходит гладко. Поиск всех участков кода и добавление вызовов для публикации событий может занять много времени и спровоцировать новые ошибки. Ситуация усугубляется еще и тем, что бизнес-логика может содержать хранимые процедуры, из которых нельзя так просто опубликовать доменные события.

Еще одним решением является публикация доменных событий на уровне базы данных. Для этого можно, к примеру, отслеживать логику транзакций пассивным или активным способом, как было описано в главе 3. Ключевое преимущество этого подхода состоит в том, что вам не нужно модифицировать монолит. Но у публикации доменных событий на уровне базы данных есть и обратная сторона: часто бывает сложно определить причину обновления и опубликовать подходящее высокоуровневое бизнес-событие. В итоге события, которые публикует сервис, представляют скорее изменения таблиц, чем бизнес-сущностей.

К счастью, монолиту обычно проще подписаться на доменные события, публикуемые сервисами. Довольно часто для написания обработчиков событий можно использовать такие фреймворки, как Eventuate Tram. Но иногда не все так просто. Например, монолит может быть написан на языке, у которого нет клиента для брокера сообщений. В таких ситуациях вам необходимо написать небольшое вспомогательное приложение, которое подписывается на события и обновляет базу данных монолита напрямую.

Итак, мы обсудили проектирование интеграционного слоя, который делает возможным взаимодействие сервиса и монолита. Остановимся еще на одной проблеме, с которой можно столкнуться при переходе на микросервисы: на обеспечении согласованности данных между сервисом и монолитом.

13.3.2. Обеспечение согласованности данных между сервисом и монолитом

При разработке сервиса может обнаружиться, что согласование данных между сервисом и монолитом вызывает определенные трудности. Сервису может понадобиться обновить данные в монолите и наоборот. Представьте, к примеру, что вы извлекли из монолита сервис `kitchen`. Вам придется перевести операции для управления заказами в монолитном коде, такие как `createOrder()` и `cancelOrder()`, на применение повествований, чтобы сущности `Ticket` и `Order` оставались согласованными.

Однако проблема повествований в том, что участие монолита в них не всегда проходит гладко. Как описывалось в главе 4, повествования должны использовать компенсирующие транзакции для отмены изменений. Повествование `Create Order`, например, помечает заказ как `REJECTED`, если сервис `Kitchen` его отклонил. Но для поддержки компенсирующих транзакций в монолите придется внести многочисленные изменения и потратить много времени. К тому же монолиту, возможно, придется реализовать контрмеры, чтобы справиться с недостаточной изоляцией между повествованиями. Стоимость таких изменений может стать большой проблемой при извлечении сервиса.

Ключевые термины, связанные с повествованиями

Повествования рассмотрены в главе 4. Вот некоторые основные термины:

- *повествование* — последовательность локальных транзакций, координируемых путем обмена асинхронными сообщениями;
- *компенсирующая транзакция* — транзакция, которая отменяет обновление, сделанное локальной транзакцией;
- *контрмера* — методика проектирования, которая позволяет справиться с недостаточной изоляцией между повествованиями;
- *семантическая блокировка* — контрмера, которая устанавливает флаг в записи, обновляемой в ходе повествования;
- *компенсируемая транзакция* — транзакция, которую нужно компенсировать на случай, если одна из последующих транзакций в повествовании завершится неудачно;
- *поворотная транзакция* — транзакция, которая определяет успешность повествования. Если она пройдет удачно, повествование дойдет до конца;
- *повторяемая транзакция* — транзакция, которая идет за поворотной и гарантированно завершается успехом.

К счастью, многие повествования реализуются довольно просто. Как было показано в главе 4, если транзакции монолита являются либо *поворотными*, либо *повторяемыми*, проблем с реализацией повествования возникнуть не должно. Вы можете даже упростить свою реализацию, тщательно спланировав последовательность извлечения сервисов таким образом, чтобы транзакции монолита никогда не нужно было делать компенсируемыми. Однако внедрить в монолит поддержку компенсируемых транзакций может оказаться непросто. Чтобы понять, чем это вызвано, рассмотрим некоторые примеры, начиная с особенно проблемного.

Трудности изменения монолита для поддержки компенсируемых транзакций

Давайте подробно рассмотрим проблему с компенсирующими транзакциями, которую необходимо решить при извлечении из монолита сервиса `Kitchen`. Этот рефакторинг подразумевает разделение сущности `Order` и создание в сервисе `Kitchen` сущности

`Ticket`. Он затрагивает множество команд, реализованных в монолите, включая `createOrder()`.

Монолит реализует команду `createOrder()` в виде единой ACID-транзакции, состоящей из следующих этапов.

1. Проверить детали заказа.
2. Убедиться в том, что клиент может размещать заказы.
3. Авторизовать банковскую карту клиента.
4. Создать заказ.

Эту ACID-транзакцию нужно заменить повествованием, которое состоит из таких шагов.

1. В монолите:
 - создать заказ с состоянием `APPROVAL_PENDING`;
 - убедиться в том, что клиент может размещать заказы.
2. В сервисе `Kitchen`:
 - проверить детали заказа;
 - создать заявку с состоянием `CREATE_PENDING`.
3. В монолите:
 - авторизовать банковскую карту клиента;
 - изменить состояние заказа на `APPROVED`.
4. В сервисе `Kitchen` — изменить состояние заявки на `AWAITING_ACCEPTANCE`.

Это повествование похоже на `CreateOrderSaga`, описанное в главе 4. Оно состоит из четырех локальных транзакций, по две в монолите и сервисе `Kitchen`. Первая транзакция создает заказ с состоянием `APPROVAL_PENDING`. Вторая создает заявку с состоянием `CREATE_PENDING`. Третья авторизует банковскую карту клиента и меняет состояние заказа на `APPROVED`. Четвертая, последняя, меняет состояние заявки на `AWAITING_ACCEPTANCE`.

Сложность реализации этого повествования состоит в том, что его первый шаг, на котором создается заказ, должен быть компенсирваемым. Это связано с тем, что вторая локальная транзакция, проходящая в сервисе `Kitchen`, может завершиться неудачно и потребовать от монолита отмены изменений, внесенных первой локальной транзакцией. В итоге сущность `Order` должна поддерживать контрмеру `APPROVAL_PENDING` — семантическую блокировку, описанную в главе 4, которая сигнализирует о том, что заказ находится в процессе создания.

Проблема с добавлением нового состояния в сущность `Order` заключается в том, что это может потребовать масштабной модификации монолита. Вам, вероятно, придется отредактировать каждый участок кода, который обращается к этой сущности. Такое масштабное обновление монолита занимает много времени, являясь при этом далеко не самым приоритетным аспектом разработки. К тому же это может оказаться рискованной затеей, ведь монолит обычно сложно тестировать.

Повествования не всегда требуют от монолита поддержки компенсируемых транзакций

Повествования сильно зависят от проблемной области. Некоторые (как только что рассмотренное) требуют, чтобы монолит поддерживал компенсирующие транзакции. Но извлечение сервисов вполне можно спланировать так, чтобы ваши повествования не нуждались в реализации компенсирующих транзакций со стороны монолита. Дело в том, что это необходимо, только последующие транзакции монолита могут завершиться неудачно. Если же все транзакции монолита являются либо поворотными, либо повторяемыми, ему никогда не нужно будет ничего компенсировать. В итоге для поддержки повествований в монолит нужно внести лишь небольшие изменения.

Представьте, к примеру, что вместо `Kitchen` вы извлекаете сервис `Order`. В ходе рефакторинга нужно разделить сущность `Order` в одноименном сервисе и сделать ее более «тонкой». Изменения коснутся и многочисленных команд, таких как `createOrder()`, которую следует перенести из монолита в сервис `Order`. Для извлечения сервиса нужно сделать так, чтобы эта команда использовала повествование, состоящее из следующих шагов.

1. Сервис `Order` — создать заказ с состоянием `APPROVAL_PENDING`.
2. Монолит:
 - убедиться в том, что клиент может размещать заказы;
 - проверить детали заказа и создать заявку;
 - авторизовать банковскую карту клиента.
3. Сервис `Order` — изменить состояние заказа на `APPROVED`.

Это повествование состоит из трех локальных транзакций: одна в монолите и две в сервисе `Order`. Первая транзакция (в сервисе `Order`) создает заказ с состоянием `APPROVAL_PENDING`. Вторая транзакция (в монолите) проверяет, может ли клиент размещать заказы, авторизует его банковскую карту и создает заявку. Третья (опять в сервисе `Order`) меняет состояние заказа на `APPROVED`.

Транзакция монолита является поворотной для этого повествования, его точкой невозврата. Если она завершится успешно, повествование доработает до самого конца. Проблемы могут возникнуть только с первыми двумя этапами. Третья транзакция не может отказать, поэтому монолиту никогда не придется откатывать вторую транзакцию. В результате вся сложность поддержки компенсируемых транзакций ложится на сервис `Order`, который тестировать намного легче, чем монолит.

Если все повествования, которые вы напишете при извлечении сервиса, будут иметь такую структуру, вам придется сделать намного меньше изменений в монолите. Более того, извлечение сервисов можно тщательно спланировать в таком порядке, чтобы все транзакции монолита были либо поворотными, либо повторяемыми. Посмотрим, как это сделать.

Планирование извлечения сервисов, чтобы избежать реализации компенсирующих транзакций в монолите

Как мы только что видели, извлечение сервиса `Kitchen` требует от монолита реализации компенсирующих транзакций, а извлечение сервиса `Order` — нет. Это говорит о том, что порядок, в котором извлекаются сервисы, имеет значение. Если тщательно его спланировать, можно избежать внесения масштабных изменений в монолит для поддержки компенсируемых транзакций. Мы можем сделать так, чтобы все транзакции в монолите были либо поворотными, либо повторяемыми. Например, если извлечь из монолита FTGO сначала сервис `Order`, а затем `Consumer`, это упростит извлечение сервиса `Kitchen`. Давайте посмотрим, как это делается.

После извлечения сервиса `Consumer` команда `createOrder()` использует следующее повествование.

1. Сервис `Order` — создать заказ с состоянием `APPROVAL_PENDING`.
2. Сервис `Consumer` — убедиться в том, что клиент может размещать заказы.
3. Монолит:
 - проверить детали заказа и создать заявку;
 - авторизовать банковскую карту клиента.
4. Сервис `Order` — изменить состояние заказа на `APPROVED`.

В этом повествовании транзакция монолита является поворотной. Компенсируемую транзакцию реализует сервис `Order`.

Вслед за `Consumer` мы можем извлечь сервис `Kitchen`. Когда мы это сделаем, команда `createOrder()` будет использовать следующее повествование.

1. Сервис `Order` — создать заказ с состоянием `APPROVAL_PENDING`.
2. Сервис `Consumer` — убедиться в том, что клиент может размещать заказы.
3. Сервис `Kitchen` — проверить детали заказа и создать заявку с состоянием `PENDING`.
4. Монолит — авторизовать банковскую карту клиента.
5. Сервис `Kitchen` — изменить состояние заявки на `APPROVED`.
6. Сервис `Order` — изменить состояние заказа на `APPROVED`.

В этом повествовании транзакция монолита по-прежнему остается поворотной. Компенсируемые транзакции реализуются сервисами `Order` и `Kitchen`.

Мы можем продолжить рефакторинг монолита и извлечь сервис `Accounting`. Если сделаем это, команда `createOrder()` будет использовать такое повествование.

1. Сервис `Order` — создать заказ с состоянием `APPROVAL_PENDING`.
2. Сервис `Consumer` — убедиться в том, что клиент может размещать заказы.
3. Сервис `Kitchen` — проверить детали заказа и создать заявку с состоянием `PENDING`.

4. Сервис **Accounting** — авторизовать банковскую карту клиента.
5. Сервис **Kitchen** — изменить состояние заявки на **APPROVED**.
6. Сервис **Order** — изменить состояние заказа на **APPROVED**.

Как видите, тщательное планирование порядка извлечения позволяет избежать использования повествований, которые требуют внесения сложных изменений в монолит. Теперь посмотрим, как обеспечить безопасность при переходе на микросервисы.

13.3.3. Аутентификация и авторизация

Еще одна проблема, которую нужно решить при переводе монолитного приложения на микросервисную архитектуру, — адаптация существующего механизма безопасности для поддержки сервисов. Реализация механизма безопасности в микросервисной архитектуре описана в главе 11. Приложение, основанное на микросервисах, передает пользовательские данные с помощью токенов, таких как JWT (JSON Web token). Это довольно сильно отличается от традиционного монолитного подхода, который предусматривает хранение состояния сеанса в памяти и передачу пользовательских данных с помощью внутривызовных локальных переменных. Трудность состоит в том, что вам придется одновременно поддерживать оба механизма безопасности, как монолитный, так и основанный на JWT.

К счастью, существует простой и понятный способ решения этой проблемы, который требует внести лишь одно небольшое изменение в обработчик входа в систему внутри монолита. На рис. 13.13 показано, как это работает. Обработчик входа в систему возвращает дополнительный cookie-файл, который я в этом примере назвал **USERINFO**. Он содержит информацию о пользователе, такую как его ID и роли. Браузер включает этот cookie в каждый запрос. API-шлюз извлекает содержимое cookie и добавляет его в HTTP-запрос, направленный к сервису. В итоге каждый сервис получает доступ к необходимой пользовательской информации.

Происходит такая последовательность событий.

1. Клиент делает запрос входа в систему, содержащий учетные данные пользователя.
2. API-шлюз направляет запрос входа в систему к монолиту FTGO.
3. Монолит возвращает ответ с cookie сеанса **JSESSIONID** и cookie **USERINFO**, который содержит такую пользовательскую информацию, как ID и роли.
4. Клиент делает запрос с cookie **USERINFO** внутри, чтобы вызвать операцию.
5. API-шлюз проверяет cookie **USERINFO** и включает его в заголовок **Authorization** запроса, который шлет сервису. Сервис проверяет токен **USERINFO** и извлекает информацию о пользователе.

Давайте подробнее рассмотрим **LoginHandler** и API-шлюз.

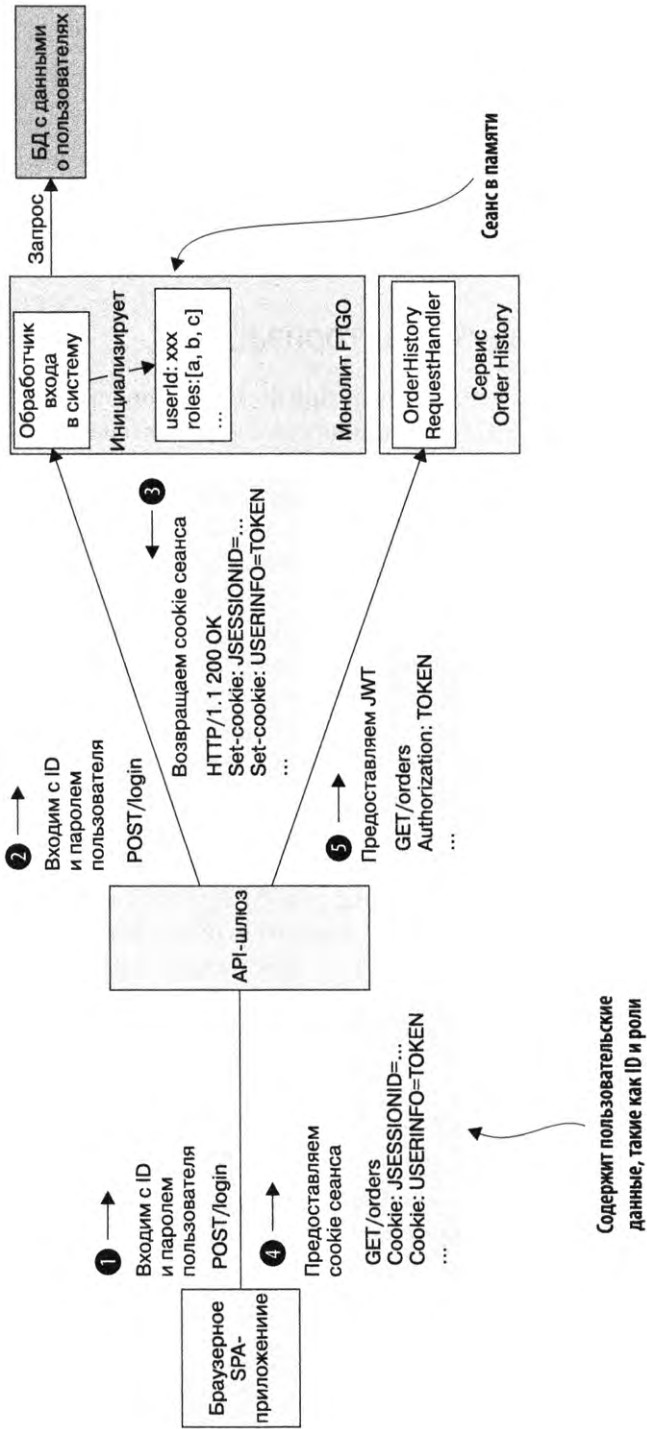


Рис. 13.13. Модифицированный обработчик входа в систему устанавливает cookie, который является JWT с информацией о пользователе. API-шлюз передает cookie USERINFO в заголовке авторизации при обращении к сервису

Обработчик LoginHandler монолита устанавливает cookie USERINFO

LoginHandler обрабатывает POST-запрос с учетными данными. Он аутентифицирует пользователя и сохраняет информацию о нем в сеансе. Часто этот процесс реализован в фреймворках безопасности, таких как Spring Security или Passport для NodeJS. Если приложение сконфигурировано для использования сеансов, хранимых в памяти, HTTP-ответ устанавливает cookie сеанса, такой как JSESSIONID. Для поддержки перехода на микросервисы обработчик LoginHandler должен также установить cookie USERINFO с токеном JWT внутри, который описывает пользователя.

API-шлюз привязывает cookie USERINFO к заголовку Authorization

Как говорилось в главе 8, API-шлюз отвечает за маршрутизацию запросов и объединение API. При обработке каждого запроса он делает одно или несколько обращений к монолиту и сервисам. Когда API-шлюз вызывает сервис, он проверяет cookie USERINFO и передает его внутри HTTP-запроса в заголовке Authorization. Привязывая cookie к этому заголовку, он гарантирует, что сервис получит пользовательскую информацию стандартным путем, который не зависит от типа клиента.

Рано или поздно мы, скорее всего, выделим вход в систему и управление пользователями в отдельные сервисы. Но, как видите, благодаря одному лишь небольшому изменению в обработчике монолита сервисы могут получить доступ к пользовательской информации. Это позволяет сосредоточиться на разработке сервисов, предоставляющих максимальную пользу с точки зрения бизнеса, и отложить на потом извлечение менее полезных возможностей, таких как управление пользователями.

Итак, мы разобрались с тем, как обеспечить безопасность при переходе на микросервисы. Теперь рассмотрим пример реализации новой функциональности в виде сервиса.

13.4. Реализация новой возможности в виде сервиса

Представьте, что вам было поручено улучшить то, как FTGO работает с недоставленными заказами. Все больше клиентов жалуется на то, как служба поддержки относится к заказам, которые не были доставлены. В большинстве случаев доставка происходит вовремя, но время от времени заказы задерживаются или вовсе не доставляются. Например, если курьер неожиданно попадет в пробку, он заберет и доставит еду с задержкой. Бывает и так, что в момент прибытия курьера ресторан уже закрыт, поэтому доставка невозможна. Ситуация усугубляется еще и тем, что служба поддержки узнает о недоставленном заказе из гневных писем недовольных клиентов.

История из жизни: мое пропавшее мороженое

В одну субботнюю ночь я разлился и заказал мороженое Smitten через популярное приложение для доставки еды. Но не получил его. На следующее утро пришло электронное письмо о том, что заказ отменен, — этим и ограничилось мое общение с компанией. Мне также пришло голосовое сообщение от очень смущенной сотрудницы службы поддержки, которая явно не знала, зачем мне звонила. Возможно, этот звонок был реакцией на один из моих твитов, в которых я описал случившееся. Очевидно, что у компании, занимающейся доставкой, не было никаких механизмов для того, чтобы должным образом справляться с неизбежными ошибками.

Первопричина многих подобных проблем состоит в примитивном алгоритме планирования доставки, который используется в приложении FTGO. Более сложный планировщик уже разрабатывается, но будет готов лишь через несколько месяцев. А пока компания FTGO будет действовать на упреждение, извиняясь перед клиентами за задержанные и отмененные заказы и в некоторых случаях предлагая компенсацию еще до поступления жалобы.

От вас требуется реализовать следующие возможности.

1. Уведомление клиента в случае, если его заказ не будет доставлен вовремя.
2. Уведомление клиента в случае, если его заказ не будет доставлен из-за того, что ресторан закрывается раньше отгрузки.
3. Уведомление работников службы поддержки о том, что заказ задерживается, чтобы они могли исправить ситуацию, заранее компенсировав клиенту неудобства.
4. Отслеживание статистики доставки.

Эти новые функции довольно просты. Новый код должен следить за состоянием заказа и в случае, если его нельзя доставить в запланированное время, уведомлять клиента и службу поддержки, например, электронным письмом.

Но как (или, если быть точным, *где*) реализовать эти новые возможности? Для этого можно создать отдельный модуль в монолите. Но это вызвало бы трудности с разработкой и тестированием данного кода. К тому же в этом случае монолит напухнет еще сильнее, что только усугубит ситуацию. Вспомните закон ямы, о котором упоминалось ранее: если вы оказались в яме, нужно перестать копать. Лучше реализовать эти возможности в виде сервиса, не увеличивая монолит.

13.4.1. Архитектура сервиса Delayed Delivery

Мы реализуем эти функции в виде сервиса `Delayed Delivery`. Этот процесс в контексте архитектуры приложения FTGO представлен на рис. 13.14. У нас есть монолит, сервис `Delayed Delivery` и API-шлюз. У сервиса есть API с одной запрашивающей операцией `getDelayedOrders()`, которая возвращает задерживающиеся и недо-

13.4.2. Проектирование интеграционного слоя для сервиса Delayed Order

Сервисы, реализующие новые возможности, определяют собственные классы сущностей, но в то же время обычно обращаются к данным, принадлежащим монолиту. Не исключение и сервис **Delayed Delivery**. Он содержит сущность **DelayedOrderNotification**, которая представляет уведомление, отправляемое клиенту. Но, как я только что отметил, его сущности **Order** и **Restaurant** реплицируют данные из монолита FTGO. Кроме того, чтобы уведомить пользователя, этому сервису нужно запросить его контактную информацию. Таким образом, необходимо реализовать интеграционный слой, который позволит сервису **Delivery** обращаться к данным монолита.

Архитектура этого интеграционного слоя показана на рис. 13.15. Монолит FTGO публикует доменные события **Order** и **Restaurant**, а сервис **Delivery** их потребляет и обновляет свои реплики соответствующих сущностей. Монолит FTGO предоставляет конечную точку REST для получения контактной информации клиента. Сервис **Delivery** обращается к этой конечной точке, когда ему нужно уведомить пользователя о том, что доставку нельзя выполнить вовремя.

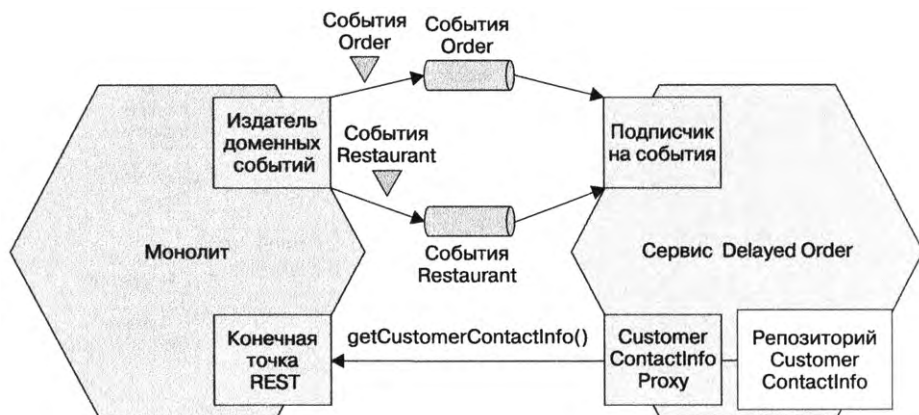


Рис. 13.15. Интеграционный слой предоставляет сервису Delayed Order доступ к данным, принадлежащим монолиту

Рассмотрим структуру каждого элемента интеграционного слоя, начиная с REST API для получения контактной информации клиента.

Запрашивание контактной информации клиента с помощью CustomerContactInfoRepository

Как было сказано в подразделе 13.3.1, такой сервис, как **Delayed Delivery**, может прочитать данные монолита несколькими способами. Самый простой вариант состоит в извлечении данных монолита через его API. Он подходит для получения

контактной информации пользователя. У нас не будет никаких проблем с латентностью и производительностью, поскольку сервис **Delayed Delivery** использует эту операцию лишь изредка и объем передаваемых данных довольно небольшой.

CustomerContactInfoRepository — это интерфейс, который позволяет сервису **Delayed Delivery** извлекать контактную информацию клиентов. Его реализация — класс **CustomerContactInfoProxy** — извлекает нужные данные, обращаясь через REST к **getCustomerContactInfo()** — конечной точке монолита.

Публикация и потребление доменных событий **Order** и **Restaurant**

К сожалению, запрашивать все открытые заказы и время работы всех ресторанов было бы непрактично. Это потребовало бы многократной передачи больших объемов данных по сети. По этой причине сервис **Delayed Delivery** должен использовать второй, более сложный вариант: хранить реплики **Order** и **Restaurant**, подписываясь на события, которые публикует монолит. Не стоит забывать, что реплика — это не полная копия данных монолита, она содержит лишь небольшое подмножество атрибутов сущностей **Order** и **Restaurant**.

Как было сказано в подразделе 13.3.1, добавить в монолит FTGO поддержку публикации доменных событий **Order** и **Restaurant** можно несколькими разными способами. Во-первых, вы можете отредактировать все участки монолита, которые обновляют **Order** и **Restaurant**, вставив туда публикацию высокоуровневых доменных событий. Во-вторых — отслеживать журнал транзакций и репликации изменений в виде событий. Публикация высокоуровневых доменных событий — не обязательное требование, поэтому нам подойдет любой вариант.

Сервис **Delayed Order** реализует обработчики, которые подписываются на события монолита и обновляют его сущности **Order** и **Restaurant**. Детали их реализации зависят от того, какие события публикует монолит, высокоуровневые или низкоуровневые (об изменениях). В любом случае мы можем сформулировать этот процесс таким образом: обработчик берет событие из изолированного контекста монолита и обновляет сущность в изолированном контексте сервиса.

У использования репликации есть важное преимущество: оно позволяет сервису **Delayed Order** эффективно запрашивать заказы и время работы ресторанов. Его недостатки — повышенная сложность и необходимость публикации событий **Order** и **Restaurant** из монолита. К счастью, сервису **Delayed Delivery** нужна лишь часть столбцов из таблиц **ORDERS** и **RESTAURANT**, поэтому у нас не должно возникнуть проблем, описанных в подразделе 13.3.1.

Реализация новых возможностей, таких как управление опаздывающими заказами, в виде отдельных сервисов ускоряет их разработку, тестирование и развертывание. Более того, это дает возможность применять самые современные технологии вместо более старого технологического стека монолита. Монолит при этом перестает расти. Управление опаздывающими заказами — это лишь одна из новых функций,

запланированных для приложения FTGO. Многие из них можно будет реализовать в виде отдельных сервисов.

К сожалению, сервисы не могут вобрать в себя все обновления. Довольно часто для реализации новых или изменения существующих возможностей приходится существенно модифицировать сам монолит. Работа с монолитным кодом обычно оказывается медленной и мучительной. Чтобы ускорить доставку таких модификаций, вы должны разбить монолит на части и перенести их функции в сервисы. Давайте посмотрим, как это сделать.

13.5. Разбиение монолита на части: извлечение управления доставкой

Чтобы ускорить доставку возможностей, реализованных в монолите, вы должны разбить монолитный код на сервисы. Представьте, к примеру, что вам захотелось улучшить управление доставкой за счет нового алгоритма маршрутизации. Основной преградой на этом пути будет то, что данный механизм переплетается с управлением заказами и является частью кодовой базы монолита. Разработка, тестирование и развертывание модуля управления доставкой, скорее всего, займут много времени. Чтобы ускорить процесс, нужно извлечь этот модуль в сервис **Delivery**.

Для начала я опишу управление доставкой и объясню, как оно в настоящий момент встроено в монолит. Затем мы поговорим о проектировании нового, отдельного сервиса **Delivery** со своим API. Я покажу, как этот сервис взаимодействует с монолитом FTGO. В конце мы рассмотрим изменения, которые необходимо внести в монолит для поддержки сервиса **Delivery**.

Начнем с обзора текущей архитектуры.

13.5.1. Обзор возможностей существующего механизма управления доставкой

Управление доставкой отвечает за планирование отгрузки заказов курьерам и их доставку клиентам. У каждого курьера есть план действий по приему и доставке заказов. *Прием* — это получение курьером заказа в ресторане в заданное время. *Доставка* — передача заказа клиенту. Планы пересматриваются при размещении, отмене или редактировании заказов, а также изменении местоположения и доступности курьеров.

Управление доставкой — одна из самых старых частей приложения FTGO. Оно встроено в систему управления заказами (рис. 13.16). Большая часть кода для работы с доставкой находится в классе **OrderService**. Более того, у нас нет сущности, которая бы явно представляла доставку. Этот код принадлежит сущности **Order**, содержащей различные поля, связанные с доставкой, включая **scheduledPickupTime** и **scheduledDeliveryTime**.

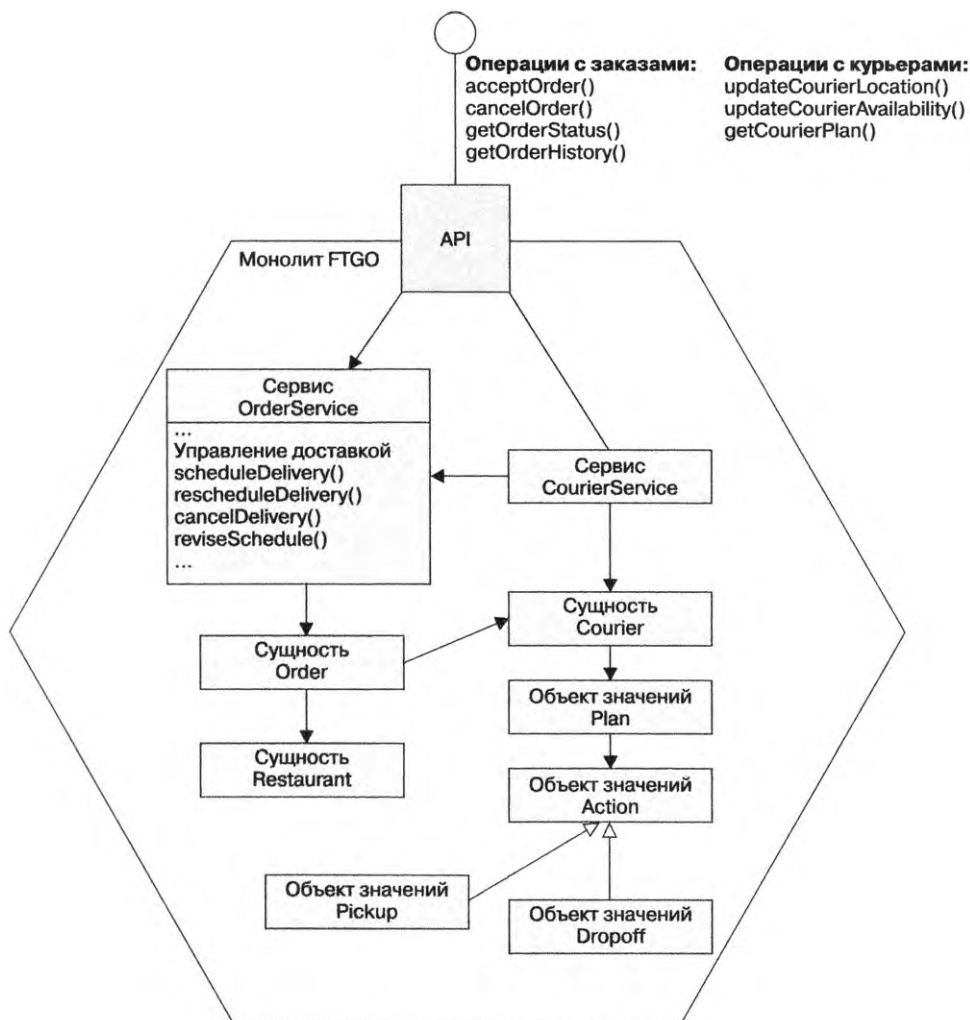


Рис. 13.16. В монолите FTGO управление доставкой переплетается с управлением заказами

Монолит реализует множество команд для управления доставкой:

- ❑ `acceptOrder()` — вызывается, когда ресторан принимает заказ и обязуется подготовить его к определенному времени. Эта операция включает в себя планирование доставки;
- ❑ `cancelOrder()` — вызывается, когда клиент отменяет заказ. В случае необходимости отменяет доставку;
- ❑ `noteCourierLocationUpdated()` — вызывается мобильным приложением курьера для обновления его местоположения. Приводит к перепланированию доставок;

- ❑ `noteCourierAvailabilityChanged()` — вызывается мобильным приложением курьера для обновления его доступности. Приводит к перепланированию доставок.

Есть также различные запросы для извлечения данных, с которыми работает механизм управления доставкой:

- ❑ `getCourierPlan()` — вызывается мобильным приложением курьера и возвращает его план действий;
- ❑ `getOrderStatus()` — возвращает состояние заказа вместе с информацией о доставке, такой как назначенный курьер и ожидаемое время прибытия;
- ❑ `getOrderHistory()` — возвращает то же, что и `getOrderStatus()`, только для нескольких заказов.

Как упоминалось в подразделе 13.2.3, в сервис довольно часто извлекается весь вертикальный срез с контроллерами вверх и таблицами базы данных вниз. Команды и запросы, связанные с курьером, можно рассматривать как часть управления доставкой. В конце концов, этот механизм создает планы действия курьеров и является основным потребителем информации об их местоположении и доступности. Но, чтобы минимизировать усилия, затрачиваемые на разработку, мы оставим эти операции в монолите и извлечем только основную часть алгоритма. Таким образом, у первой версии сервиса **Delivery** не будет публичного API, она будет вызываться исключительно монолитом. Теперь исследуем архитектуру сервиса **Delivery**.

13.5.2. Обзор сервиса Delivery

Новый сервис **Delivery**, который мы хотим создать, отвечает за планирование, перепланирование и отмену доставки. Обобщенное представление архитектуры приложения FTGO после извлечения сервиса **Delivery** показано на рис. 13.17. Монолит и сервис общаются между собой с помощью интеграционного слоя, API которого размещены по обе стороны. Сервис **Delivery** имеет собственную доменную модель и базу данных.

Чтобы воплотить эту архитектуру в жизнь и определить доменную модель сервиса, необходимо ответить на следующие вопросы.

- ❑ Какие данные и логика перемещаются в сервис?
- ❑ Какой API сервис **Delivery** предоставляет монолиту?
- ❑ Какой API монолит предоставляет сервису **Delivery**?

Эти вопросы связаны между собой, так как распределение ответственности между монолитом и сервисом влияет на API. Например, для доступа к данным, размещенным в БД монолита, сервису **Delivery** нужно будет обращаться к API, которые тот предоставляет, и наоборот. Позже я опишу структуру интеграционного слоя, который делает возможным общение сервиса **Delivery** и монолита FTGO. Но сначала рассмотрим доменную модель сервиса **Delivery**.

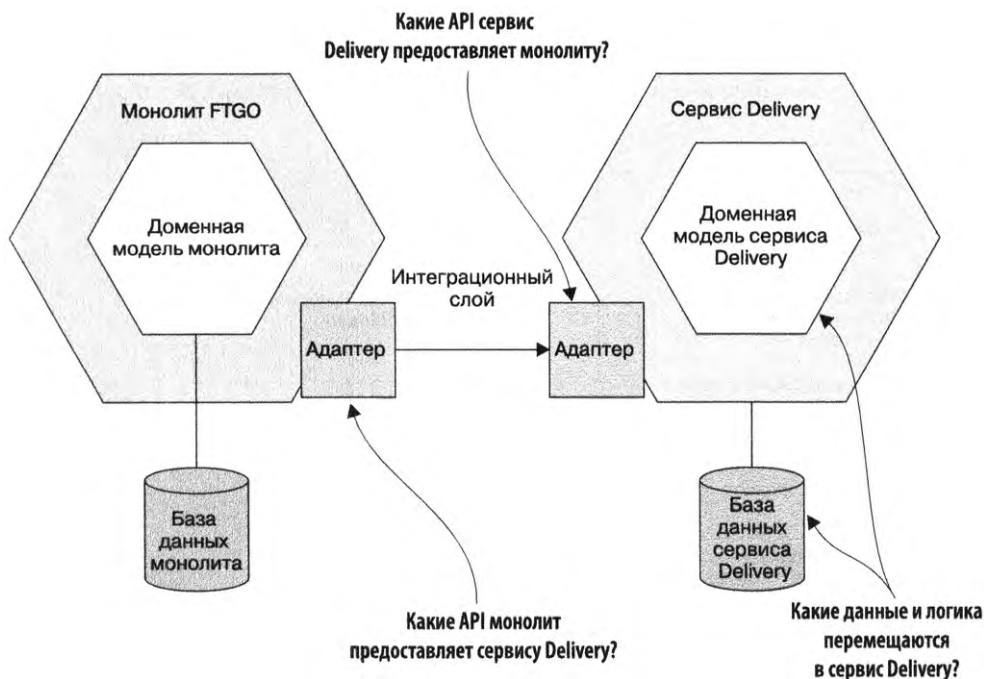


Рис. 13.17. Обобщенная схема приложения FTGO после извлечения сервиса Delivery. Монолит FTGO и сервис Delivery общаются с помощью интеграционного слоя, API которого размещены по обе стороны. Необходимо принять два ключевых решения: какие данные и функциональность перемещаются в сервис Delivery и через какие API монолит и сервис Delivery будут взаимодействовать между собой

13.5.3. Проектирование доменной модели сервиса Delivery

Чтобы извлечь управление доставкой, нужно сначала определить классы, которые ее реализуют. Сделав это, мы сможем решить, какие классы следует переместить в сервис Delivery, чтобы сформировать его доменную логику. Иногда классы приходится разделять. Также нужно определиться с тем, какие данные будут реплицироваться между сервисом и монолитом.

Для начала выделим классы, которые реализуют управление доставкой.

Какие сущности и их поля относятся к управлению доставкой

Первое, что нужно сделать при проектировании сервиса Delivery, — тщательно проанализировать код управления доставкой и определить, какие сущности и их поля в этом участвуют. Сущности и их поля, которые принимают участие в управлении

доставкой, показаны на рис. 13.18. Некоторые из полей служат параметрами для алгоритма планирования доставки, а другие хранят возвращаемые значения. Далее также показано, какие из них используются другими функциями, реализованными в монолите.

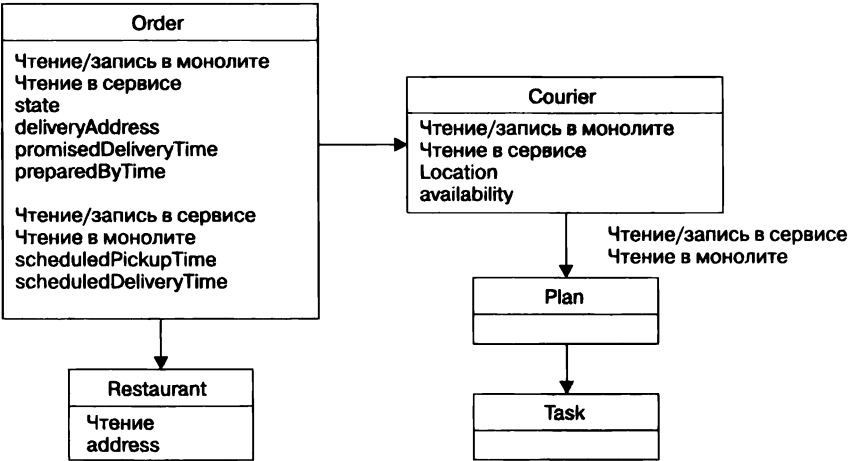


Рис. 13.18. Сущности и поля, которые участвуют в управлении доставкой и других функциях, реализованных монолитом. Поле может быть доступно для чтения и/или записи. К нему может обращаться механизм управления доставкой и/или монолит

Алгоритм планирования доставки считывает различные атрибуты, включая restaurant, promisedDeliveryTime и deliveryAddress из Order и location и availability из Courier, а также текущие планы. Он обновляет планы Courier и поля scheduledPickupTime и scheduledDeliveryTime из Order. Как видите, поля, участвующие в управлении доставкой, используются и в монолите.

Какие данные следует перенести в сервис Delivery

Определившись с тем, какие сущности и поля участвуют в управлении доставкой, мы должны решить, какие из них должны попасть в сервис. Если бы данные, нужные сервису, использовались только им самим, их извлечение было бы крайне прямым. Но такие идеальные ситуации встречаются довольно редко, и этот случай не исключение. Все сущности и поля, которые задействуются для управления доставкой, участвуют в других функциях, реализованных монолитом.

В итоге при определении того, какие данные нужно переместить в сервис, следует учитывать два момента: каким образом сервис обращается к данным, остающимся в монолите, и как монолит обращается к данным, вынесенным в сервис. К тому же, как сказано в разделе 13.3, нужно тщательно продумать способ согласования данных между сервисом и монолитом.

Основная функция сервиса **Delivery** заключается в управлении планами действий курьеров и обновлении полей `scheduledPickupTime` и `scheduledDeliveryTime` сущности **Order**. Таким образом, имеет смысл перенести эти поля в сам сервис. То же самое можно было бы сделать с полями `Courier.location` и `Courier.availability`. Но поскольку мы пытаемся минимизировать изменения, их лучше пока оставить в монолите.

Структура доменной логики сервиса Delivery

Структура доменной модели сервиса **Delivery** показана на рис. 13.19. В ее основе лежат доменные классы **Delivery** и **Courier**. Класс **DeliveryServiceImpl** служит точкой входа в бизнес-логику управления доставкой. Он реализует интерфейсы **DeliveryService** и **CourierService**, к которым обращаются обработчики **DeliveryServiceEventsHandler** и **DeliveryServiceNotificationsHandlers**, описанные ранее в этом разделе.

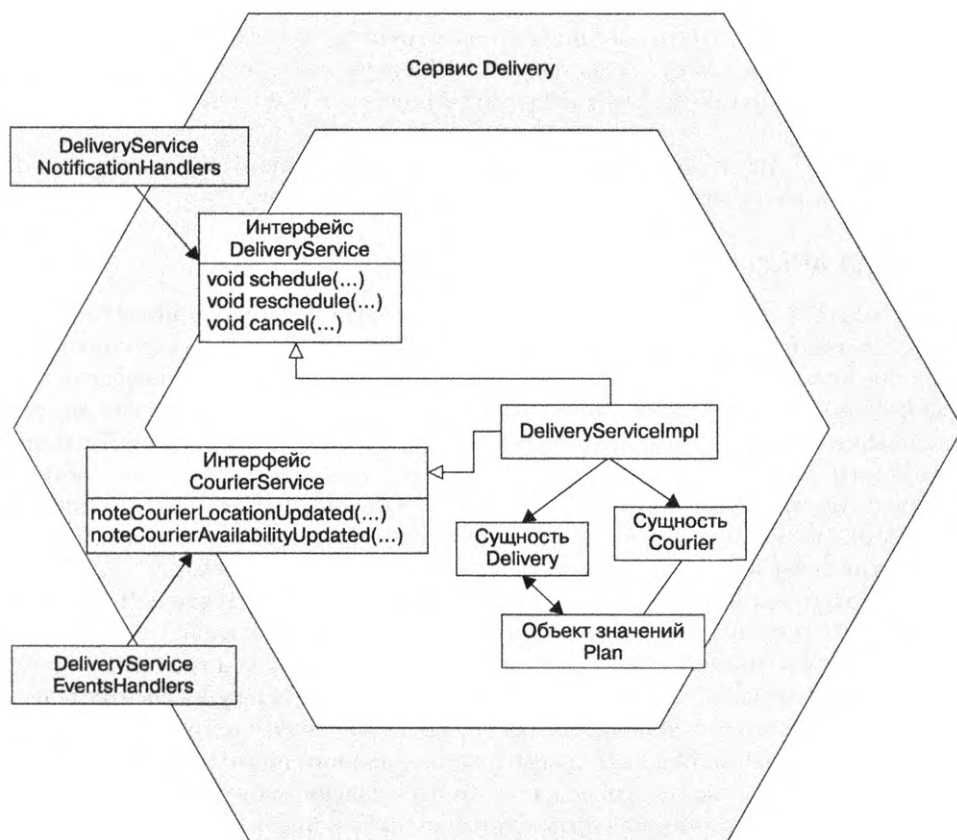


Рис. 13.19. Структура доменной модели сервиса Delivery

Бизнес-логика управления доставкой в основном состоит из кода, скопированного из монолита. Например, мы скопируем в сервис **Delivery** сущность **Order** и переименуем ее в **Delivery**, а также удалим из нее все поля, кроме задействованных в управлении доставкой. То же самое сделаем с сущностью **Courier**. Чтобы разработать доменную логику для сервиса **Delivery**, придется отвязать код от монолита, разрывая множество зависимостей. Это может занять много времени. И снова мы приходим к тому, что рефакторинг кода намного проще происходит в статически типизированных языках, поскольку в этом случае вам будет помогать компилятор.

Сервис **Delivery** самостоятельный. Рассмотрим структуру интеграционного слоя, который позволяет ему взаимодействовать с монолитом **FTGO**.

13.5.4. Структура интеграционного слоя для сервиса **Delivery**

Чтобы управлять доставкой, монолиту **FTGO** необходимо обращаться к сервису **Delivery** и обмениваться с ним данными. Это взаимодействие становится возможным благодаря интеграционному слою, структура которого представлена на рис. 13.20. Сервис **Delivery** предоставляет API для управления доставкой и публикует доменные события **Delivery** и **Courier**. Монолит **FTGO** публикует доменные события **Courier**.

Рассмотрим структуру всех составляющих интеграционного слоя, начиная с API для управления доставкой, принадлежащего сервису **Delivery**.

Структура API сервиса **Delivery**

Сервис **Delivery** должен предоставлять API, который позволит монолиту планировать, пересматривать и отменять доставку. Как вы уже видели на страницах этой книги, предпочтительным подходом является обмен асинхронными сообщениями, поскольку он поощряет слабую связанность и улучшает доступность. Сервис **Delivery** может подписаться на доменные события **Order**, публикуемые монолитом. Это позволит создавать, пересматривать и отменять доставку в зависимости от типа события. Преимущество этого решения заключается в том, что монолиту не нужно обращаться к сервису **Delivery** напрямую. Но есть и недостаток: сервис должен знать, как каждый тип события **Order** влияет на соответствующую сущность **Delivery**.

Более разумным подходом будет реализовать в сервисе **Delivery** API на основе уведомлений, который позволит монолиту явно управлять доставкой — планировать, пересматривать и отменять. Этот API будет состоять из канала и трех типов проходящих через него сообщений: **ScheduleDelivery**, **ReviseDelivery** и **CancelDelivery**. Например, уведомление **ScheduleDelivery** содержит время отгрузки, время доставки и адрес клиента. Важным преимуществом данного подхода является то, что сервису **Delivery** не нужно знать подробностей о жизненном цикле сущности **Order**. Он полностью сосредоточен на управлении доставкой и никак не связан с заказами.

Взаимодействие сервиса **Delivery** и монолита **FTGO** не ограничивается лишь этим API. Также им нужно обмениваться данными.

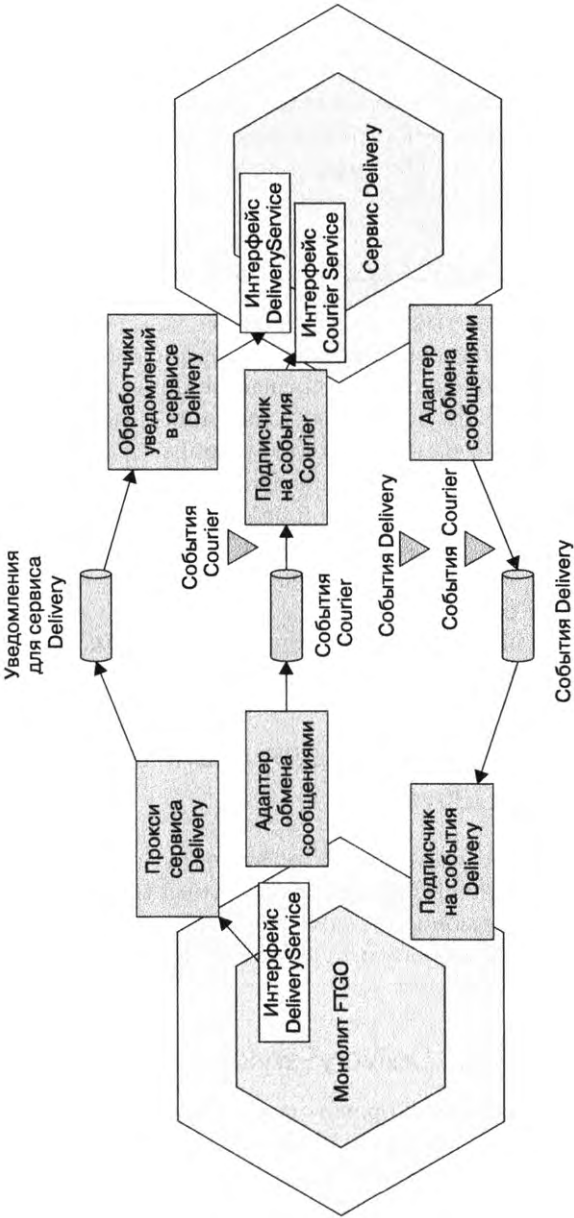


Рис. 13.20. Структура интеграционного слоя для сервиса Delivery. Сервис Delivery имеет API для управления доставкой. Сервис и монолит FTGO синхронизируют данные, обмениваясь доменными событиями

Как сервис Delivery получает доступ к данным монолита FTGO

Сервису Delivery нужна информация о местоположении курьера и его доступности, принадлежащая монолиту. Поскольку объем этих данных может быть большим, сервису непрактично постоянно запрашивать их у монолита. Вместо этого монолит может реплицировать данные в сервис Delivery, публикуя доменные события `CourierLocationUpdated` и `CourierAvailabilityUpdated`. У сервиса Delivery есть объект `CourierEventSubscriber`, который подписывается на эти события и обновляет свою версию сущности `Courier`. Он может также инициировать повторное планирование доставки.

Как монолит FTGO получает доступ к данным сервиса Delivery

Монолиту FTGO нужно считывать такие данные, как план действий курьера, которые были перенесены в сервис Delivery. Теоретически он мог бы их просто запрашивать, но это потребовало бы его существенной модификации. Пока что проще оставить доменную модель и схему базы данных монолита без изменений и реплицировать данные из сервиса обратно в монолит.

Чтобы этого добиться, проще всего публиковать из сервиса Delivery доменные события для сущностей `Courier` и `Delivery`. Сервис публикует событие `CourierPlanUpdated` при обновлении плана действий курьера и событие `DeliveryScheduleUpdate` при обновлении доставки. Все это потребляется монолитом, который обновляет свою базу данных.

Мы рассмотрели, как монолит FTGO и сервис Delivery взаимодействуют между собой. Теперь возьмемся за изменение монолита.

13.5.5. Изменение монолита для взаимодействия с сервисом Delivery

Реализация сервиса Delivery — во многом самая простая часть процесса извлечения. Изменение монолита FTGO будет намного более сложным. К счастью, репликация данных из сервиса обратно в монолит сокращает объем этих изменений. Но нам все равно нужно сделать так, чтобы монолит управлял доставкой через сервис Delivery. Давайте посмотрим, как это сделать.

Определение интерфейса DeliveryService

Прежде всего необходимо инкапсулировать код управления доставкой внутри Java-интерфейса, который будет привязан к API на основе обмена сообщениями, определенному ранее. Этот интерфейс (рис. 13.21) содержит методы для планирования, перепланирования и отмены доставки.

Позже мы реализуем этот интерфейс с помощью прокси-класса, который шлет сообщения сервису Delivery. Но сначала наша реализация будет вызывать код управления доставкой.

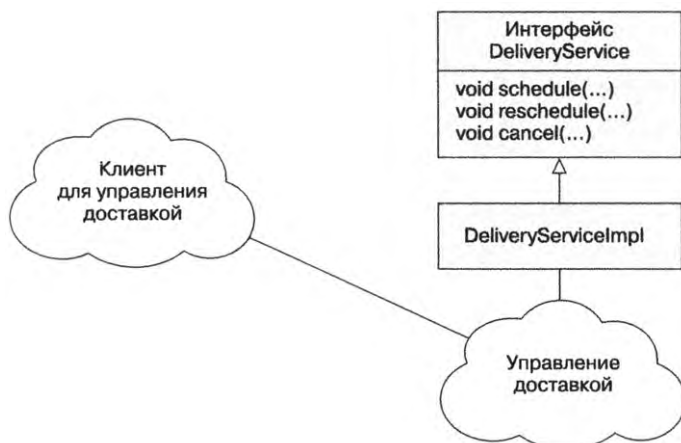


Рис. 13.21. Первый шаг — определение обобщенного API `DeliveryService` для удаленного обращения к логике управления доставкой

Интерфейс `DeliveryService` является обобщенным, поэтому для его реализации хорошо подходит механизм IPC. Он содержит методы `schedule()`, `reschedule()` и `cancel()`, которые относятся к разным типам сообщений, описанным ранее.

Рефакторинг монолита для вызова интерфейса `DeliveryService`

Далее нужно определить в монолите FTGO все участки кода, которые занимаются управлением доставкой, и сделать так, чтобы они использовали интерфейс `DeliveryService` (рис. 13.22). Это один из самых сложных аспектов извлечения сервиса из монолита. Он может занять некоторое время.

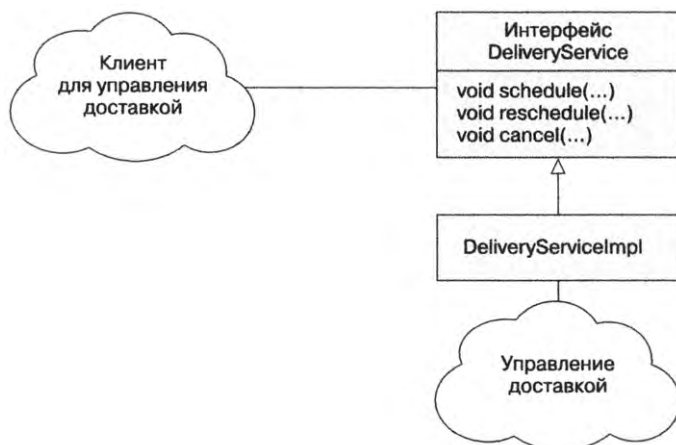


Рис. 13.22. Вторым шагом будет модификация монолита FTGO для управления доставкой через интерфейс `DeliveryService`

Процесс, безусловно, будет проще, если монолит написан на статически типизированном языке, таком как Java, который предоставляет лучшие инструменты для определения зависимостей. В противном случае желательно иметь автоматические тесты с достаточным охватом участков кода, которые нужно изменить.

Реализация интерфейса DeliveryService

Завершающим шагом будет замена класса `DeliveryServiceImpl` прокси-классом, который шлет уведомления отдельному сервису `Delivery`. Но вместо того, чтобы сразу отказываться от существующей реализации, мы воспользуемся схемой, показанной на рис. 13.23, которая позволяет монолиту динамически переключаться между классом `DeliveryServiceImpl` и сервисом `Delivery`. Для этого мы реализуем интерфейс `DeliveryService` с помощью класса, который будет применять динамическое переключение возможностей.

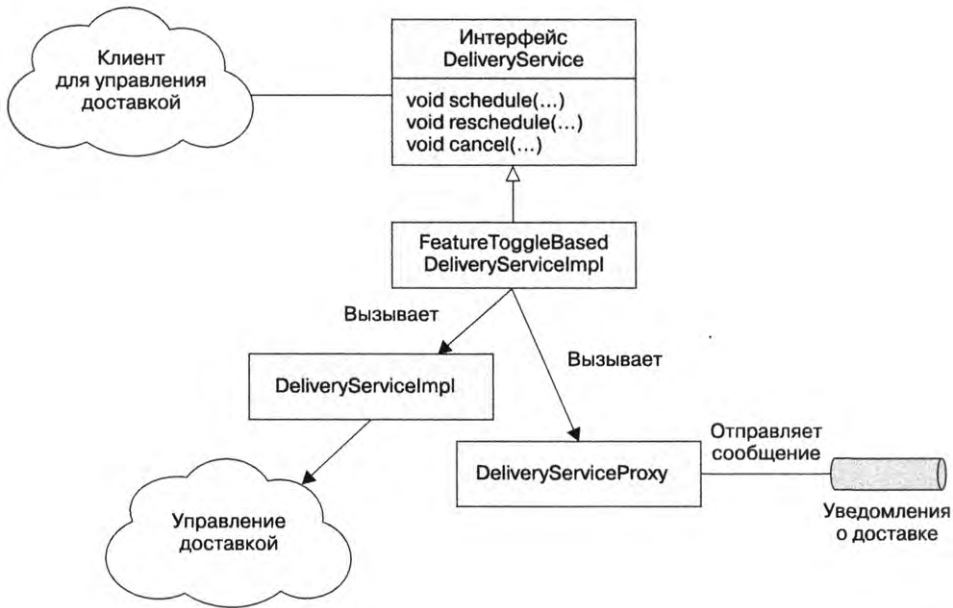


Рис. 13.23. Завершающий шаг — реализация `DeliveryService` с помощью прокси-класса, который шлет сообщения сервису `Delivery`. Переключатель возможностей определяет, что должен вызывать монолит FTGO — старую реализацию или новый сервис `Delivery`

Использование переключателя возможностей значительно снижает риск отката сервиса `Delivery`. Сначала сервис разворачивается и тестируется. Убедившись в том, что он работает, мы можем переключить на него входящий трафик. Если обнаружится, что сервис `Delivery` ведет себя не так, как ожидалось, можно вернуться к старой реализации.

О переключателях возможностей

Переключатели, или флаги, возможностей позволяют развертывать обновления, не делая их доступными пользователям. С их помощью можно также динамически изменять поведение приложения, развертывая новый код. Прекрасный обзор этой темы дается в статье Мартина Фаулера по адресу martinfowler.com/articles/feature-toggles.html.

Убедившись в том, что сервис **Delivery** работает как следует, можно убрать из монолита код для управления доставкой.

Delivery и **Delayed Order** — это примеры сервисов, которые команда FTGO разрабатывает в процессе перехода на микросервисную архитектуру. Последующие шаги будут зависеть от бизнес-приоритетов. Один из вероятных вариантов — извлечение сервиса **Order History**, описанного в главе 7. Это частично устранил необходимость в репликации данных из сервиса **Delivery** обратно в монолит.

После реализации **Order History** команда FTGO может извлекать сервисы в порядке, описанном в подразделе 13.3.2: **Order**, **Consumer**, **Kitchen** и т. д. В ходе этого процесса приложение будет становиться все более простым в обслуживании и тестировании, а темп разработки станет увеличиваться.

Резюме

- ❑ Прежде чем мигрировать на микросервисы, следует убедиться в том, что проблемы с доставкой ПО вызваны недостатками монолитной архитектуры. Иногда доставку можно ускорить за счет внесения корректив в процесс разработки.
- ❑ Переходить на микросервисы следует постепенно, разрабатывая удушающее приложение. Оно состоит из микросервисов, которые вы создаете вокруг существующего монолита. Вы должны как можно раньше и чаще демонстрировать преимущества перехода, чтобы заручиться поддержкой руководства.
- ❑ Отличный способ внедрения микросервисов в вашу архитектуру — реализация новых возможностей в виде сервисов. Это позволяет быстро и просто добавлять новые функции с помощью современных технологий и эффективного процесса разработки. Вы сможете быстро продемонстрировать выгоду от миграции на микросервисы.
- ❑ Чтобы разбить монолит, можно отделить уровень представления от внутренних компонентов, в результате чего получатся два монолита поменьше. Это не очень существенное улучшение, но оно позволит развертывать каждый монолит по отдельности. Благодаря этому, например, отдельной команде будет проще развивать пользовательский интерфейс, не затрагивая внутреннюю часть приложения.
- ❑ Основной способ разбиения монолита заключается в постепенном переносе его функций в сервисы. В первую очередь стоит сосредоточиться на наиболее

полезных возможностях. Например, вы сможете ускорить процесс разработки, если реализуете в виде сервиса активно развивающийся код.

- ❑ Новым сервисам почти всегда приходится взаимодействовать с монолитом. Например, сервису нужно часто обращаться к данным монолита и его функциям. Монолиту тоже иногда необходим доступ к данным и возможностям сервиса. Чтобы организовать такое взаимодействие, следует создать интеграционный слой, который состоит из входящих и исходящих адаптеров, размещенных в монолите.
- ❑ Чтобы доменная модель монолита не засоряла доменную модель сервиса, интеграционный код должен использовать предохранительный слой, который производит преобразования между этими доменными моделями.
- ❑ Чтобы извлечение сервисов как можно меньше влияло на монолит, в его БД можно реплицировать данные, вынесенные в сервис. Схема монолита остается неизменной, поэтому не нужно вносить существенные правки в его кодovou базу.
- ❑ Разработка сервиса часто требует выполнения повествования, в котором участвует монолит. Однако реализация компенсируемых транзакций, требующих внесения масштабных изменений в данные кода монолита, может вызвать определенные трудности. Поэтому иногда следует тщательно планировать порядок извлечения сервисов, чтобы компенсируемые транзакции не попали в монолит.
- ❑ При переходе на микросервисную архитектуру нужно одновременно поддерживать два механизма безопасности: тот, что уже реализован в монолитном приложении и обычно основан на сеансе, хранящемся в памяти, и систему токенов, использованную в сервисах. К счастью, мы можем просто модифицировать обработчик входа в систему так, чтобы он генерировал cookie с токеном безопасности, затем API-шлюз будет передавать этот токен сервисам.

Крис Ричардсон

Микросервисы. Паттерны разработки и рефакторинга

Перевел с английского *С. Черников*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>С. Давид</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>И. Рощина</i>
Художественный редактор	<i>В. Мостипан</i>
Корректор	<i>Е. Павлович</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России Изготовитель ООО «Прогресс книга»
Место нахождения и фактический адрес 194044, Россия, г. Санкт-Петербург,
Б Сампсониевский пр., д. 29А, пом. 52 Тел.: +78127037373.

Дата изготовления 07.2019 Наименование книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01

Подписано в печать 21.06.19 Формат 70×100/16 Бумага офсетная. Усл. п. л. 43,860. Тираж 1000 Заказ 5232

Отпечатано в АО «Первая Образцовая типография» Филиал «Чеховский Печатный Двор»

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

Сайт www.chpd.ru, E-mail: sales@chpd.ru

тел. 8(499) 270-73-59

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт – гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами. Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF – самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – Анна Титова, (812) 703-73-73, titova@piter.com
Москва – Сергей Клебанов, (495) 234-38-15, klebanov@piter.com

Микросервисы

Крис Ричардсон

Крис Ричардсон является пионером в области микросервисной архитектуры. Познакомьтесь с 44 шаблонами проектирования для декомпозиции сервисов, управления транзакциями, запросов и межсервисного взаимодействия, чтобы научиться разрабатывать и развертывать микросервисные приложения, соответствующие требованиям бизнеса. Здесь рассмотрены новые методики для написания сервисов и объединения их в надежные масштабируемые приложения. Многолетний опыт работы автора с распределенными системами позволил создать уникальное практическое руководство по проектированию, тестированию и развертыванию микросервисных систем.

В этой книге:

- Как (и зачем!) использовать микросервисную архитектуру.
- Стратегии декомпозиции сервисов.
- Управление транзакциями и шаблоны запросов.
- Эффективные стратегии тестирования.
- Шаблоны развертывания, включая контейнеры и бессерверные платформы.

В первую очередь книга адресована разработчикам ПО, знакомым со стандартной архитектурой промышленных приложений. Примеры написаны на Java.

Крис Ричардсон является апологетом языка Java, звездой конференции JavaOne и создателем платформы CloudFoundry.com.

Паттерны разработки и рефакторинга

«Прагматичный
подход к новому
архитектурному
ландшафту».

— Симеон Лейзерзон,
Excelsior Software

«Комплексный взгляд
на задачи, с которыми
сталкиваются команды
разработчиков
при переходе
на микросервисы,
а также их решений,
проверенных
индустрией».

— Тим Мьпп, Lightbend

«Добротное
руководство, которое
ускорит переход
на современную
облачную
архитектуру».

— Джон Гатри, Dell/EMC

«Как освоить
микросервисный
подход и научиться
использовать его
в реальном мире».

— Потито Колучелли,
Bizmatica Econocom



Заказ книг:

тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM

каталог книг и интернет-магазин



instagram.com/piterbooks



youtube.com/ThePiterBooks



vk.com/piterbooks



facebook.com/piterbooks

ISBN: 978-5-4461-0996-8



9 785446 110996 8