O'REILLY®

2-е издание

Создание микросервисов

Сэм Ньюмен

SECOND EDITION

Building Microservices

Designing Fine-Grained Systems

Sam Newman

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Создание микросервисов

2-е издание

Сэм Ньюмен

Санкт-Петербург • Москва • Минск

2023

ББК 32.988.02-018
УДК 004.738.5
Н93

Ньюмен Сэм

Н93 Создание микросервисов. 2-е изд. — СПб.: Питер, 2023. — 624 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1145-9

По мере того как организации переходят от монолитных приложений к небольшим автономным микросервисам, распределенные системы становятся все более детализированными. Второе дополненное издание предлагает целостный взгляд на самые актуальные темы, в которых необходимо разбираться при создании и масштабировании архитектуры микросервисов, а также управлении ею.

Вы познакомитесь с современными решениями для моделирования, интеграции, тестирования, развертывания и мониторинга собственных автономных сервисов. Примеры из реальной жизни показывают, как получить максимальную отдачу от этих архитектур. Книга будет полезна всем — от архитекторов и разработчиков до тестировщиков и специалистов по эксплуатации.

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 O'Reilly. 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 978-1492034025 англ.
Authorized Russian translation of the English edition of Building Microservices 2E, ISBN 9781492034025 © 2021 Sam Newman
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.
ISBN 978-5-4461-1145-9
© Перевод на русский язык ООО «Прогресс книга», 2023
© Издание на русском языке, оформление ООО «Прогресс книга», 2023
© Серия «Бестселлеры O'Reilly», 2023

Краткое содержание

Предисловие ... 20

ЧАСТЬ I. ОСНОВЫ

Глава 1. Что такое микросервисы ... 28

Глава 2. Как моделировать микросервисы ... 61

Глава 3. Разделение монолита на части ... 97

Глава 4. Стили взаимодействия микросервисов ... 113

ЧАСТЬ II. РЕАЛИЗАЦИЯ

Глава 5. Реализация коммуникации микросервисов ... 144

Глава 6. Рабочий поток ... 202

Глава 7. Сборка ... 224

Глава 8. Развертывание ... 247

Глава 9. Тестирование ... 306

Глава 10. От мониторинга к наблюдаемости ... 338

Глава 11. Безопасность ... 379

Глава 12. Отказоустойчивость ... 423

Глава 13. Масштабирование ... 457

ЧАСТЬ III. ЛЮДИ

Глава 14. Пользовательские интерфейсы ... 494

Глава 15. Организационные структуры ... 532

Глава 16. Эволюционный архитектор ... 569

Послесловие: соберем все вместе ... 597

Библиография ... 610

Глоссарий ... 615

Об авторе ... 620

Иллюстрация на обложке ... 621

Оглавление

Предисловие ... 20

Кому стоит прочитать эту книгу ... 20

Почему я написал эту книгу ... 20

Что изменилось с момента выхода первого издания ... 21

Навигация по книге ... 22

Часть I. Основы ... 22

Часть II. Реализация ... 23

Часть III. Люди ... 24

Условные обозначения ... 24

Благодарности ... 25

От издательства ... 26

ЧАСТЬ I. ОСНОВЫ

Глава 1. Что такое микросервисы ... 28

Первый взгляд на микросервисы ... 28

Ключевые понятия микросервисов ... 31

Независимое развертывание ... 31

Моделирование вокруг предметной области бизнеса ... 32

Контроль над ситуацией ... 33

Размер ... 34

Гибкость ... 35

Согласование архитектуры и структуры организации ... 36

Монолит ... 40

Однопроцессный монолит ... 40

Модульный монолит ... 41

Распределенный монолит ... 42

Монолиты и конфликт доставки ... 43

Преимущества монолитов ... 43

Технологии, обеспечивающие развитие ... 44

Агрегирование логов и распределенная трассировка ... 44

Контейнеры и Kubernetes ... 45

Потоковая передача данных ... 46
Публичное облако и бессерверный подход ... 47
Преимущества микросервисов ... 47
Технологическая неоднородность ... 48
Надежность ... 49
Масштабирование ... 49
Простота развертывания ... 51
Согласованность рабочих процессов в организации ... 51
Компонуемость ... 51
Слабые места микросервисов ... 52
Опыт разработчика ... 52
Технологическая перегрузка ... 53
Стоимость ... 53
Отчетность ... 54
Мониторинг и устранение неполадок ... 55
Безопасность ... 55
Тестирование ... 55
Время ожидания ... 56
Согласованность данных ... 56
Стоит ли вам использовать микросервисы? ... 57
Кому микросервисы не подойдут ... 57
Где микросервисы хорошо работают ... 59
Резюме ... 60

Глава 2. Как моделировать микросервисы ... 61
Представляем MusicCorp ... 61
Что делает границу микросервиса качественной ... 62
Скрытие информации ... 62
Связность ... 64
Связанность ... 64
Взаимодействие связанности и связности ... 65
Типы связанности ... 65
Предметная связанность ... 67
Сквозная связанность ... 69
Общая связанность ... 72
Связанность по содержимому ... 76
Немного предметно-ориентированного проектирования ... 77
Единый язык ... 78
Агрегат ... 79

Ограниченный контекст ... 82
Сопоставление агрегатов и ограниченных контекстов с микросервисами ... 85
Метод Event Storming ... 86
Аргументы в пользу предметно-ориентированного проектирования микросервисов ... 88
Альтернативы границам предметной области бизнеса ... 89
Волатильность ... 89
Данные ... 90
Технологии ... 92
Организационный подход ... 93
Смешивание моделей и исключений ... 95
Резюме ... 96

Глава 3. Разделение монолита на части ... 97
Осознайте цель ... 97
Постепенный переход ... 98
Монолит не всегда плохой вариант ... 99
Опасность преждевременной декомпозиции ... 99
Что отделить в первую очередь ... 100
Декомпозиция по слоям ... 102
Сначала код ... 102
Сначала данные ... 103
Полезные шаблоны декомпозиции ... 104
Шаблон «Душитель» ... 104
Параллельное выполнение ... 105
Шаблон переключаемых функций ... 105
Проблемы декомпозиции данных ... 106
Производительность ... 106
Целостность данных ... 109
Транзакции ... 109
Инструментарий ... 110
База данных отчетов ... 110
Резюме ... 112

Глава 4. Стили взаимодействия микросервисов ... 113
От внутрипроцессного к межпроцессному ... 113
Производительность ... 114
Изменение интерфейсов ... 115
Обработка ошибок ... 115

Технология межпроцессного взаимодействия: так много вариантов выбора ... 117
Стили взаимодействия микросервисов ... 118
Смешивание и сочетание ... 119
Шаблон: синхронная блокировка ... 119
Преимущества ... 120
Недостатки ... 120
Где использовать ... 121
Шаблон: асинхронная неблокирующая связь ... 123
Преимущества ... 123
Недостатки ... 124
Где использовать ... 125
Шаблон: связь через общие данные ... 126
Реализация ... 126
Преимущества ... 128
Недостатки ... 128
Где использовать ... 128
Шаблон: связь «запрос — ответ» ... 129
Реализация: синхронная или асинхронная ... 130
Где использовать ... 133
Шаблон: событийное взаимодействие ... 133
Реализация ... 135
Что входит в событие ... 136
Где использовать ... 139
Действуйте с осторожностью ... 140
Резюме ... 142

ЧАСТЬ II. РЕАЛИЗАЦИЯ

Глава 5. Реализация коммуникации микросервисов ... 144
В поисках идеальной технологии ... 144
Упростите обратную совместимость ... 144
Сделайте свой интерфейс выразительным ... 145
Следите за тем, чтобы ваши API не зависели от технологий ... 145
Сделайте свой сервис простым для потребителей ... 145
Скройте детали внутренней реализации ... 146
Выбор технологий ... 146
Удаленные вызовы процедур ... 146
REST ... 151

GraphQL ... 157
Брокеры сообщений ... 159
Форматы сериализации ... 164
Текстовые форматы ... 165
Двоичные форматы ... 165
Схемы ... 166
Структурные и семантические разрывы контрактов ... 167
Стоит ли использовать схемы ... 167
Обработка изменений между микросервисами ... 168
Избегание критических изменений ... 169
Наращивание изменений ... 169
Устойчивое считывание ... 169
Правильные технологии ... 171
Явный интерфейс ... 171
Своевременно выявляйте случайные критические изменения ... 173
Управление критическими изменениями ... 174
Поэтапное развертывание ... 174
Сосуществование несовместимых версий микросервиса ... 174
Эмулируйте старый интерфейс ... 176
Какой подход предпочтителен ... 177
Общественный договор ... 178
Отслеживание использования ... 179
Крайние меры ... 179
DRY и опасности повторного использования кода в мире микросервисов ... 180
Совместное использование кода через библиотеки ... 180
Обнаружение сервиса ... 182
Система доменных имен (DNS) ... 183
Динамические реестры сервисов ... 185
Не забывайте о людях! ... 188
Сервисные сети и API-шлюзы ... 188
API-шлюзы ... 189
Сервисные сети ... 192
А как насчет других протоколов? ... 196
Документирование сервисов ... 196
Явные схемы ... 197
Самоописывающаяся система ... 198
Резюме ... 200

Глава 6. Рабочий поток ... 202
Транзакции базы данных ... 202
Транзакции ACID ... 202
Все еще ACID, но с недостаточной атомарностью? ... 204
Распределенные транзакции — двухфазная фиксация ... 206
Просто скажите «нет» распределенным транзакциям ... 208
Саги ... 209
Режимы сбоя саги ... 211
Реализация саг ... 216
Саги в сравнении с распределенными транзакциями ... 222
Резюме ... 223

Глава 7. Сборка ... 224
Краткое введение в непрерывную интеграцию ... 224
Вы действительно выполняете CI? ... 225
Модели ветвления ... 226
Конвейеры сборки и непрерывная доставка ... 228
Инструментарий ... 230
Компромиссы и среды выполнения ... 231
Создание артефакта ... 231
Сопоставление исходного кода и сборок с микросервисами ... 233
Один гигантский репозиторий — одна гигантская сборка ... 233
Шаблон: один репозиторий на один микросервис (то есть мультирепозиторий) ... 235
Шаблон: монорепозиторий ... 239
Какой подход использовал бы я? ... 245
Резюме ... 246

Глава 8. Развертывание ... 247
От логического к физическому ... 247
Несколько экземпляров ... 248
База данных ... 250
Среды выполнения ... 253
Принципы развертывания микросервисов ... 256
Изолированное выполнение ... 257
Сосредоточьтесь на автоматизации ... 259
Инфраструктура как код ... 260
Управление желаемым состоянием ... 263

Варианты развертывания ... 266
Физические машины ... 267
Виртуальные машины ... 268
Контейнеры ... 270
Контейнеры приложений ... 276
Платформа как услуга (PaaS) ... 277
Функция как услуга (FaaS) ... 278
Какой вариант развертывания подходит именно вам ... 286
Kubernetes и оркестрация контейнеров ... 288
Пример для контейнерной оркестровки ... 288
Упрощенный взгляд на концепции Kubernetes ... 289
Мультиарендность и федерация ... 291
Федерация нативных облачных вычислений ... 294
Платформы и мобильность ... 295
Helm, Operator и CRD ... 296
Knative ... 297
Будущее ... 298
Стоит ли вам его использовать ... 298
Поэтапная доставка ... 299
Отделение развертывания от релиза ... 300
Переходим к поэтапной доставке ... 301
Переключатели функций ... 301
Канареечный релиз ... 302
Параллельное выполнение ... 303
Резюме ... 304

Глава 9. Тестирование ... 306
Типы тестов ... 306
Охват тестирования ... 309
Модульное тестирование ... 310
Сервисное тестирование ... 311
Сквозное тестирование ... 312
Компромиссы ... 313
Внедрение сервисных тестов ... 314
Макетирование или заглушки ... 315
Более самостоятельная сервис-заглушка ... 315
Внедрение (этих хитрых) сквозных тестов ... 316
Хрупкие и flaky-тесты ... 318

Кто пишет эти сквозные тесты 319
Как долго должны выполняться сквозные тесты 322
Великое нагромождение 323
Метаверсия 323
Отсутствие возможности независимого тестирования 324
Следует ли избегать сквозных тестов 325
Контрактное тестирование 325
Заключительное слово 329
Опыт разработчика 329
От предварительного тестирования к эксплуатационному 330
Виды эксплуатационных тестов 331
Среднее время восстановления превышает среднее время между отказами? 332
Кросс-функциональное тестирование 333
Тесты производительности 334
Тесты надежности 336
Резюме 337

Глава 10. От мониторинга к наблюдаемости 338
Сбой, паника и замешательство 338
Один микросервис — один сервер 339
Один микросервис — несколько серверов 340
Несколько микросервисов — несколько серверов 341
Наблюдаемость и мониторинг 342
Столпы наблюдаемости? Не так быстро 343
Строительные блоки для наблюдаемости 344
Агрегация логов 345
Агрегация метрики 355
Распределенная трассировка 358
Все ли у нас в порядке? 361
Оповещение 363
Семантический мониторинг 367
Тестирование в эксплуатации 369
Стандартизация 372
Выбор инструментов 373
Демократичность 373
Простота интеграции 374
Обеспечение контекста 374

Своевременность ... 374
Подходит для вашего масштаба ... 375
Эксперт в машине ... 375
Приступая к работе ... 377
Резюме ... 377

Глава 11. Безопасность ... 379
Основные принципы ... 380
Принцип наименьших привилегий ... 381
Глубокая оборона ... 381
Автоматизация ... 383
Встраивание безопасности в процесс доставки ... 383
Пять функций кибербезопасности ... 384
Выявление ... 385
Защита ... 387
Определение ... 387
Реакция ... 387
Восстановление ... 388
Основы безопасности приложений ... 388
Учетные данные ... 388
Исправление ... 395
Резервное копирование (бэкап) ... 398
Повторная сборка (ребилд) ... 399
Безусловное или нулевое доверие ... 400
Безусловное доверие ... 401
Нулевое доверие ... 401
Доверие — это спектр возможных вариантов ... 402
Защита данных ... 404
Данные в процессе передачи ... 404
Данные в состоянии покоя ... 407
Аутентификация и авторизация ... 410
Аутентификация между сервисами ... 411
Аутентификация человека ... 411
Общие реализации единого входа ... 412
SSO-шлюз ... 413
Детализированная авторизация ... 414

Проблема иерархии полномочий ... 415
Авторизация в вышестоящих элементах, централизованная ... 417
Децентрализованная авторизация ... 417
Веб-токены JSON ... 418
Резюме ... 422

Глава 12. Отказоустойчивость ... 423
Что такое отказоустойчивость ... 423
Надежность ... 424
Восстановление ... 425
Стабильная расширяемость ... 426
Непрерывная адаптивность ... 426
Микросервисная архитектура ... 427
Сбои повсюду ... 427
Слишком много — это сколько? ... 429
Снижение функциональности ... 430
Шаблоны стабильности ... 431
Тайм-ауты ... 434
Повторные попытки ... 436
Переборки ... 437
Автоматические выключатели ... 438
Изоляция ... 441
Избыточность ... 442
Промежуточное ПО ... 443
Идемпотентность ... 443
Распределение рисков ... 445
Теорема CAP ... 446
Жертвуя согласованностью ... 447
Жертвуя доступностью ... 448
Жертвуя устойчивостью к разделению ... 449
AP или CP? ... 449
Это не «все или ничего» ... 450
А теперь реальный мир ... 450
Хаос-инжиниринг ... 451
Игровые дни ... 452
Эксперименты в эксплуатационной среде ... 453
От надежности к запредельному ... 453

Поиск виновных ... 454
Резюме ... 455

Глава 13. Масштабирование ... 457
Четыре оси масштабирования ... 457
Вертикальное масштабирование ... 458
Горизонтальное дублирование ... 461
Разделение данных ... 464
Функциональная декомпозиция ... 469
Сочетание моделей ... 471
Начните с малого ... 472
Кэширование ... 474
Для производительности ... 475
Для масштабирования ... 475
Для надежности ... 476
Где кэшировать ... 476
Аннулирование ... 482
Свежесть или оптимизация ... 488
Отравление кэша: поучительная история ... 488
Автоматическое масштабирование ... 489
Начинаем все сначала ... 491
Резюме ... 492

ЧАСТЬ III. ЛЮДИ

Глава 14. Пользовательские интерфейсы ... 494
К цифровым технологиям ... 495
Модели владения ... 495
Драйверы для специализированных фронтенд-команд ... 497
На пути к потоковым командам ... 498
Обмен специалистами ... 499
Обеспечение согласованности ... 501
Решение технических проблем ... 502
Шаблон: монолитный интерфейс ... 503
Когда использовать ... 504
Шаблон: микрофронтенды ... 505
Реализация ... 505
Когда использовать ... 505

Шаблон: декомпозиция на основе страниц ... 507
Где использовать ... 508
Шаблон: декомпозиция на основе виджетов ... 509
Реализация ... 510
Когда использовать ... 513
Ограничения ... 514
Шаблон: центральный объединяющий шлюз ... 515
Владение ... 517
Различные типы пользовательских интерфейсов ... 517
Многочисленные проблемы ... 519
Когда использовать ... 520
Шаблон: бэкенд для фронтенда (BFF) ... 520
Сколько должно быть BFF ... 522
Повторное использование и BFF ... 524
BFF для настольных веб-сайтов и не только ... 527
Когда использовать ... 529
GraphQL ... 529
Гибридный подход ... 531
Резюме ... 531

Глава 15. Организационные структуры ... 532
Слабо связанные организации ... 532
Закон Конвея ... 534
Подтверждение ... 534
Размер команды ... 536
Понимание закона Конвея ... 537
Маленькие команды, большая организация ... 538
Об автономии ... 539
Сильное или коллективное владение ... 541
Сильное владение ... 541
Коллективное владение ... 543
Уровень команд или уровень организации ... 544
Баланс моделей ... 544
Команды поддержки ... 545
Профессиональные сообщества ... 547
Платформа ... 548
Общие микросервисы ... 551
Слишком трудно разделить ... 551

Сквозные изменения ... 552
Узкие места в доставке ... 553
Решение с открытым исходным кодом внутри компании ... 554
Роль доверенных коммиттеров ... 554
Завершенность ... 555
Инструментарий ... 555
Подключаемые модульные микросервисы ... 555
Обзоры изменений ... 558
Осиротевший сервис ... 562
Тематическое исследование: realestate.com.au ... 562
Географическое распределение ... 564
Закон Конвея наоборот ... 566
Люди ... 567
Резюме ... 568

Глава 16. Эволюционный архитектор ... 569
Что в имени твоем? ... 569
Что такое архитектура ПО ... 571
Делая изменения возможными ... 573
Эволюционное видение для архитектора ... 574
Определение границ системы ... 575
Социальная конструкция ... 578
Обитаемость ... 579
Принципиальный подход ... 580
Стратегические цели ... 581
Принципы ... 581
Методы ... 582
Сочетание принципов и методов ... 582
Пример из жизни ... 583
Руководство эволюционной архитектурой ... 584
Архитектура в потоковой организации ... 585
Создание команды ... 588
Требуемый стандарт ... 588
Мониторинг ... 589
Интерфейсы ... 589
Архитектурная безопасность ... 590
Управление и мощеная дорога ... 590
Примеры ... 591

Адаптированный шаблон микросервиса ... 592
Мощеная дорога при масштабировании ... 593
Технический долг ... 594
Обработка исключений ... 595
Резюме ... 595

Послесловие: подведем итог основных тем ... 597
Что такое микросервисы ... 597
Переход к микросервисам ... 598
Стили взаимодействия ... 599
Рабочий поток ... 601
Сборка ... 601
Развертывание ... 602
Тестирование ... 603
Мониторинг и наблюдаемость ... 603
Безопасность ... 604
Отказоустойчивость ... 604
Масштабирование ... 605
Пользовательские интерфейсы ... 606
Организация ... 606
Архитектура ... 607
Дополнительная литература ... 608
Взгляд в будущее ... 608
Заключительные слова ... 609

Библиография ... 610

Глоссарий ... 615

Об авторе ... 620

Иллюстрация на обложке ... 621

Предисловие

Архитектурный стиль микросервисов — это подход к распределенным системам, при котором используются небольшие сервисы, каждый из которых можно изменять, развертывать и выпускать независимо друг от друга. Для организаций, переходящих к менее связанным между собой системам, с автономными командами, предоставляющими функциональность, ориентированную на пользователя, микросервисы значительно повышают эффективность. Помимо этого, они обеспечивают огромное количество вариантов построения систем, дают впечатляющую гибкость, позволяя системе изменяться в соответствии с потребностями пользователей.

Однако микросервисы обладают и существенными недостатками: будучи распределенной системой, они создают множество сложностей, которые могут поставить в тупик даже опытных разработчиков.

В этой книге собраны идеи с конкретными примерами из реальной жизни, которые помогут понять, подходят ли вам микросервисы.

Кому стоит прочитать эту книгу

Область применения данной книги широка, как и возможности микросервисных архитектур. Таким образом, это издание должно понравиться людям, интересующимся аспектами проектирования, разработки, развертывания, тестирования и обслуживания систем. Те из вас, кто уже вступил на путь создания мелкомодульных архитектур для использования в новых проектах или в рамках декомпозиции существующей монолитной системы, найдут здесь множество практических рекомендаций. Это руководство также поможет тем, кто хочет тщательнее разобраться в микросервисах и наконец определиться в необходимости их применения.

Почему я написал эту книгу

С одной стороны, я написал эту книгу, потому что хотел убедиться, что информация в первом издании остается актуальной, точной и полезной. Первое издание появилось, потому что на тот момент были действительно интересные идеи, которыми я мечтал поделиться. С самого начала я писал о микросервисах

с достаточно объективной точки зрения, потому что не работал на крупного поставщика технологий. Я не продавал людям решения и надеялся, что не продавал и микросервисы, — мне просто нравилось разбираться в них и находить способы их более широкого использования.

Откровенно говоря, второе издание я написал по двум причинам. Во-первых, появилось ощущение, что на этот раз смогу сделать работу лучше: я больше узнал и, надеюсь, стал немного лучше как писатель. Во-вторых, я чувствую свою ответственность за популяризацию описанных в книге идей и поэтому хотел попробовать изложить их подробнее. Микросервисы стали для многих архитектурным выбором по умолчанию, который, на мой взгляд, трудно обосновать, поэтому я бы хотел поделиться своим видением причин.

В книге я не настаиваю на повсеместном использовании микросервисов, но и не отговариваю вас от их применения. Для меня важно донести до вас все изученные мной плюсы и минусы данного подхода.

Что изменилось с момента выхода первого издания

Первое издание я писал примерно год, на протяжении 2014 года, а выпущено оно было уже в феврале 2015-го. Это было в самом начале истории микросервисов, по крайней мере с точки зрения понимания этого термина широкими кругами в отрасли. С тех пор микросервисы стали популярны настолько, что я и предположить не мог. Чем популярнее становилась данная отрасль, тем больше появлялось возможностей и технологий для ее реализации.

По мере того как я работал с большим количеством команд после выхода первого издания, я совершенствовал свои знания о микросервисах. Иногда это означало, что идеи, существовавшие только на периферии моего сознания (например, скрытие информации), становились более ясными как основополагающие концепции, требующие более широкого освещения. Иногда новые технологии предоставляют не только новые решения, но и дополнительные сложности. Видя, как много людей стекаются в Kubernetes в надежде, что эта платформа поможет решить все их проблемы с микросервисными архитектурами, я, безусловно, задумался.

Кроме того, я написал первое издание книги «Создание микросервисов», чтобы не только рассказать о микросервисах, но и продемонстрировать, как этот архитектурный подход меняет суть разработки программного обеспечения (ПО). Поэтому, более глубоко изучив вопросы, связанные с безопасностью и отказоустойчивостью, я обнаружил, что хочу подробнее остановиться на тех темах, которые становятся все более важными для современной разработки программного обеспечения.

Таким образом, в этом, втором издании я потратил больше времени на подготовку наглядных примеров. Каждая глава была пересмотрена, и каждое предложение проанализировано. От первого издания осталось не так уж много с точки зрения непосредственно текста, но все идеи сохранились. Я старался излагать свои мысли более ясно, в то же время признавая существование нескольких способов решения проблемы. Это привело к более широкому описанию межпроцессной коммуникации, которое теперь занимает три главы. Я также потратил больше времени на изучение влияния применения таких технологий, как контейнеры, Kubernetes и бессерверные вычисления. В результате теперь есть отдельные главы о сборке и развертывании.

Я надеялся написать книгу примерно того же объема, что и первое издание, и при этом найти способ привести больше деталей. Как видно, мне не удалось достичь своей цели — это издание стало больше! Но я думаю, что смог более четко сформулировать свои идеи.

Навигация по книге

Книгу лучше читать целиком от начала и до конца, но, конечно, можно перейти к конкретным темам, которые вас больше всего интересуют. Если вы все-таки решите углубиться в конкретную главу, возможно, глоссарий в конце книги объяснит вам новые или незнакомые термины. Что касается терминологии, я использую слова «микросервис» и «сервис» взаимозаменяемо на протяжении всего повествования. Считайте, что эти два термина относятся к одному и тому же понятию, если явно не указано иное. Я также резюмирую основные концепции в послесловии, однако учтите, что вы упустите много важных деталей, если просто перейдете в конец книги!

Книга разбита на три отдельные части: «Основы», «Реализация» и «Люди». Давайте рассмотрим, какие вопросы охватывает каждая из них.

Часть I. Основы

В этой части я подробно описываю некоторые ключевые идеи, лежащие в основе микросервисов.

Глава 1 «Что такое микросервисы». Это общее введение в микросервисы, в нем я привожу ряд тем, которые будут подробно описаны позже в книге.

Глава 2 «Как моделировать микросервисы». В этой главе рассматривается важность таких понятий, как скрытие информации, связность и связанность, а также использование предметно-ориентированного проектирования для определения правильных границ ваших микросервисов.

Глава 3 «Разделение монолита на части». Здесь приведены некоторые рекомендации о том, как взять существующее монолитное приложение и разбить его на микросервисы.

Глава 4 «Стили взаимодействия микросервисов». В последней главе этой части мы обсудим различные типы связи микросервисов, включая асинхронные и синхронные вызовы, а также стили взаимодействия «запрос — ответ» и событийную архитектуру.

Часть II. Реализация

Переходя от концепций более высокого уровня к деталям реализации, в этой части мы рассмотрим методы и технологии, которые могут помочь получить максимальную отдачу от микросервисов.

Глава 5 «Реализация коммуникации микросервисов». В этой главе мы подробно рассмотрим конкретные технологии, используемые для реализации взаимодействия между микросервисами.

Глава 6 «Рабочий поток». В ней предлагается сравнение саг и распределенных транзакций и обсуждается их полезность при моделировании бизнес-процессов с использованием нескольких микросервисов.

Глава 7 «Сборка». В этой главе микросервис сопоставляется с репозиториями и сборками.

Глава 8 «Развертывание». В этой главе мы обсудим множество вариантов развертывания микросервиса, в том числе использование контейнеров, Kubernetes и FaaS.

Глава 9 «Тестирование». Здесь обсуждаются проблемы тестирования микросервисов, в том числе проблемы, вызванные сквозными тестами, и то, как могут помочь контракты, ориентированные на потребителя, и продакшен-тестирование.

Глава 10 «От мониторинга к наблюдаемости». В этой главе мы переходим от изучения деятельности по статическому мониторингу к более широкому взгляду на улучшение наблюдаемости микросервисных архитектур. Здесь приводятся некоторые рекомендации относительно инструментария.

Глава 11 «Безопасность». Микросервисные архитектуры создают большую площадь для внешних атак, но также дают нам больше возможностей для глубокой обороны. В этой главе мы рассмотрим правильный баланс между уязвимостью и защитой.

Глава 12 «Отказоустойчивость». В ней предлагается более широкий взгляд на то, что такое отказоустойчивость, и на ту роль, которую микросервисы могут сыграть в повышении отказоустойчивости ваших приложений.

Глава 13 «Масштабирование». В этой главе я описываю четыре оси масштабирования и показываю, как их можно использовать совместно для масштабирования микросервисной архитектуры.

Часть III. Люди

Идеи и технологии ничего не значат без людей и организаций, которые их используют.

Глава 14 «Пользовательские интерфейсы». В этой главе рассматриваются принципы совместной работы микросервисов и пользовательских интерфейсов, начиная с перехода от выделенных команд разработки пользовательского интерфейса (фронтенд-разработки) к использованию BFF и GraphQL.

Глава 15 «Организационные структуры». В предпоследней главе основное внимание уделяется тому, как потоковые команды и команды поддержки могут работать в контексте микросервисных архитектур.

Глава 16 «Эволюционный архитектор». Микросервисные архитектуры не статичны, поэтому вам, возможно, потребуется пересмотреть ваше отношение к системной архитектуре — тема, подробно рассматриваемая в этой главе.

Условные обозначения

В этой книге используются следующие типографические обозначения.

Курсив

Обозначает новые термины и важные понятия.

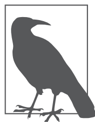
`Моноширинный шрифт`

Используется в листингах кода, а также в тексте, обозначая такие программные элементы, как имена переменных и функций, базы данных, типы данных, значения и ключевые слова.

Рубленый шрифт

Применяется для выделения URL, адресов электронной почты.

Обозначает совет или предложение.



Обозначает примечание общего характера.

Обозначает предупреждение или предостережение.

Благодарности

Я постоянно удивляюсь той поддержке, которую получаю от своей семьи, особенно от моей жены Линди Стивенс. Откровенно говоря, без нее этой книги бы не было. Спасибо ей. Я также благодарю своего отца, Джека, Джози, Кейна и весь клан Джилманко Стейнс.

Большая часть этой книги была написана во время глобальной пандемии, которая все еще продолжается, пока я пишу эти строки. Возможно, это мало что значит, но я хочу выразить свою благодарность Национальной службе здравоохранения Великобритании и всем людям в мире, которые обеспечивают нашу безопасность, работая над вакцинами, излечивая больных, доставляя нам еду и помогая тысячью различных способов, которые мне не так очевидны. Эта благодарность также предназначается всем вам.

Второго издания не было бы без первого, поэтому я хотел бы еще раз сказать спасибо всем, кто помогал мне в сложном процессе написания моей первой книги, включая технических рецензентов Бена Кристенсена, Мартина Фаулера, Венката Субраманьяма, Джеймса Льюиса за наши многочисленные поучительные беседы, команду O'Reilly в лице Брайана Макдональда, Рэйчел Монаган, Кристен Браун и Бетси Валишевски, и за отличные отзывы читателей Ананда Кришнасвами, Кента Макнила, Чарльза Хейнса, Криса Форда, Эйди Льюиса, Уилла Темза, Джона Ивса, Рольфа Рассела, Бадринатха Янакирамана, Дэниела Брайанта, Иэна Робинсона, Джима Уэббера, Стюарта Глидоу, Эвана Боттчера, Эрика Суорда и Оливию Леонард. И спасибо Майку Лукидесу, я думаю, за то, что он втянул меня в эту неразбериху в первую очередь!

Для второго издания Мартин Фаулер снова вернулся в качестве научного редактора, и к нему присоединились Дэниел Брайант и Сара Уэллс, которые не пожалели своего времени на отзывы. Я также хотел бы поблагодарить Ники Райтсон и Александра фон Цитцервитца за помощь в доведении технического обзора до конца. Что касается O'Reilly, то весь процесс контролировался моим потрясающим редактором Николь Таше, без которой я бы точно сошел с ума, Мелиссой Даффилд, которая, похоже, справляется с моей рабочей нагрузкой лучше, чем я. Благодарю также Деба Бейкера, Артура Джонсона и остальную производственную команду (мне жаль, что я не знаю всех ваших имен, но спасибо вам!), а также Мэри Трезелер за то, что в трудные времена брала штурвал в свои руки.

Кроме того, огромное спасибо за неоценимую помощь ряду людей, в том числе (в произвольном порядке) Дэйву Кумбсу и команде Tyro, Дэйву Хэлси и команде Money Supermarket, Тому Керхову, Эрике Доерненбург, Грэму Тэкли, Кенту Бек, Кевлин Хенни, Лоре Белл, Адриане Муат, Саре Тарапоревалла, Уве Фридрихсу, Лиз Фонг-Джонс, Кейну Стивенсу, Гилманко Стейнсу, Адаму Торнхиллу, Венкату Субраманьяму, Сюзанне Кайзер, Яне Шауманн, Грейди Бучу, Пини Резник, Николь Форсгрен, Джезу Хамблу, Джин Ким, Мануэлю Паис,

Мэтью Скелтону и команде «Саут Сидней Рэббитоуз». Наконец, я хотел бы поблагодарить восхищенных читателей ранней версии книги, предоставивших бесценные отзывы. Среди них Фелипе де Мораис, Марк Гарднер, Дэвид Лозон, Ассам Зафар, Майкл Блетерман, Никола Мусатти, Элеонора Лестер, Фелипе де Мораис, Натан Димауро, Даниэль Лемке, Сонер Экер, Риппл Шах, Джоэл Лим и Химаншу Пант. И наконец, привет Джейсону Айзексу.

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.

ЧАСТЬ I

Основы

ГЛАВА 1

Что такое микросервисы

С тех пор как я написал первое издание этой книги, прошло более пяти лет, и все это время микросервисы становились все более и более популярным архитектурным решением. Последовавший бум популярности этой технологии не моя заслуга, но взрывной рост микросервисных архитектур связан с тем, что пока новые идеи проверялись на практике, устаревшие методы становились все менее удобными. Итак, пришло время еще раз раскрыть суть архитектуры микросервисов, выделив при этом основные концепции, заставляющие микросервисы работать.

Основная цель книги — показать, как микросервисы влияют на различные аспекты поставки программного обеспечения. Для начала мы рассмотрим ключевые идеи, лежащие в основе микросервисов, предшествующий уровень развития техники и причины, по которым эти архитектуры так широко используются.

Первый взгляд на микросервисы

Микросервисы — это независимо выпускаемые сервисы, которые моделируются вокруг предметной области бизнеса. Сервис инкапсулирует функциональность и делает ее доступной для других сервисов через сети — вы создаете более сложную, комплексную систему из этих строительных блоков. Один микросервис может представлять складские запасы, другой — управление заказами и еще один — доставку, но вместе они могут составлять целую систему онлайн-продаж. Микросервисы — это пример архитектуры, где есть возможность выбирать из множества вариантов решения проблем, с которыми вы сталкиваетесь.

Они представляют собой *тип* сервис-ориентированной архитектуры (хотя и с особым пониманием того, как следует проводить границы сервисов), в которой ключевым фактором выступает возможность независимого развертывания. Они не зависят от технологий, что является одним из преимуществ.

Снаружи отдельный микросервис рассматривается как черный ящик. Он размещает бизнес-функции в одной или нескольких конечных точках сети (например, в очереди или REST API, как показано на рис. 1.1) по любым наиболее

подходящим протоколам. Потребители, будь то другие микросервисы или иные виды программ, получают доступ к этой функциональности через такие точки. Внутренние детали реализации (например, технология, по которой был создан сервис, или способ хранения данных) полностью скрыты от внешнего мира. Это означает, что в микросервисных архитектурах в большинстве случаев не используются общие базы данных. Вместо этого каждый микросервис инкапсулирует свою собственную БД там, где это необходимо.

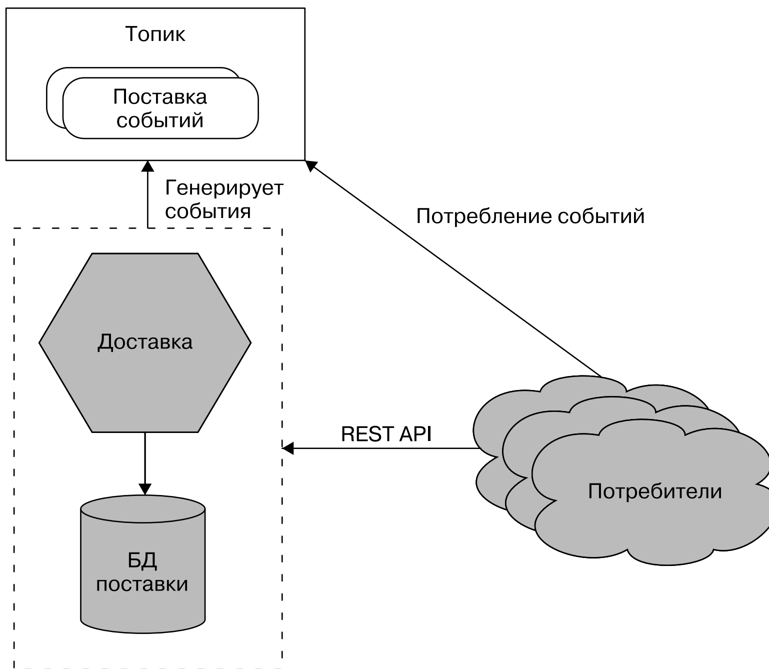


Рис. 1.1. Микросервис, предоставляющий свои функциональные возможности через REST API и топик

Микросервисы используют концепцию *скрытия информации*[1]. Это означает скрытие как можно большего количества информации внутри компонента и как можно меньшее ее раскрытие через внешние интерфейсы. Так можно провести четкую границу между легко и сложно изменяемыми данными. Реализацию, скрытую от сторонних участников процесса, можно свободно преобразовывать, пока у сетевых интерфейсов, предоставляемых микросервисом, сохраняется

[1] *Parnas D.* Information Distribution Aspects of Design Methodology (https://oreil.ly/rDPWA) // Information Processing: Proceedings of the IFIP Congress 1971. — Amsterdam: North-Holland, 1972. — P. 339–344.

обратная совместимость. Изменения внутри границ микросервиса (как показано на рис. 1.1) не должны влиять на вышестоящего потребителя, обеспечивая возможность независимого выпуска функциональных возможностей. Это необходимо для того, чтобы микросервисы могли работать изолированно и выпускаться по требованию. Наличие четких, стабильных границ сервисов, не изменяющихся при преобразовании внутренней реализации, приводит к тому, что системы получают более слабую связанность (coupling) и более сильную связность (cohesion).

Пока мы говорим о скрытии деталей внутренней реализации, с моей стороны было бы упущением не упомянуть шаблон *гексагональной архитектуры*, впервые подробно описанный Алистером Кокберном[1]. Этот шаблон определяет важность сохранения внутренней реализации отдельно от ее внешних интерфейсов, поскольку вы, возможно, захотите взаимодействовать с одной и той же функциональностью через разные типы интерфейсов. Я изображаю свои микросервисы в виде шестиугольников (гексагонов) отчасти для того, чтобы отличить их от «обычных» сервисов, но также по причине моей любви к этим фигурам.

СЕРВИС-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА И МИКРОСЕРВИСЫ — РАЗНЫЕ ВЕЩИ?

Сервис-ориентированная архитектура (SOA, service-oriented architecture) — это подход к проектированию, при котором несколько сервисов взаимодействуют для обеспечения определенного конечного набора возможностей (сервис здесь обычно означает полностью отдельный процесс операционной системы). Связь между этими сервисами осуществляется посредством сетевых вызовов, а не с помощью вызовов методов внутри границ процесса.

SOA возникла как подход к решению проблем, связанных с большими монолитными приложениями. Данный подход направлен на поощрение повторного использования ПО. Например, два приложения или более могут использовать одни и те же сервисы. SOA стремится упростить обслуживание или переписывание ПО, поскольку теоретически мы можем заменить один сервис другим без чьего-либо ведома, если семантика сервиса не изменится слишком сильно.

По своей сути SOA — разумная идея. Однако, несмотря на многочисленные усилия, отсутствует единый стандарт *правильной* реализации SOA. На мой взгляд, нет целостного взгляда на проблему, поэтому не получилось прийти к единому мнению, с которым были бы согласны все представители отрасли.

Многие из проблем, лежащих в основе SOA, на самом деле относятся к проблемам с протоколами связи (например, SOAP), промежуточным ПО поставщика, отсутствием рекомендаций по детализации сервиса или неправильным руководством по выбору мест для разделения вашей системы. Циник мог бы

[1] *Cockburn A.* Hexagonal Architecture, 4 января 2005 года. https://oreil.ly/NfvTP.

предположить, что поставщики навязывали (а в некоторых случаях и стимулировали) внедрение SOA как способ продать больше продуктов, и эти самые продукты в конечном счете подорвали цель SOA.

Я видел множество примеров SOA, в которых команды стремились сделать сервисы меньше, но эти сервисы все еще были связаны с базой данных, и приходилось развертывать все вместе. Сервис-ориентированно? Да. Но это не микросервисы.

Микросервисный подход появился благодаря накопленному практическому опыту успешных реализаций SOA и лучшему пониманию систем и архитектуры. Вы должны воспринимать микросервисы как специфический подход к SOA. Это то же самое, что и экстремальное программирование (XP, Extreme Programming) или Scrum — особый подход к гибкой разработке ПО.

Ключевые понятия микросервисов

При изучении микросервисов необходимо усвоить несколько ключевых идей. Учитывая, что некоторые детали часто упускаются из виду, важно продолжить изучение этих концепций, чтобы убедиться, что вы понимаете, что именно заставляет микросервисы работать.

Независимое развертывание

Возможность *независимого развертывания* — это идея о том, что мы можем внести изменения в микросервис, развернуть его и предоставить это изменение нашим пользователям без необходимости развертывания каких-либо других микросервисов. Важно не только то, что мы можем это сделать, но и то, что *именно* так вы управляете развертываниями в своей системе. Подходите к идее независимого развертывания как к чему-то обязательному. Это простая для понимания, но довольно сложная в реализации концепция.

Основная мысль, которую я хочу донести, звучит так: убедитесь, что вы придерживаетесь концепции независимого развертывания ваших микросервисов. Заведите привычку развертывать и выпускать изменения в одном микросервисе в готовом ПО без необходимости развертывания чего-либо еще. Это будет полезно.

Чтобы иметь возможность независимого развертывания, нам нужно убедиться, что наши микросервисы *слабо связаны*, то есть обеспечена возможность изменять один сервис без необходимости изменять что-либо еще. Это означает, что нужны явные, четко определенные и стабильные контракты между сервисами. Некоторые варианты реализации (например, совместное использование баз данных) затрудняют эту задачу.

Возможность независимого развертывания сама по себе, безусловно, невероятно ценна, но, чтобы добиться ее, вам предстоит решить множество других задач, которые, в свою очередь, имеют свои уникальные преимущества. Таким образом, можно рассматривать возможность независимого развертывания как принудительную функцию. Сосредоточив внимание на ней как на результате, вы получите ряд дополнительных преимуществ.

Стремление к слабо связанным сервисам со стабильными интерфейсами заставляет задуматься об определении границ микросервисов.

Моделирование вокруг предметной области бизнеса

Такие методы, как предметно-ориентированное проектирование, могут позволить структурировать код, чтобы лучше представлять реальную область, в которой работает программное обеспечение[1]. В микросервисных архитектурах мы используем ту же идею для определения границ сервисов. Моделируя сервисы вокруг предметных областей бизнеса, можно упростить внедрение новых функций и процесс комбинирования микросервисов для предоставления новых функциональных возможностей нашим пользователям.

Развертывание функции, требующей внесения изменений более чем в один микросервис, обходится дорого. Вам придется координировать работу каждого сервиса (и, возможно, отдельных команд) и тщательно отслеживать порядок развертывания новых версий этих сервисов. Это потребует гораздо большего объема работ, чем внесение таких же преобразований внутри одного сервиса (или внутри монолита). Следовательно, нужно найти способы сделать межсервисные изменения как можно более редкими.

Я часто вижу многоуровневые архитектуры, типичный пример которых представлен на рис. 1.2. Здесь каждый уровень определяет отдельную границу обслуживания, причем каждая из них относится к соответствующей технической функциональности. Если бы в этом примере я вносил изменения только в уровень представления, это было бы довольно эффективно. Однако опыт показывает, что изменения в функциональности обычно охватывают несколько уровней, что требует изменений в представлении, приложениях и уровне данных. Эта проблема усугубляется, если архитектура еще более многоуровневая, чем в простом примере на рис. 1.2. Часто каждый уровень разбит на дополнительные слои.

Наши сервисы выполнены в виде сквозных срезов бизнес-функциональности. Такая архитектура гарантирует, что вносимые изменения будут максимально эффективными. Можно утверждать, что в случае с микросервисами мы

[1] Более подробно о предметно-ориентированном проектировании рассказано в книге Эрика Эванса «Предметно-ориентированное проектирование», а более кратко — в книге «Дистиллированное предметно-ориентированное проектирование» Вона Вернона.

отдаем приоритет сильной связности бизнес-функциональности, а не технической функциональности.

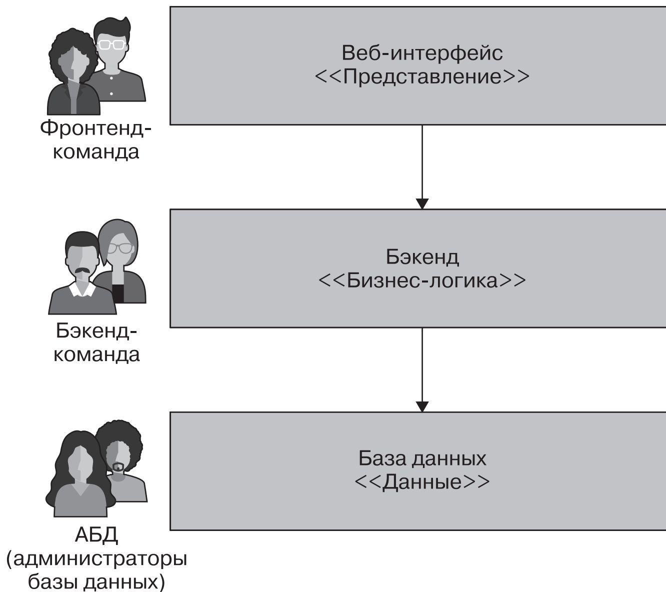


Рис. 1.2. Традиционная трехуровневая архитектура

Позже в этой главе мы еще вернемся к предметно-ориентированному проектированию и к тому, как оно взаимодействует с организационным проектированием.

Контроль над ситуацией

Одна из самых непривычных рекомендаций при использовании микросервисной архитектуры состоит в том, что необходимо избегать использования общих баз данных. Если микросервис хочет получить доступ к данным, хранящимся в другом микросервисе, он должен напрямую запросить их у него. Благодаря такому подходу микросервисы могут определять, что является общим, а что — скрытым. Это позволяет нам четко отделять свободно изменяемую (наша внутренняя реализация) функциональность от функциональности, которую не требуется часто преобразовывать (внешний контракт, используемый потребителями информации).

Если мы хотим реализовать независимое развертывание, нужно убедиться, что есть ограничения на обратно несовместимые изменения в микросервисах. Если нарушить совместимость с вышестоящими потребителями, это неизбежно повлечет за собой необходимость внесения изменений и в них тоже. Четкое разграничение между внутренними деталями реализации и внешним контрактом

для микросервиса может помочь уменьшить потребность в обратно несовместимых преобразованиях.

Скрытие внутреннего состояния в микросервисе аналогично практике инкапсуляции в объектно-ориентированном (ОО) программировании. Инкапсуляция данных в ОО-системах представляет собой пример скрытия информации в действии.

Не используйте базы данных совместно без крайней нужды. И даже при необходимости старайтесь избегать этого. На мой взгляд, совместное использование баз данных — одна из худших идей, которые вы можете реализовать при попытке добиться независимого развертывания.

Как обсуждалось в предыдущем разделе, необходимо рассматривать наши сервисы как сквозные срезы бизнес-функциональности, которые, где это уместно, инкапсулируют пользовательский интерфейс (user interface, UI), бизнес-логику и данные. Это связано с желанием прикладывать как можно меньше усилий, необходимых для изменения бизнес-функциональности. Инкапсуляция данных и подобное поведение обеспечивают сильную связность бизнес-функций. Скрывая поддерживающую сервис БД, мы также обеспечиваем ослабление связанности. Мы вернемся к связанности и связности в главе 2.

Размер

«Насколько большим должен быть микросервис?» — один из самых распространенных вопросов, которые я слышу. Учитывая, что часть «микро» присутствует прямо в названии, ответ однозначный. Однако, когда вы поймете, что из себя представляет микросервис как архитектура, размер перестанет быть одной из наиболее интересующих характеристик.

Как узнать размер? Сосчитав строки кода? Для меня это не имеет особого смысла. Задача, требующая 25 строк кода на Java, может быть написана в десяти строках Clojure. Это не значит, что Clojure лучше или хуже Java. Некоторые языки просто более выразительны, чем другие.

Джеймс Льюис, технический директор Thoughtworks, известен своим высказыванием: «Микросервис должен быть размером с мою голову». На первый взгляд, это выражение кажется бессмысленным. В конце концов, насколько велика голова Джеймса на самом деле? Однако суть этого утверждения в том, что микросервис должен быть такого размера, при котором его можно легко понять. Проблема, конечно, заключается в том, что разные люди могут неодинаково понимать какую-либо информацию, поэтому на вопрос, какой размер подходит именно вам, можете ответить только вы. Более опытная команда лучше справится с управлением крупной кодовой базы, чем любая другая. Так

что, возможно, было бы правильнее интерпретировать цитату Джеймса как «микросервис должен быть размером с *вашу* голову».

Крис Ричардсон, автор книги «Микросервисы. Паттерны разработки и рефакторинга»[1], говорит, что цель микросервисов — получить «как можно меньший интерфейс». Это снова согласуется с концепцией скрытия информации, но представляет собой попытку найти смысл в термине «микросервисы», которого изначально не было. Когда этот термин впервые использовался для определения архитектур, основное внимание, по крайней мере на начальном этапе, уделялось не размеру интерфейсов.

В конечном счете понятие размера в значительной степени зависит от контекста. Поговорите с человеком, который работал над системой в течение 15 лет, и он скажет, что по его ощущениям система со 100 000 строк кода действительно проста для понимания, в то время как недавно привлеченному к проекту сотруднику покажется, что она слишком масштабна. Аналогично и в компаниях, только что приступивших к переходу на микросервисы и создавших, возможно, десять или меньше элементов логики, по сравнению с компанией, для которой микросервисы были нормой в течение многих лет, и сейчас их в ней сотни.

Я призываю людей не беспокоиться о размере. В самом начале работы гораздо важнее сосредоточиться на двух ключевых моментах. Во-первых, сколько микросервисов вы можете обработать? По мере увеличения количества сервисов сложность вашей системы будет возрастать, и вам потребуется осваивать новые навыки (и, возможно, внедрять новые технологии), чтобы справиться с этим. Переход на микросервисы приведет к появлению дополнительных источников сложности со всеми вытекающими из этого проблемами. Именно по этой причине я являюсь убежденным сторонником постепенного перехода на микросервисную архитектуру. Во-вторых, как максимально эффективно определить границы микросервисов, не создавая при этом хаос в системе? Именно на этих темах гораздо важнее сосредоточиться, когда вы в начале пути.

Гибкость

Еще одна цитата Джеймса Льюиса гласит: «Приобретая микросервисы, вы покупаете себе новые возможности». Льюис преднамеренно употребил словосочетание *«покупаете возможности»*. У микросервисов есть своя цена, и вы должны самостоятельно решить, стоит ли игра свеч. Результирующая гибкость по целому ряду направлений — организационному, техническому, масштабированию, надежности — может быть невероятно привлекательной.

Мы не знаем, что ждет нас в будущем, поэтому нужна архитектура, теоретически способная помочь решить любые возможные проблемы. Нахождение

[1] *Ричардсон К.* Микросервисы. Паттерны разработки и рефакторинга. — Питер, 2019.

баланса между сохранением открытых возможностей и затратами на подобную архитектуру может быть настоящим искусством.

Внедрение микросервисов не происходит по щелчку пальцев. Это постепенный процесс, по мере реализации которого повышается гибкость. Но, скорее всего, так же увеличивается и количество слабых мест. Это еще одна причина, по которой я решительно выступаю за последовательное внедрение микросервисов, потому что только так вы сможете объективно оценить их воздействие и при необходимости остановиться.

Согласование архитектуры и структуры организации

MusicCorp — компания, торгующая компакт-дисками онлайн. В ее системе использована простая трехуровневая архитектура (как на рис. 1.2). Мы решили перенести сопротивляющуюся современным тенденциям MusicCorp в XXI век и в рамках этого перемещения оцениваем существующую системную архитектуру. У нас есть веб-интерфейс, уровень бизнес-логики в виде монолитного бэкенда и хранилище данных в виде традиционной БД. За эти слои, как обычно бывает, отвечают разные команды. Мы будем возвращаться к примеру MusicCorp на протяжении всей книги.

Хочется внести простое обновление в функциональность: мы решили позволить клиентам указывать свой любимый жанр музыки. Это обновление требует внесения изменений в пользовательский интерфейс (для выбора жанра), в бэкенд сервиса (чтобы разрешить отображение жанра в пользовательском интерфейсе и иметь возможность менять значение), а также в базу данных (чтобы принять это изменение). Каждая команда должна будет управлять этими изменениями и внедрять их в правильном порядке, как показано на рис. 1.3.

Теперь эта архитектура выглядит получше. Вся она в конечном итоге оптимизируется вокруг набора целей. Трехуровневая архитектура так распространена отчасти потому, что она универсальна — все о ней слышали. Таким образом, тенденция выбирать то, что на слуху, является одной из причин, по которым мы продолжаем повсеместно встречать этот шаблон. Однако я считаю, что основная проблема, из-за которой мы сталкиваемся с подобной архитектурой снова и снова, заключается в привычных для нас принципах формирования рабочих команд.

Ныне известный закон Конвея гласит следующее.

> Организации, разрабатывающие системы... создают проекты, представляющие собой копии коммуникационных структур этих организаций.
>
> *Мелвин Конвей, Как комитеты изобретают? (Melvin Conway, How Do Committees Invent?, https://oreil.ly/NhE86)*

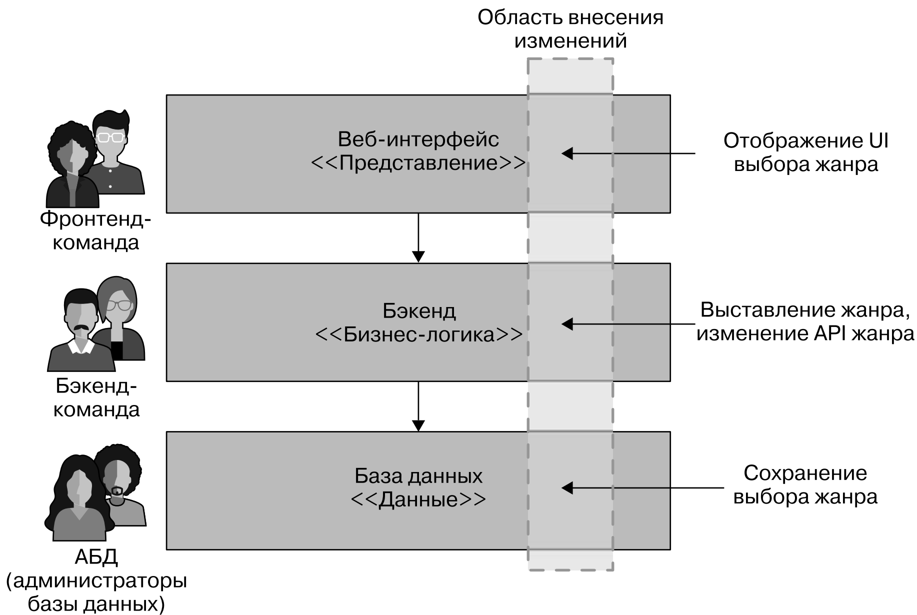


Рис. 1.3. Внесение изменений на всех трех уровнях требует больших усилий

Трехуровневая архитектура представляет собой хороший пример этого закона в действии. В прошлом ИТ-организации в основном группировали работников по их ключевым компетенциям: администраторы баз данных были в команде с другими администраторами БД, разработчики Java — с другими разработчиками Java, а фронтенд-разработчики (которые сегодня знают такие экзотические вещи, как JavaScript, и занимаются разработкой собственных мобильных приложений) — в еще одной такой же команде. Мы объединяем сотрудников на основе их базовых компетенций, поэтому создаем ИТ-ресурсы, адаптированные к этим командам.

Это объясняет, почему рассматриваемая архитектура так популярна. Она не плохая, просто такой способ группировки людей сосредоточен вокруг принципа «мы всегда так делали». Но времена меняются, а вместе с ними и наши устремления в отношении разрабатываемого программного обеспечения. Теперь мы объединяем в команды людей с различной квалификацией, чтобы сократить количество передаваемых друг другу функций и число случаев разрозненности данных. Сейчас требуется поставлять ПО гораздо быстрее, чем когда-либо прежде. Это заставляет нас по-другому формировать наши команды с точки зрения разделения системы на части.

Большинство изменений, которые нас просят внести в систему, связаны с преобразованиями в бизнес-функциональности. Но на рис. 1.3 бизнес-функциональность фактически распределена по всем трем уровням, что

увеличивает вероятность того, что изменения затронут всю систему. Эта архитектура с сильной связностью технологий, но слабой связностью бизнес-функциональности. Если мы хотим упростить процесс внесения изменений, нужно пересмотреть способ группировки кода, выбрав связность бизнес-функциональности, а не технологий. Каждый сервис может как содержать, так и не содержать эти три уровня, но это проблема его локальной реализации.

Сравним это с потенциальной альтернативной архитектурой, проиллюстрированной на рис. 1.4. Вместо горизонтальной многоуровневой архитектуры и организации — вертикальная. Здесь мы видим отдельную команду, которая несет полную сквозную ответственность за внесение изменений в профиль клиента. Это гарантирует, что объем преобразований в этом примере будет ограничен одной командой.

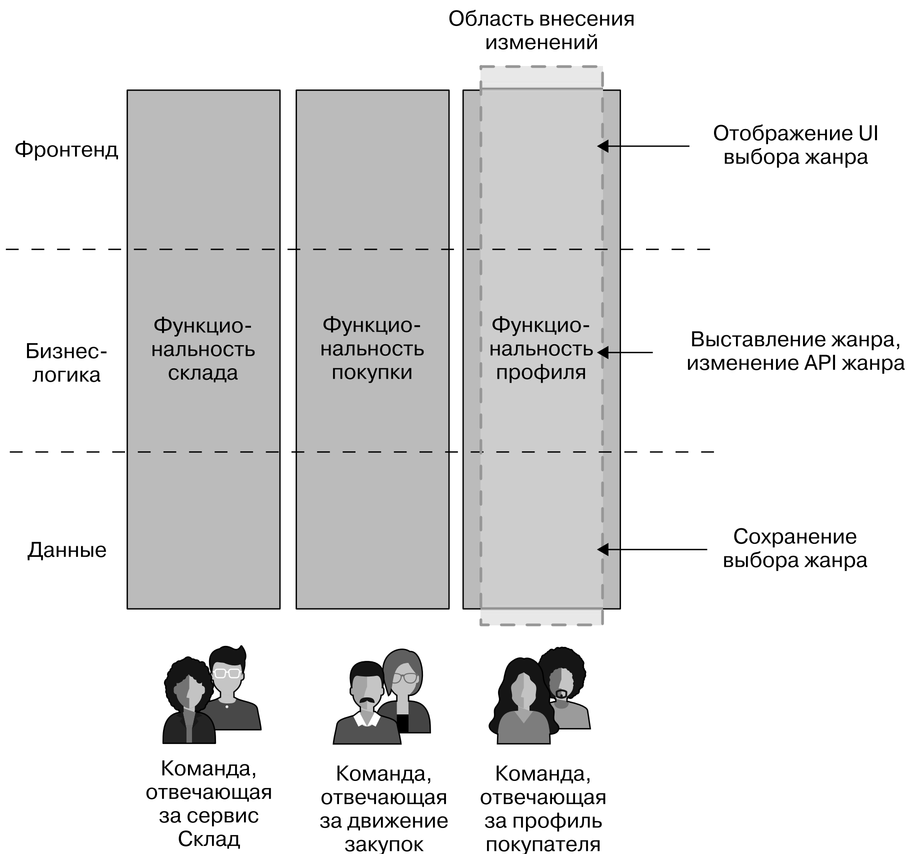


Рис. 1.4. Пользовательский интерфейс разделен на части и управляется командой, которая также управляет функциональностью, поддерживающей UI, но на стороне сервера

Поставленная задача может быть достигнута с помощью единого микросервиса, принадлежащего команде по работе с профилями и предоставляющего UI. Он позволит клиентам обновлять свою информацию, сохраняя при этом их данные. Выбор предпочтительного жанра связан с конкретным клиентом, поэтому такое изменение гораздо более локализовано. На рис. 1.5 также показан список доступных жанров, получаемых из микросервиса `Каталог`, который, вероятно, уже существует. Мы также видим новый микросервис `Рекомендация`, предоставляющий доступ к информации о наших любимых жанрах.

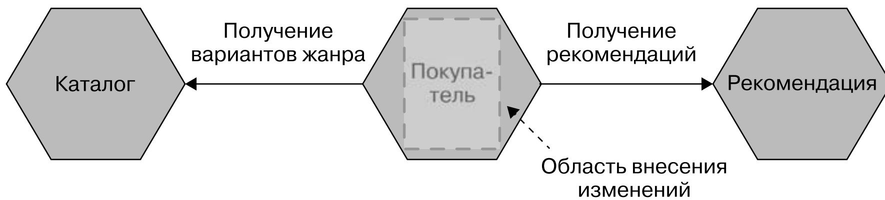


Рис. 1.5. Специализированный микросервис Покупатель может значительно упростить запись любимого музыкального жанра для покупателя

Наш микросервис `Покупатель` инкапсулирует тонкий срез каждого из трех уровней: в нем есть немного UI, немного логики приложения и немного хранилища данных. Наша предметная область бизнеса становится основной движущей силой системной архитектуры, что, как мы надеемся, облегчает внесение изменений и согласование между командами и бизнес-отделами внутри организации.

Часто UI не предоставляется непосредственно микросервисом, но даже если это так, мы ожидаем, что часть его, связанная с этой функциональностью, по-прежнему будет принадлежать команде, отвечающей за профиль клиента, как показано на рис. 1.4. Концепция создавать команды, владеющие сквозным набором функций, ориентированных на пользователя, набирает обороты. В книге «Топологии команд»[1] представлена идея потоковой команды, которая воплощает эту концепцию.

> Потоковая команда — это команда, ориентированная на отдельный, представляющий значение поток работы... Команда уполномочена создавать и предоставлять ценности для клиентов или пользователей максимально быстро, безопасно и независимо, не требуя передачи другим командам части работы для выполнения.

Команды, показанные на рис. 1.4, были бы потоковыми. Эту концепцию мы рассмотрим более подробно в главах 14 и 15, в том числе обсудим, как эти типы организационной структуры работают на практике, и поговорим о степени их соответствия микросервисам.

---

[1] *Skelton M., Pais M.* Team Topologies. — Portland, OR: IT Revolution, 2019.

ЗАМЕТКА О «ВЫДУМАННЫХ» КОМПАНИЯХ

На протяжении всей книги мы будем встречаться с MusicCorp, FinanceCo, FoodCo, AdvertCo и PaymentCo.

FoodCo, AdvertCo и PaymentCo — это реальные компании, названия которых я изменил по соображениям конфиденциальности. Кроме того, делясь информацией об этих компаниях, я всегда стремился избавиться от посторонних, не несущих практической ценности деталей без ущерба для основного повествования, чтобы внести больше ясности.

С другой стороны, MusicCorp — это собирательный образ многих организаций, с которыми я работал. Истории, рассказанные мной о MusicCorp, представляют собой отражение реальных событий, свидетелем которых я был, но не все они произошли с одной и той же компанией!

Монолит

Поговорим о микросервисах как об архитектурном подходе, представляющем собой альтернативу монолитной архитектуре. Чтобы более четко понять микросервисную архитектуру и помочь вам разобраться, стоит ли в принципе рассматривать микросервисы, необходимо также пояснить, что именно я подразумеваю под *монолитами*.

Говоря о монолитах, я в первую очередь имею в виду единицы развертывания. Когда все функциональные возможности в системе должны развертываться вместе, я считаю ее монолитом. Возможно, под это определение подходит несколько архитектур, но я собираюсь обсудить те, которые встречаются чаще всего: однопроцессный, модульный и распределенный монолиты.

Однопроцессный монолит

Наиболее распространенный пример, который приходит на ум при обсуждении монолитов, — это система, в которой весь код развертывается как *единый процесс* (рис. 1.6). Может существовать несколько экземпляров этого процесса (из соображений надежности или масштабирования), но в реальности весь код упакован в один процесс. В действительности эти однопроцессные системы сами по себе могут быть простыми распределенными системами, поскольку они почти всегда завершаются считыванием данных из базы данных или их сохранением, или представлением информации мобильным или веб-приложениям.

Хотя большинство людей именно так и представляют классический монолит, основная масса встречающихся систем несколько сложнее. Может существовать два или более тесно связанных друг с другом монолита, возможно, с использованием программного обеспечения какого-либо производителя.

Классическое однопроцессное монолитное развертывание может иметь смысл для многих компаний. Дэвид Хайнемайер Хансон, создатель фреймворка Ruby on Rails, доказал, что такая архитектура имеет смысл для небольших организаций[1]. Однако даже по мере роста компании монолит потенциально может расти вместе с ней, что подводит нас к модульному монолиту.

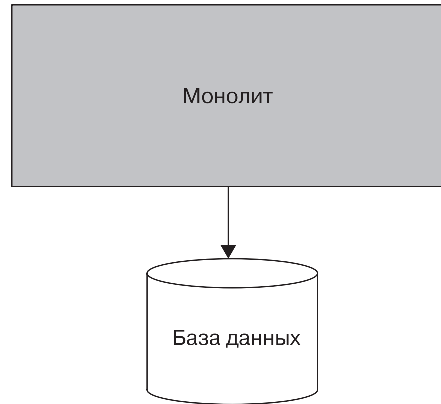


Рис. 1.6. В однопроцессном монолите весь код упакован в один процесс

Модульный монолит

Модульный монолит, являясь подмножеством однопроцессного, представляет собой систему, в которой один процесс состоит из отдельных модулей. С каждым модулем можно работать независимо, но для развертывания их все равно необходимо объединить, как показано на рис. 1.7. Концепция разбиения ПО на модули не нова — модульное ПО появилось благодаря работе, проделанной в области структурного программирования в 1970-х годах и даже раньше. Тем не менее я все еще не вижу достаточного количества организаций, которые должным образом используют этот подход.

Для многих компаний модульный монолит может стать отличным выбором архитектуры. Если границы модулей четко определены, это может обеспечить высокую степень параллельной работы и отсутствие проблем, связанных с более распределенной микросервисной архитектурой, благодаря упрощенной топологии развертывания. Shopify — отличный пример организации, в которой этот метод успешно реализован в качестве альтернативы декомпозиции микросервисов[2].

Одна из проблем модульного монолита заключается в том, что базе данных, как правило, не хватает декомпозиции, которую мы находим на уровне кода, что приводит к значительным проблемам, если потребуется разобрать монолит в будущем. Я видел, как некоторые команды пытались развить идею модульного монолита дальше, разделив БД по границам разбиения модулей, как показано на рис. 1.8.

---

[1] *Hansson D. H.* The Majestic Monolith // Signal v. Noise, 29 февраля 2016 года. https://oreil.ly/WwG1C.

[2] Для получения полезной информации о том, как Shopify использует модульный монолит, а не микросервисы, см.: *Westeinde K.* Deconstructing the Monolith.

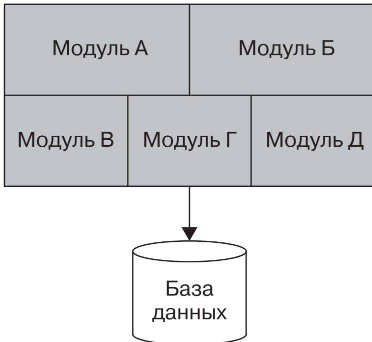


Рис. 1.7. В модульном монолите код процесса разделен на модули

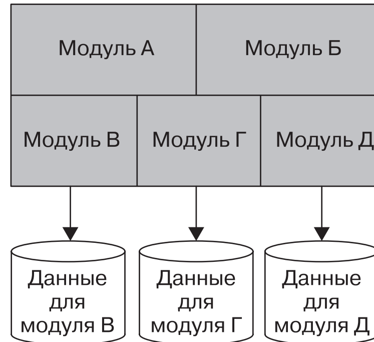


Рис. 1.8. Модульный монолит с разделенной базой данных

Распределенный монолит

> Распределенная система — это система, в которой сбой компьютера, о существовании которого вы даже не подозревали, может привести к сбою в работе вашего собственного компьютера[1].
>
> *Лесли Лэмпорт*

Распределенный монолит — это состоящая из нескольких сервисов система, которая должна быть развернута единовременно. Распределенный монолит вполне подходит под определение SOA, однако он не всегда соответствует требованиям SOA.

По моему опыту, распределенный монолит обладает всеми недостатками распределенной системы и однопроцессного монолита, не имея при этом достаточного количества преимуществ ни того ни другого. Поработав с несколькими распределенными монолитами, я и заинтересовался микросервисной архитектурой.

Рассматриваемые системы обычно формируются в среде, в которой недостаточно внимания уделялось таким понятиям, как скрытие информации и связность бизнес-функций. Вместо этого сильно связанные архитектуры приводят к тому, что изменения распространяются через границы сервисов и, казалось бы, незначительные локальные преобразования нарушают другие части системы.

[1] Лесли Лэмпорт, сообщение по электронной почте (https://oreil.ly/2nHF1) на доску объявлений DEC SRC в 12:23:29 PDT, 28 мая 1987 года.

Монолиты и конфликт доставки

Когда над одним проектом работает слишком много людей, неизбежен конфликт интересов. Например, одни разработчики хотят изменить определенный фрагмент кода, а другие хотят запустить функциональность, за которую они отвечают (или отложить развертывание). На этом этапе становится непонятно, кто за что отвечает и кто принимает решения. В результате множества исследований были выявлены проблемы, связанные с границами владения[1]. Я называю их *конфликтом доставки*.

Использование монолита не означает, что вы обязательно столкнетесь с конфликтом доставки, равно как и наличие микросервисной архитектуры не гарантирует, что на вас никогда не свалится эта проблема. Но микросервисная архитектура действительно предлагает более конкретные границы, по которым можно определить «зоны влияния» в системе, что дает гораздо больше гибкости.

Преимущества монолитов

Некоторые монолиты, такие как модульные и однопроцессные, обладают целым рядом преимуществ. Их гораздо более простая топология развертывания позволяет избежать многих ошибок, связанных с распределенными системами. Это может привести к значительному упрощению рабочих процессов, мониторинга, устранения неполадок и сквозного тестирования.

Для разработчика монолиты также могут упростить повторное использование кода внутри самого монолита. Если необходимо вторично использовать код в распределенной системе, то потребуется решить, стоит ли копировать код, разбивать библиотеки или внедрять общие функции в сервис. С монолитом все намного проще: весь код уже есть и готов к использованию — и многим это нравится!

К сожалению, люди стали относиться к монолиту как к чему-то, чего следует избегать, — чему-то изначально проблематичному. Я встречал множество разработчиков, для которых термин «монолит» стал синонимом слова «устаревший» — и это проблема. Монолитная архитектура — это решение, и притом обоснованное. Более того, на мой взгляд, это стандартный и разумный выбор архитектурного стиля. Другими словами, назовите мне убедительную причину использовать микросервисы.

Попадая в ловушку систематического отрицания монолита как работающего варианта доставки ПО, мы рискуем поступить неправильно по отношению к себе или к пользователям нашего программного продукта.

---

[1] Microsoft Research провела исследования в этой области, и я рекомендую узнать о них подробнее, но в качестве отправной точки предлагаю статью Don't Touch My Code! Examining the Effects of Ownership on Software Quality за авторством Christian Bird (https://oreil.ly/0ahXX).

Технологии, обеспечивающие развитие

Как я уже говорил, поначалу вам не стоит внедрять много новых технологий — это может быть контрпродуктивно. По мере расширения микросервисной архитектуры лучше сосредоточиться на поиске проблем, вызванных распределением системы, а уже затем на технологиях для их решения.

Тем не менее технологии сыграли большую роль в принятии микросервисов как концепции. Понимание доступных инструментов позволит вам получить максимальную отдачу от этой архитектуры и станет ключевым фактором успеха любой реализации микросервисов. В целом я бы сказал, что микросервисы требуют четкого понимания различий между логической и физической архитектурой.

Мы подробно поговорим об этом в последующих главах, но для начала давайте кратко представим некоторые вспомогательные технологии, способные помочь при использовании микросервисов.

Агрегирование логов и распределенная трассировка

С увеличением числа процессов, которыми вы управляете, становится сложнее понять, как система поведет себя в эксплуатации, что значительно затруднит устранение неполадок. Более детально об этом поговорим в главе 10, но как минимум я настоятельно рекомендую внедрить систему агрегирования логов (журналов) в качестве предварительного условия для реализации микросервисной архитектуры.

Будьте осторожны, не используйте слишком много новых технологий в начале работы с микросервисами. Тем не менее инструмент агрегации логов настолько важен, что стоит рассматривать его как обязательное условие для внедрения микросервисов.

Такие системы позволяют собирать и объединять логи из всех ваших сервисов, предоставляя возможности для анализа и включения журналов в активный механизм оповещения. Мне очень нравится сервис ведения логов Humio (https://www.humio.com), однако на первое время вам хватит чего-то попроще из того, что предоставляют основные поставщики публичных облачных сервисов.

Можно сделать эти инструменты агрегирования еще более полезными, внедрив идентификаторы корреляции, где один идентификатор используется для связанного набора вызовов сервисов, например цепочки вызовов, возникающей из-за взаимодействия с пользователем. Если регистрировать этот идентификатор как часть каждой записи журнала, становится намного проще изолировать логи, связанные с данным потоком вызовов, что, в свою очередь, значительно упрощает устранение неполадок.

По мере усложнения системы важно подобрать инструменты, позволяющие лучше изучить, что делает ваша система. Они предоставляют возможность анализировать трассировки между несколькими сервисами, находить узкие места и задавать о вашей системе вопросы, о которых вы даже не подозревали. Инструменты с открытым исходным кодом могут предоставить некоторые из этих функций. Одним из примеров может служить Jaeger (https://www.jaegertracing.io), фокусирующийся на стороне распределенной трассировки уравнения.

Но такие программные продукты, как Lightstep (https://lightstep.com) и Honeycomb (https://honeycomb.io) (показан на рис. 1.9), дают больше возможностей. Они представляют собой новое поколение инструментов, выходящих за рамки традиционных подходов к мониторингу, значительно упрощая изучение состояния работающей системы. Возможно, вы уже пользуетесь чем-то более привычным, но тем не менее я советую обратить внимание на возможности, предоставляемые этими продуктами. Они были созданы с нуля для решения тех проблем, с которыми приходится сталкиваться операторам микросервисных архитектур.

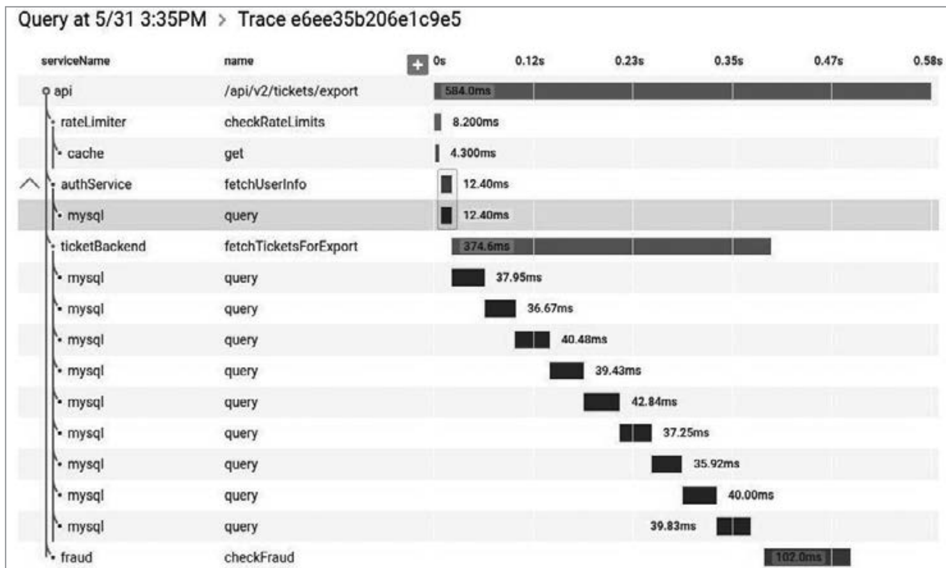


Рис. 1.9. Распределенная трассировка, показанная в Honeycomb, позволяет определить, где тратится время на операции, которые могут затрагивать несколько микросервисов

Контейнеры и Kubernetes

В идеале хотелось бы запускать каждый экземпляр микросервиса изолированно. Это бы гарантировало, что проблема одного микросервиса не будет влиять на работу другого, например, путем поглощения всех ресурсов про-

цессора. Виртуализация — это один из способов создания изолированных сред выполнения на имеющемся оборудовании. Обычные методы виртуализации могут быть довольно затратными, если учесть размер микросервисов. А вот *контейнеры*, напротив, обеспечивают гораздо более легкий способ максимально быстро развернуть новые экземпляры контейнеров, а также повысить рентабельность многих архитектур.

Поэкспериментировав с контейнерами, вы придете к мысли, что для администрирования ими на множестве машин вам потребуется дополнительный инструментарий. Платформы оркестрации контейнеров, такие как Kubernetes, дают возможность распределять экземпляры контейнеров, обеспечивая надежность и пропускную способность, которые необходимы вашему сервису, при этом позволяя эффективно использовать базовые машины. В главе 8 описана концепция операционной изоляции, контейнеров и Kubernetes.

Однако не стоит спешить с внедрением Kubernetes или даже контейнеров. Они, безусловно, предлагают значительные преимущества по сравнению с более традиционными технологиями развертывания, но в них нет особого смысла, если у вас всего несколько микросервисов. После того как накладные расходы на управление развертыванием перерастут в серьезную головную боль, начните рассматривать возможность контейнеризации своего сервиса и использования Kubernetes. И если вы решитесь на этот шаг, сделайте все возможное, чтобы кто-то другой управлял кластером Kubernetes вместо вас, пусть даже и с использованием управляемого сервиса от облачного провайдера. Запуск собственного кластера Kubernetes может потребовать значительного объема работы!

Потоковая передача данных

Микросервисы позволяют отойти от использования монолитных баз данных, однако потребуется найти иные способы обмена данными. Такая необходимость возникает, когда организации хотят отказаться от пакетной отчетности и перейти к более оперативной обратной связи. Поэтому среди людей, применяющих микросервисную архитектуру, стали популярными продукты, дающие возможность легко передавать и обрабатывать внушительные объемы данных.

Для многих людей Apache Kafka (https://kafka.apache.org) — стандартный выбор для потоковой передачи информации в среде микросервисов, и на то есть веские причины. Такие возможности, как постоянство сообщений, сжатие и способность масштабирования для обработки больших объемов сообщений, могут быть невероятно полезными. С появлением Kafka возникла возможность потоковой обработки в виде базы данных ksqlDB, которую также можно использовать с выделенными решениями для потоковой обработки, такими как Apache Flink (https://flink.apache.org).

Debezium (https://debezium.io) — это инструмент с открытым исходным кодом, разработанный для потоковой передачи из существующих источников данных через Kafka. Такой способ передачи гарантирует, что традиционные источники данных могут стать частью потоковой архитектуры. В главе 4 мы рассмотрим, какую роль технология потоковой передачи может сыграть в интеграции микросервисов.

Публичное облако и бессерверный подход

Поставщики (провайдеры) публичных (общедоступных) облачных сервисов, или, точнее, три основных поставщика — Google Cloud, Microsoft Azure и Amazon Web Services (AWS), — предлагают огромный набор управляемых сервисов и вариантов развертывания для управления вашим приложением. По мере роста микросервисной архитектуры работа все больше и больше будет переноситься в эксплуатационное пространство. Провайдеры облаков предоставляют множество услуг: от управляемых экземпляров баз данных или кластеров Kubernetes до брокеров сообщений или распределенных файловых систем. Используя эти управляемые сервисы, вы перекладываете большой объем работы на третью сторону, которая, возможно, лучше справится с такими задачами.

Особый интерес среди облачных предложений представляют программные продукты, которые позиционируются как *бессерверные*. Они скрывают базовые машины, позволяя вам работать на более высоком уровне абстракции. Примерами программных продуктов с бессерверной стратегией могут служить брокеры сообщений, решения для хранения данных и БД. Платформы «функция как услуга» (FaaS, function as a service) представляют особый интерес, поскольку обеспечивают хорошую абстракцию вокруг развертывания кода. Не беспокоясь о том, сколько серверов требуется для запуска вашего сервиса, вы просто развертываете код и позволяете базовой платформе запускать его экземпляры по требованию. Мы рассмотрим бессерверный подход более подробно в главе 8.

Преимущества микросервисов

У микросервисов множество разнообразных преимуществ (некоторые из них могут быть заложены в основу любой распределенной системы), которые достигаются прежде всего благодаря независимой позиции в отношении определения границ сервисов. Сочетая концепции скрытия информации и предметно-ориентированного проектирования с возможностями распределенных систем, микросервисы по своим возможностям могут уйти далеко вперед по сравнению с другими формами распределенных архитектур.

Технологическая неоднородность

В системе, состоящей из нескольких взаимодействующих микросервисов, могут быть использованы разные технологии для каждого отдельного микросервиса. Так можно выбирать правильный инструмент для конкретной задачи вместо того, чтобы искать более стандартизированный, универсальный подход, который часто приводит к наименьшей отдаче.

Для повышения производительности одной части системы можно использовать другой технологический стек, более подходящий для достижения требуемого уровня эффективности. Может также измениться способ хранения данных в разных частях нашей системы. Например, для социальной сети оптимально хранить взаимодействия пользователей в графовой БД, чтобы отразить сильно взаимосвязанный характер социального графа. А сообщения, которые оставляют пользователи, хранить в документо-ориентированном хранилище данных, используя гетерогенную архитектуру (рис. 1.10).

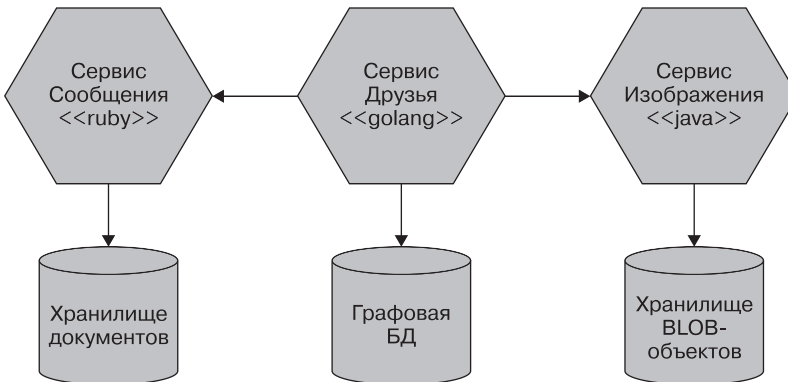


Рис. 1.10. Микросервисы могут упростить использование различных технологий

Микросервисы не только помогут быстрее интегрировать технологии, но и позволят оценить их пользу. Одно из самых больших препятствий на пути внедрения новой технологии — связанные с ее использованием риски. В монолитном приложении, если возникнет желание попробовать новый язык программирования, базу данных или фреймворк, любое изменение повлияет на большую часть системы. В системе, состоящей из сервисов, существует несколько мест, где можно опробовать новую технологию. Появляется возможность выбрать микросервис с наименьшим, по вашему мнению, риском и реализовать технологию там, зная, что любое потенциальное негативное воздействие может быть ограничено. Быстрое внедрение новых идей — реальное преимущество, по мнению многих организаций.

Конечно, использование различных технологий не обходится без накладных расходов. Некоторые компании предпочитают устанавливать определенные ограничения на выбор языка. Например, Netflix и Twitter в основном в качестве платформы используют виртуальную машину Java (JVM, Java Virtual Machine) из-за ее надежности и производительности. Они также разрабатывают библиотеки и инструменты для JVM, которые действительно очень полезны. Но зависимость от них усложняет работу сервисов или клиентов, не базирующихся на Java. Однако ни Twitter, ни Netflix не используют только один технологический стек для всех задач.

Тот факт, что внутренняя реализация технологий скрыта от потребителей, также может облегчить их модернизацию. Например, если вся ваша микросервисная архитектура основана на Spring Boot, то вы можете изменить версию JVM или фреймворка только для одного микросервиса, что упрощает управление рисками обновлений.

Надежность

Основной концепцией повышения надежности вашего приложения является переборка (bulkhead). Вовремя обнаруженный вышедший из строя компонент системы можно изолировать, при этом остальная часть системы сохранит работоспособность. Границы сервисов становятся потенциальными переборками. В монолитной системе, если сервис выходит из строя, перестает функционировать все. В ней мы можем работать на нескольких машинах, чтобы снизить вероятность сбоя, а с микросервисами появляется возможность создавать системы, способные справляться с полным отказом некоторых сервисов и соответствующим образом понижать производительность.

Однако следует быть осторожными. Чтобы гарантировать такую надежность, нужно проанализировать новые источники сбоев, с которыми приходится сталкиваться распределенным системам. Сети могут и будут выходить из строя, как и машины. Необходимо знать, как справляться с такими сбоями и какое влияние (если таковое имеется) они окажут на конечных пользователей. В свое время я работал с командами, которые из-за недостаточно серьезного подхода к такого рода проблемам после перехода на микросервисы получили менее стабильную систему.

Масштабирование

С массивным монолитным сервисом масштабировать придется все целиком. Допустим, одна небольшая часть общей системы ограничена в производительности, и если это поведение заложено в основе гигантского монолитного

приложения, нам потребуется масштабировать всю систему как единое целое. С помощью сервисов поменьше можно масштабировать только ту часть программы, для которой это необходимо. Это позволяет запускать другие части системы на меньшем и менее мощном оборудовании, как показано на рис. 1.11.

Интернет-магазин модной одежды Gilt внедрил микросервисы именно по этой причине. В 2007 году компания начала с монолитного приложения Rails, но к 2009 году система Gilt не смогла справиться с нагрузкой. Разделив основные части своей системы, компания сумела решить проблему со скачками трафика, и сегодня в этой системе более 450 микросервисов, каждый из которых работает на нескольких отдельных машинах.

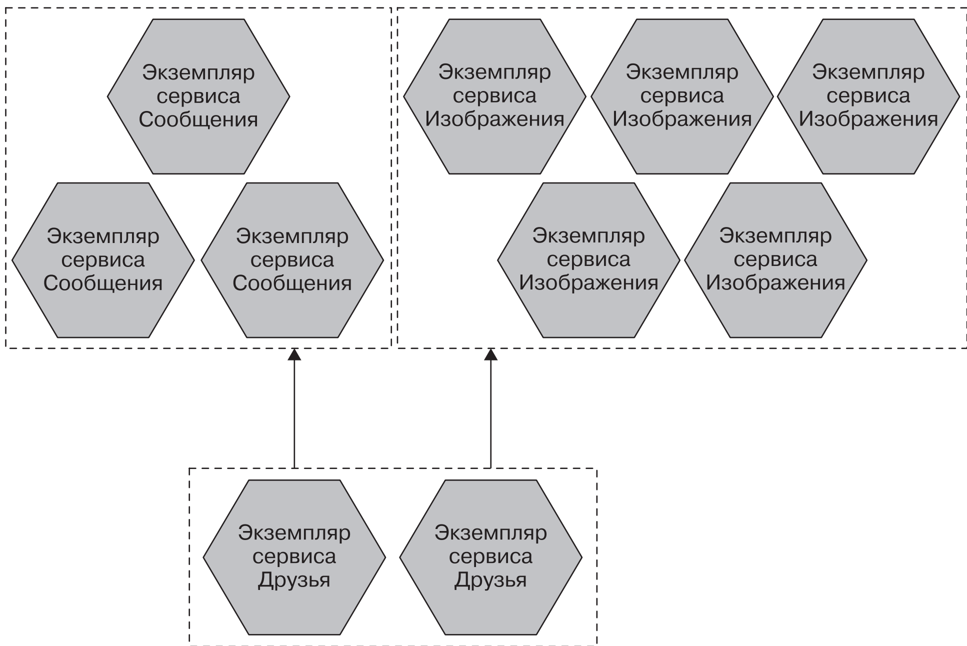


Рис. 1.11. Можно сделать целью масштабирования только те микросервисы, которые в нем нуждаются

В системах обеспечения по требованию, похожих на те, что предоставляет AWS, мы можем использовать масштабирование по требованию только для тех элементов, которые в нем нуждаются. Это позволяет более эффективно контролировать расходы. Нечасто архитектурный подход бывает тесно связан с экономией средств.

В конечном счете можно масштабировать свои приложения множеством способов, и микросервисы — эффективная часть этого процесса. Мы рассмотрим масштабирование микросервисов более подробно в главе 13.

Простота развертывания

Изменение одной строки в монолитном приложении на миллион строк требует развертывания всего приложения для реализации новой сборки. Подобные события происходят нечасто из-за высоких рисков и страха серьезных последствий. К сожалению, это означает, что изменения накапливаются до момента выхода новой версии приложения. И чем больше разница между релизами, тем выше риск того, что что-то пойдет не так!

Микросервисы же позволяют внести изменения в один сервис и развернуть его независимо от остальной части системы. Если проблема все же возникает, ее можно за короткое время изолировать и откатиться к предыдущему состоянию. Такие возможности помогают быстрее внедрить новые функции в действующее приложение. Именно поэтому такие организации, как Amazon и Netflix, используют аналогичные архитектуры.

Согласованность рабочих процессов в организации

Многие из нас сталкивались с трудностями, которые возникают из-за слишком больших рабочих команд и раздутого кода. Эти проблемы могут усугубиться при рассредоточении команды. Известно, что маленькие коллективы, работающие над небольшими кодовыми базами, как правило, более продуктивны.

Микросервисы помогают оптимизировать количество людей, работающих над какой-либо одной кодовой базой, чтобы достичь оптимального соотношения размера команды и эффективности разработки, и дают возможность переопределять ответственность за сервисы по мере изменения организации. Это позволяет поддерживать согласованность между архитектурой и организацией в будущем.

Компонуемость

Одной из ключевых особенностей распределенных систем и сервис-ориентированных архитектур является возможность повторного применения функциональности. Микросервисы позволяют использовать функциональные возможности по-разному для разных целей. Это может быть особенно важно, когда мы думаем о том, как потребители эксплуатируют наше ПО.

Прошли те времена, когда для взаимодействия с пользователем достаточно было только веб-сайта для ПК или мобильного приложения. Теперь требуется мультиплатформенность. По мере того как организации переходят от мышления в терминах узких каналов коммуникации с пользователями к более целостным концепциям привлечения клиентов, появляется необходимость в архитектурах, способных идти в ногу со временем.

Представьте, что с микросервисами мы стираем границы между нашей системой и внешней стороной. По мере изменения обстоятельств мы можем собирать приложения различными способами. В монолите же обычно есть один крупный канал, с которым можно взаимодействовать со стороны. И если я захочу разделить его на части, чтобы получить что-то более полезное, мне понадобится кувалда!

Слабые места микросервисов

Как вы уже поняли, микросервисные архитектуры дают множество преимуществ. Но их использование влечет за собой и определенные сложности. Если вы рассматриваете возможность внедрения такой архитектуры, важно, чтобы у вас было полное представление о ней. Большинство проблем, связанных с микросервисами, можно отнести к распределенным системам, поэтому они с такой же вероятностью проявятся как в распределенном монолите, так и в микросервисной архитектуре.

Более подробно об этом мы поговорим в других главах. На самом деле я бы сказал, что большая часть книги посвящена преодолению всех возможных трудностей, связанных с применением микросервисной архитектуры.

Опыт разработчика

По мере того как у вас появляется все больше и больше сервисов, отношение разработчиков к ним может начать ухудшаться. Более ресурсоемкие среды выполнения (например, JVM) способны ограничить количество микросервисов, допущенных к запуску на одной машине. Вероятно, я мог бы запустить четыре или пять микросервисов на основе JVM как отдельные процессы на своем ноутбуке, но могу ли я запустить 10 или 20? Скорее всего, нет. Даже при меньшем времени выполнения будет ограничение на количество процессов, доступных для локального запуска. Это неизбежно приведет к невозможности запустить всю систему на одной машине. Ситуация осложнится еще больше при использовании облачных сервисов, которые нельзя запустить локально.

Экстремальные решения могут включать разработку в облаке, когда у программистов больше нет возможности заниматься разработкой локально. Я не поклонник такого решения, потому что циклы обратной связи могут сильно пострадать. Я думаю, что более простой подход — ограничить объем фрагментов, над которыми будет трудиться специалист. Однако это может быть проблематично, если появится желание использовать модель «коллективного владения». Она подразумевает, что любой разработчик может работать над любой частью системы.

Технологическая перегрузка

Огромное количество новых технологий, появившихся для внедрения микросервисных архитектур, ошеломляет. Откровенно говоря, многие из них были просто переименованы в «совместимые с микросервисами», но некоторые достижения действительно помогли справиться со сложностью такого рода архитектур. Однако существует опасность, что это изобилие новых игрушек может привести к определенной форме технологического фетишизма. Я видел очень много компаний, внедряющих архитектуру микросервисов, которые решили, что это еще и самое удачное время для введения огромного количества новых и непонятных технологий.

Архитектура микросервисов вполне может предоставить *возможность* написать каждый микросервис на отдельном языке программирования, выполнять его в своей среде или использовать отдельную базу данных, но это возможности, а не требования. Необходимо найти баланс масштабности и сложности используемой вами технологии с издержками, которые она может повлечь.

В самом начале внедрения микросервисов неизбежны некоторые фундаментальные проблемы: затраты времени на понимание проблем, связанных с согласованностью данных, задержкой, моделированием сервисов и т. п. Если вы возьметесь все это изучать во время освоения огромного количества новых технологий, вам придется нелегко. Также стоит отметить, что, пытаясь понять все эти новые технологии, вы отнимете у себя время, доступное для фактической реализации проекта.

Старайтесь внедрять новые сервисы в свою микросервисную архитектуру по мере необходимости. Нет нужды в кластере Kubernetes, когда у вас в системе всего три сервиса! Вы не только не будете перегружены сложностью этих инструментов, но и получите больше вариантов решения задач, которые, несомненно, появятся со временем.

Стоимость

Весьма вероятно, что в краткосрочной перспективе вы увидите увеличение затрат из-за ряда факторов. Во-первых, вам, скорее всего, потребуется запускать больше процессов, больше компьютеров, сетей, хранилищ и вспомогательного программного обеспечения (а это дополнительные лицензионные сборы).

Во-вторых, любые изменения, вносимые в команду или организацию, замедлят процесс работы. Чтобы изучить новые идеи и понять, как их эффективно использовать, потребуется время. Пока вы будете заняты этим, иные задачи никуда не денутся. Это приведет либо к прямому замедлению рабочего процесса, либо к необходимости добавить больше сотрудников, чтобы компенсировать эти затраты.

По моему опыту, микросервисы — плохой выбор для организации, в первую очередь озабоченной снижением издержек, поскольку тактика сокращения затрат, когда сфера ИТ рассматривается как центр расходов, а не доходов, постоянно будет мешать получить максимальную отдачу от этой архитектуры. С другой стороны, микросервисы могут принести больше прибыли, если вы сможете использовать их для привлечения новых клиентов или для параллельной разработки дополнительных функциональных возможностей. Итак, помогут ли микросервисы разбогатеть? Возможно. Способны ли микросервисы снизить затраты? Не уверен.

Отчетность

В монолитной системе обычно присутствует и монолитная БД. Это означает, что при анализе всех данных единовременно, часто с использованием крупных операций объединения данных, стейкхолдеры применяют готовую схему для создания отчетов. Они могут просто запускать инструменты формирования отчетности непосредственно в монолитной базе данных, возможно в копии для считывания, как показано на рис. 1.12.

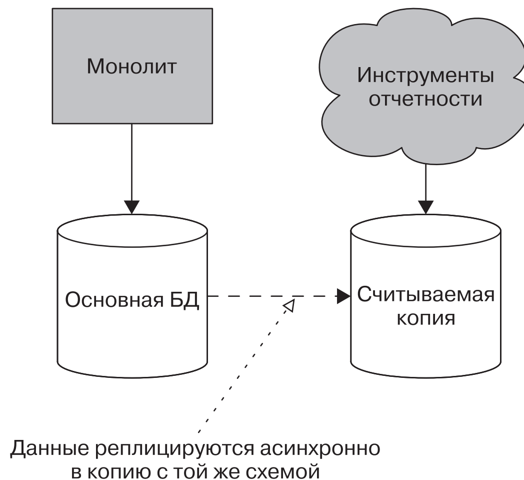


Рис. 1.12. Отчетность ведется непосредственно по базе данных монолита

С помощью архитектуры микросервисов мы разрушили эту монолитную схему. Это не означает, что необходимость в отчетности по всем данным отпала. Она просто значительно усложнится, потому что теперь данные разбросаны по нескольким логически изолированным схемам.

Более современные подходы к отчетности, такие как потоковая передача данных в режиме реального времени, отлично взаимодействуют с микросервисной архитектурой, однако требуют внедрения новых идей и связанных с ними

технологий. В качестве альтернативы можно просто публиковать данные из ваших микросервисов в централизованных базах данных отчетов (или, возможно, в менее структурированных озерах данных).

Мониторинг и устранение неполадок

В стандартном монолитном приложении допускается использовать довольно упрощенный подход к мониторингу. У нас есть небольшое количество машин, о которых необходимо позаботиться, а режим сбоя приложения в некотором роде бинарный — приложение обычно либо работает, либо нет. Понимаем ли мы последствия для архитектуры микросервисов, если выйдет из строя хотя бы один экземпляр сервиса?

В монолите, если центральный процессор (ЦП) долгое время зависает при 100%-ной нагрузке, становится ясно: в системе большая проблема. Можно ли сказать то же самое о микросервисной архитектуре с десятками или сотнями процессов? Стоит ли будить кого-то в 3 часа ночи, когда один процесс застрял на 100%-ной нагрузке ЦП?

К счастью, в этом случае есть огромное пространство для маневров. Если вы хотите изучить эту концепцию более подробно, то в качестве отправной точки рекомендую книгу *Distributed Systems Observability* за авторством Cindy Sridharan (O'Reilly), хотя мы тоже еще рассмотрим мониторинг и наблюдаемость в главе 10.

Безопасность

В однопроцессной монолитной системе большая часть информации протекала в рамках этого же процесса. Теперь больше информации передается по сетям между нашими сервисами. Это может повлечь утечку данных при передаче, а также позволит манипулировать ими в рамках кибератак типа «человек посередине». Это означает, что вам, возможно, потребуется уделять больше внимания защите передаваемых данных, чтобы только авторизованные лица имели к ним доступ. Глава 11 полностью посвящена рассмотрению проблем в этой области.

Тестирование

При любом типе автоматизированного функционального тестирования нужно соблюдать тонкий баланс. Чем больше функций покрывает тест, то есть чем шире его охват, тем больше уверенности в стабильности работы приложения. С другой стороны, увеличение объема теста влечет за собой сложности в настройке тестовых данных и вспомогательных приспособлений, затраты времени для его выполнения вырастут, а определить, что нарушено при возникновении сбоя, станет затруднительно. В главе 9 я поделюсь рядом методов, позволяющих эффективнее проводить тестирование.

Сквозные тесты для любого типа систем находятся на конце шкалы с точки зрения функциональности, которую они охватывают. Их сложнее писать и поддерживать, чем модульные тесты с меньшим объемом. Однако цель оправдывает средства: процесс сквозного тестирования максимально приближен к пользовательскому опыту.

С микросервисной архитектурой объем таких тестов становится *очень* внушительным. Теперь потребуется запускать тесты в нескольких процессах, и все они должны быть развернуты и соответствующим образом настроены для тестовых сценариев. Также необходимо быть готовыми к ложным срабатываниям, возникающим, когда проблемы с окружающей средой, такие как отказ экземпляров сервисов или сетевые тайм-ауты неудачных развертываний, приводят к сбою тестов.

Эти факторы означают, что по мере роста архитектуры микросервисов вы получите меньшую отдачу от инвестиций, когда дело дойдет до сквозного тестирования. Оно будет стоить дороже, но не сможет дать ту же степень уверенности, что и раньше. Это подтолкнет вас к новым формам тестирования, таким как контрактное тестирование или тестирование в эксплуатации, а также к изучению поэтапных методов доставки, таких как параллельное выполнение или канареечные релизы, которые мы рассмотрим в главе 8.

Время ожидания

Благодаря микросервисной архитектуре обработку, которую ранее можно было выполнять только локально на одном процессоре, теперь можно разделить на несколько отдельных микросервисов. Информацию, ранее передаваемую только в рамках одного процесса, теперь необходимо сериализовать, передавать и десериализовать по сетям, которые вы, возможно, используете больше прежнего. Все это может привести к увеличению времени ожидания в вашей системе.

На этапе проектирования или кодирования может быть трудно измерить точное влияние операций на задержку, однако это еще пункт в пользу поэтапного перехода к микросервисам. Внесите небольшое изменение, а затем оцените его воздействие. Это предполагает, что у вас есть какой-то способ измерения сквозного времени ожидания для интересующих операций (здесь могут помочь распределенные инструменты трассировки, такие как Jaeger). Но также необходимо иметь представление о том, какая задержка приемлема. Иногда замедление вполне допустимо, если операция все еще достаточно быстрая!

Согласованность данных

Переход от монолита, где данные хранятся и управляются в единой базе данных, к распределенной системе, в которой несколько процессов управляют состояни-

ем в разных БД, создает потенциальные проблемы в отношении согласованности данных. В прошлом вы, возможно, полагались на транзакции базы данных для управления изменениями состояния, но, работая с микросервисной архитектурой, необходимо понимать, что подобную безопасность нелегко обеспечить. Использование распределенных транзакций в большинстве случаев оказывается весьма проблематичным при координации изменений состояния.

Вместо этого, возможно, придется использовать такие концепции, как саги (о чем я подробно расскажу в главе 6) и согласованность в конечном счете, чтобы управлять состоянием вашей системы и анализировать его. Эти идеи могут потребовать основательных изменений вашего отношения к данным в своих системах, что может оказаться довольно сложной задачей при переносе существующих систем. Опять же это еще одна веская причина быть осторожными в скорости декомпозиции своего приложения. Принятие поэтапного подхода к декомпозиции, чтобы получить возможность оценить влияние изменений на архитектуру, действительно важно.

Стоит ли вам использовать микросервисы?

Несмотря на стремление определенных групп сделать микросервисные архитектуры подходом по умолчанию, я считаю, что их внедрение все еще требует тщательного обдумывания из-за многочисленных сложностей, описанных выше. Необходимо оценить предметную область, навыки разработчиков и технологический арсенал и понять, каких целей вы пытаетесь достичь, прежде чем принимать решение, подходят ли вам микросервисы. Они представляют собой общий архитектурный подход, а не какой-то конкретный метод. Контекст вашей организации должен сыграть решающую роль в принятии решения, идти ли по этому пути.

Тем не менее я хочу обрисовать несколько ситуаций, которые обычно склоняют меня к выбору микросервисов и наоборот.

Кому микросервисы не подойдут

Учитывая важность определения стабильных границ сервисов, я считаю, что микросервисные архитектуры часто не подходят для совершенно новых продуктов или стартапов. В любом случае предметная область, с которой вы работаете, обычно претерпевает значительные изменения по мере создания проекта. Этот сдвиг, в свою очередь, приведет к большему количеству преобразований, вносимых в границы сервисов, а их координация представляется дорогостоящим мероприятием. В целом я считаю, что более целесообразно подождать, пока модель предметной области не стабилизируется, прежде чем пытаться определить границы сервиса.

Действительно существует соблазн для стартапов начать работать на микросервисах, мотивируя это тем, что «если мы действительно добьемся успеха, нам нужно будет масштабироваться!». Проблема в том, что заранее не известно, захочет ли кто-нибудь вообще использовать ваш продукт. И даже если вы добьетесь такого успеха, что потребуется масштабируемая архитектура, то финальный вариант может сильно отличаться от того, что вы начали создавать в принципе. Изначально компания Uber ориентировалась на лимузины, а хостинг Flickr сформировался из попыток создать многопользовательскую онлайн-игру. Процесс поиска соответствия продукта рынку означает, что в конечном счете вы рискуете получить продукт, совершенно отличный от того, что вы задумывали.

Стартапы, как правило, располагают меньшим количеством людей, что создает больше проблем в отношении микросервисов. Микросервисы приносят с собой источники новых задач и усложнение системы, а это может ограничить ценную пропускную способность. Чем меньше команда, тем более заметными будут эти затраты. Поэтому при работе с небольшими коллективами, состоящими всего из нескольких разработчиков, я настоятельно не рекомендую микросервисы.

Камнем преткновения в вопросе микросервисов для стартапов является весьма ограниченное количество человеческих ресурсов. Для небольшой команды может быть трудно обосновать использование рассматриваемой архитектуры из-за проблем с развертыванием и управлением самими микросервисами. Некоторые называют это «налогом на микросервисы». Когда эти инвестиции приносят пользу большему количеству людей, их легче оправдать. Но если в команде из пяти человек один тратит свое время на решение этих проблем, это означает, что создание вашего продукта замедляется. Гораздо проще перейти к микросервисам позже, когда вы поймете, где находятся ограничения в архитектуре и каковы слабые места системы, — тогда вы сможете сосредоточить свою энергию на внедрении микросервисов.

Наконец, испытывать трудности с микросервисами могут и организации, создающие ПО, которое будет развертываться и управляться их клиентами. Как я уже говорил, архитектуры микросервисов могут значительно усложнить процесс развертывания и эксплуатации. Если вы используете программное обеспечение самостоятельно, можете компенсировать это, внедрив новые технологии, развив определенные навыки и изменив методы работы. Но не стоит ожидать подобного от своих клиентов, которые привыкли получать ваше ПО в качестве установочного пакета для Windows. Для них будет шоком, если вы отправите следующую версию своего приложения и скажете: «Просто поместите эти 20 подов в свой кластер Kubernetes!» Скорее всего, они даже не представляют, что такое под, Kubernetes или кластер.

Где микросервисы хорошо работают

По моему опыту, главной причиной, по которой организации внедряют микросервисы, является возможность большему количеству программистов работать над одной и той же системой, при этом не мешая друг другу. Правильно определив свою архитектуру и организационные границы, вы позволите многим людям работать независимо друг от друга, снижая вероятность возникновения разногласий. Если пять человек организовали стартап, они, скорее всего, сочтут микросервисную архитектуру обузой. В то время как быстро растущая компания, состоящая из сотен сотрудников, скорее всего, придет к выводу, что ее рост гораздо легче приспособить к рассматриваемой нами архитектуре.

Приложения типа «программное обеспечение как услуга» (software as a service, SaaS) также хорошо подходят для архитектуры микросервисов. Обычно такие продукты работают 24/7, что создает проблемы, когда дело доходит до внедрения изменений. Возможность независимого выпуска микросервисных архитектур предоставляет огромное преимущество. Кроме того, микросервисы по мере необходимости можно увеличить или уменьшить. Это означает, что по мере того, как вы устанавливаете базовый уровень нагрузки вашей системы, вы наиболее экономичным способом получаете больше контроля над обеспечением возможности масштабирования.

Благодаря технологически независимой природе микросервисов вы сможете получить максимальную отдачу от облачных платформ. Провайдеры публичных облачных сервисов предоставляют широкий спектр услуг и механизмов развертывания для вашего кода. Вам гораздо проще сопоставить требования конкретных служб с облачными сервисами, которые наилучшим образом помогут вам реализовать ваши задачи. Например, вы можете решить развернуть один сервис как набор функций, другой — как управляемую виртуальную машину (VM), а третий — на управляемой платформе «платформа как услуга» (platform as a service, PaaS).

Хотя стоит отметить, что внедрение различных технологий развертывания часто может представлять собой проблему. Возможность легко опробовать какую-либо технологию — хороший способ быстро определить новые подходы. Одним из таких примеров является набирающая популярность платформа FaaS. Для соответствующих рабочих нагрузок она может значительно сократить объем операционных накладных расходов, но в настоящее время это не универсальный механизм развертывания.

Микросервисы также дают очевидные преимущества для организаций, стремящихся предоставлять услуги своим клиентам по множеству различных каналов. Многие компании ведут работы по цифровому преобразованию, по-видимому, связанные с попытками раскрыть функциональность, скрытую

в существующих системах. Они хотят дать пользователям новые впечатления и уникальный опыт взаимодействия с приложением, чтобы удовлетворить их потребности.

Прежде всего, микросервисная архитектура может предоставить вам большую гибкость по мере дальнейшего развития системы. Конечно, такая гибкость имеет свою цену. Но если вы не хотите отказываться от возможностей для будущих изменений — эту цену стоит заплатить.

Резюме

Микросервисные архитектуры могут предоставить огромную степень гибкости в выборе технологий, обеспечении надежности и масштабировании, организации команд и многом другом. Эта гибкость отчасти объясняет, почему многие люди используют микросервисные архитектуры. Но микросервисы влекут за собой значительные издержки, и вам необходимо убедиться, что они оправданны. Для многих микросервисы стали системной архитектурой по умолчанию, которую можно использовать практически во всех ситуациях. Тем не менее я по-прежнему считаю, что этот архитектурный выбор должен быть оправдан поставленными целями. Часто наименее сложные подходы могут обеспечить более легкое достижение результата.

Вместе с тем многие организации, особенно крупные, показали, насколько эффективными могут быть микросервисы. Когда основные концепции правильно поняты и реализованы, микросервисы могут стать чем-то большим, чем просто суммой частей системы.

Я надеюсь, что эта глава послужила хорошим введением. Далее мы рассмотрим, как лучше определять границы микросервиса, попутно исследуя темы структурированного программирования и предметно-ориентированного проектирования.

ГЛАВА 2

Как моделировать микросервисы

> Мой оппонент в своих рассуждениях напоминает мне язычника, который, когда его спросили, на чем стоит мир, ответил: «На черепахе». — «Но на чем стоит черепаха?» — «На другой черепахе».
>
> *Преподобный Джозеф Фредерик Берг (1854)*

Итак, вы знаете, что такое микросервисы, и, я надеюсь, имеете представление об их ключевых преимуществах. Вам, наверное, не терпится прямо сейчас пойти и создать их, правда? Но с чего начать? В этой главе мы поговорим о некоторых основополагающих концепциях, таких как скрытие информации, связанность (coupling) и связность (cohesion), и поймем, как они могут изменить представление о проведении границ микросервисов. Затем рассмотрим различные формы декомпозиции, а также более подробно обсудим методы предметно-ориентированного проектирования. Узнаем, как продумать границы ваших микросервисов, чтобы максимально использовать преимущества этого подхода и избежать возможного появления некоторых потенциальных проблем. Но сначала нам нужно определиться, с чем мы будем работать.

Представляем MusicCorp

Любые книги лучше усваиваются, если в них есть примеры. Там, где это возможно, я буду делиться реальными историями из жизни, а иногда описывать вымышленный сценарий, чтобы вы понимали, как концепция микросервисов работает в реальном мире.

Итак, представьте себе современный интернет-магазин MusicCorp. Компания MusicCorp до недавнего времени была традиционным розничным магазином, но после выхода из бизнеса по продаже виниловых пластинок она все больше и больше сосредотачивала свои усилия в Интернете. У фирмы есть веб-сайт, но ее руководство считает, что сейчас самое время удвоить усилия в онлайн-мире.

В конце концов, эти смартфоны для прослушивания музыки — всего лишь мимолетное увлечение (очевидно, что портативный плеер Zune намного лучше), и меломаны с нетерпением ждут, когда им доставят их любимые пластинки. Качество важнее удобства, верно? И хотя, возможно, только недавно стало известно, что Spotify на самом деле является цифровым музыкальным сервисом, а не каким-то средством для ухода за кожей для подростков, компания MusicCorp вполне довольна своей собственной направленностью и уверена, что весь этот стриминговый бизнес скоро прогорит.

Несмотря на то что MusicCorp немного отстает от жизни, у нее большие планы. К счастью, в компании решили, что отличный шанс захватить рынок — убедиться, что есть возможность вносить изменения самым простым способом. Микросервисы для победы!

Что делает границу микросервиса качественной

Прежде чем команда MusicCorp встанет на путь создания сервиса за сервисом в попытке доставить восьмидорожечные кассеты всем и каждому, давайте немного поговорим о самой важной основополагающей идее, которую не стоит забывать. Мы хотим, чтобы микросервисы можно было изменять и развертывать, а их функциональность предоставлялась нашим пользователям независимым образом. Возможность изменять один микросервис изолированно от другого имеет жизненно важное значение. Итак, о чем нужно помнить, когда мы хотим провести границы вокруг сервисов?

По сути, микросервисы — это просто еще одна форма модульной декомпозиции, хотя и с сетевым взаимодействием между моделями и всеми вытекающими проблемами. К счастью, это означает, что мы можем полагаться на высокий уровень развития техники в области модульного ПО и использовать структурированное программирование для определения границ. Имея это в виду, давайте более подробно рассмотрим три ключевые концепции, которые определяют качество границ микросервиса и которые мы кратко затронули в главе 1, — скрытие информации, связанность и связность.

Скрытие информации

Скрытие информации — это концепция, разработанная Дэвидом Парнасом для поиска наиболее эффективного способа определения границ модулей[1]. Скрытие информации подразумевает скрытие как можно большего количества деталей за границей модуля (или в нашем случае микросервиса). Парнас рассмотрел преимущества, которые теоретически должны дать нам модули.

---

[1] *Parnas D.* On the Criteria to Be Used in Decomposing Systems into Modules (статья в журнале, Университет Карнеги-Меллона, 1971). https://oreil.ly/BnVVg.

Ускоренное время разработки

Разрабатывая модули независимо, мы можем выполнять больше параллельной работы и уменьшить влияние от добавления большего количества разработчиков в проект.

Понятность

Каждый модуль можно рассматривать и понимать изолированно. Это, в свою очередь, дает представление о том, что делает система в целом.

Гибкость

Модули можно изменять независимо друг от друга, что позволяет вносить изменения в функциональность системы, не требуя преобразований других модулей. Кроме того, их можно комбинировать различными способами для обеспечения новых возможностей.

Этот список желаемых характеристик прекрасно дополняет то, чего мы пытаемся достичь с помощью микросервисных архитектур — и действительно, теперь я рассматриваю микросервисы просто как еще одну форму модульной архитектуры. Адриан Колер просмотрел ряд работ Дэвида Парнаса и, изучив их с точки зрения микросервисов, составил свое резюме[1].

Реальность такова, что наличие модулей не приводит к фактическому достижению необходимых результатов. Многое зависит от того, как формируются границы модуля. Согласно исследованиям Парнаса, скрытие информации стало ключевым методом, помогающим получить максимальную отдачу от модульных архитектур, и с современной точки зрения то же самое относится к микросервисам.

В другой статье Парнаса[2] мы нашли такую цитату: «Связи между модулями — это предположения, которые модули делают друг о друге».

Уменьшая количество предположений, которые один модуль (или микросервис) делает относительно другого, мы напрямую влияем на связи между ними. Сохраняя число предположений небольшим, легче гарантировать, что мы сможем изменить одну часть, не затрагивая другие. Если разработчик, модифицирующий блок, имеет четкое представление о том, как модуль используется другими элементами системы, ему будет проще безопасно вносить преобразования, чтобы вышестоящим абонентам не приходилось делать то же самое.

Это относится и к микросервисам, за исключением уже имеющейся у нас возможности развернуть измененный сервис без необходимости развертывания чего-либо еще, тем самым усиливая описанные Парнасом преимущества: ускорение времени разработки, понятность и гибкость.

Последствия скрытия информации проявляются во многих отношениях, и я буду поднимать эту тему на протяжении всей книги.

[1] *Colyer A.* On the Criteria... https://oreil.ly/cCtSV.

[2] *Parnas D.* Information Distribution Aspects.

Связность

Самое краткое объяснение, которое я слышал для описания связности[1], звучит так: «Код, в который вносят изменения как в единое целое, остается единым целым». Для наших целей это довольно хорошее определение. Как обсуждалось ранее, мы оптимизируем нашу микросервисную архитектуру с учетом простоты преобразований бизнес-функциональности. Поэтому нам требуется функциональность, сгруппированная таким образом, чтобы иметь возможность вносить изменения в как можно меньшем количестве мест.

Хотелось бы, чтобы связанное поведение находилось в одном месте, а несвязанное — где-то в другом. Почему? Да потому, что, если требуется пересмотреть поведение, хотелось бы иметь возможность изменить его в одном месте и опубликовать как можно скорее. Если же придется изменять его во многих разных местах, то понадобится выпустить множество различных сервисов (возможно, одновременно). Такой процесс происходит медленнее, а одновременное развертывание большого количества сервисов сопряжено с риском, поэтому хотелось бы избежать и того и другого. Следовательно, нужно найти в нашей предметной области границы, которые помогут обеспечить нахождение связанного поведения в одном месте; требуется также, чтобы эти границы имели как можно более слабую связь с другими границами. Если соответствующая функциональность распределена по всей системе, мы говорим, что связность слабая, тогда как для наших микросервисных архитектур мы стремимся к сильной связности.

Связанность

Когда между сервисами наблюдается слабая связанность[2], изменения, вносимые в один сервис, не требуют изменений в другом. Для микросервиса самое главное — возможность внесения изменений в один сервис и его развертывания без необходимости вносить изменения в любую другую часть системы. И это действительно очень важно.

Что вызывает необходимость сильной связанности? Классической ошибкой является выбор такого стиля интеграции, который тесно привязывает один сервис к другому, что при изменении внутри сервиса требует изменений в его потребителях.

Слабо связанный сервис имеет необходимый минимум сведений о сервисах, с которыми ему приходится сотрудничать. Это также означает, что нам следует ограничить количество различных типов вызовов от одного сервиса к другому, потому что помимо потенциальной проблемы с производительностью интенсивное общение может привести к сильной связанности.

---

[1] Связность (cohesion) — мера силы взаимосвязанности элементов сервиса; способ и степень, в которой задачи, выполняемые им, связаны друг с другом. — *Примеч. пер.*

[2] Связанность (coupling) представляет собой степень взаимосвязи между сервисами. При создании систем необходимо стремиться к максимальной независимости сервисов, то есть их связанность должна быть минимальной. — *Примеч. пер.*

Взаимодействие связанности и связности

Как упоминалось ранее, понятия связанности и связности, очевидно, смежные. Логично предположить, что, если связанная функциональность распределена по всей системе, изменения в этой функциональности будут распространяться через проложенные границы, что подразумевает более сильную связанность. Закон Константина, названный в честь пионера структурированного проектирования Ларри Константина, четко резюмирует это.

> Структура стабильна, если связность сильная, а связанность слабая[1].

Здесь для нас важна концепция стабильности. Чтобы границы обеспечивали возможность независимого развертывания и позволяли работать над микросервисами параллельно, снижая уровень координации между командами, работающими над этими сервисами, необходима определенная степень стабильности самих границ. Если контракт, предоставляемый микросервисом, постоянно меняется обратно несовместимым образом, то это приведет к постоянному изменению вышестоящих потребителей.

Связанность и связность тесно переплетены, то есть совпадают в том смысле, что оба понятия описывают взаимосвязь между вещами. Связность применима к отношениям между вещами *внутри* границы (микросервис в нашем контексте), а связанность представляет отношения между объектами *через* границу. Нет *наилучшего* способа организовать код. Связанность и связность — всего лишь характеристики, позволяющие сформулировать различные компромиссы, на которые мы идем в отношении кода. Все, к чему мы можем стремиться, — найти правильный баланс между этими двумя идеями, наиболее подходящий для вашего конкретного контекста и проблем.

Помните, что мир не статичен — вполне возможно, что по мере изменения системных требований найдутся причины пересмотреть принятые решения. Иногда части системы могут претерпевать столь значительные преобразования, что стабильность может стать невозможной. Мы рассмотрим такой пример в главе 3, когда я поделюсь опытом разработчиков, стоящих за системой Snap CI.

Типы связанности

Из вышеприведенной информации можно сделать вывод, что любая связанность — плохо. Однако это не совсем так. В конечном счете определенная связаность в нашей системе будет неизбежно. Все, что мы хотим сделать, — уменьшить ее.

---

[1] В моей книге «От монолита к микросервисам. Эволюционные шаблоны для трансформации монолитной системы» я приписал это самому Ларри Константину. Однако цитату действительно следует отнести к книге 2003 года Handbook of Software and Systems Engineering (Addison-Wesley) за авторством Albert Endres и Dieter Rombach.

Была проделана огромная работа по изучению различных форм связанности в контексте парадигмы структурированного программирования, позволявшей рассматривать по большей части модульное (нераспределенное, монолитное) ПО. Многие из моделей оценки связанности пересекаются или противоречат друг другу. Но в первую очередь они говорят о процессах на уровне кода, а не о рассмотрении сервис-ориентированных взаимодействий. Поскольку микросервисы представляют собой стиль модульной архитектуры (хотя и с дополнительной сложностью распределенных систем), можно использовать многие из этих оригинальных концепций и применять их в отношении наших систем на основе микросервисов.

УРОВЕНЬ ТЕХНИКИ В ОБЛАСТИ СТРУКТУРИРОВАННОГО ПРОГРАММИРОВАНИЯ

Большая часть нашей текущей работы в области вычислительной техники опирается на предыдущую. Невозможно учесть все, что было ранее, но в этом издании я стремился охватить как можно больше вопросов. Отчасти для того, чтобы отдать должное там, где это необходимо, отчасти, чтобы оставить несколько хлебных крошек для читателей, желающих изучить определенные темы более подробно. А также чтобы показать, что многие из этих идей опробованы и проверены.

Если говорить о создании новых фрагментов систем на основе предыдущих работ, в этой книге есть несколько предметных областей, у которых такой же предшествующий уровень техники, как у структурированного программирования. Книга Ларри Константина, написанная в соавторстве с Эдвардом Йордоном, считается одной из самых важных в этой области. Также советую книгу Мейлира Пейджа-Джонса «Практическое руководство по проектированию структурированных систем». К сожалению, обе книги сложно достать, поскольку тиражи уже давно распроданы, а электронные варианты не выпускались.

Не все идеи четко соответствуют друг другу, поэтому я сделал все возможное, чтобы синтезировать рабочую модель для различных типов связанности микросервисов. Там, где эти идеи сопоставляются с предыдущими определениями, я придерживаюсь уже описанных терминов. В других местах мне приходилось придумывать новые понятия или смешивать идеи из иных источников. Поэтому, пожалуйста, считайте весь материал построенным на основе множества предыдущих работ в этой области. Я просто пытаюсь придать этому больше смысла в контексте микросервисов.

На рис. 2.1 представлен краткий обзор различных типов связанности, организованных от слабых (желательно) до сильных (нежелательно).

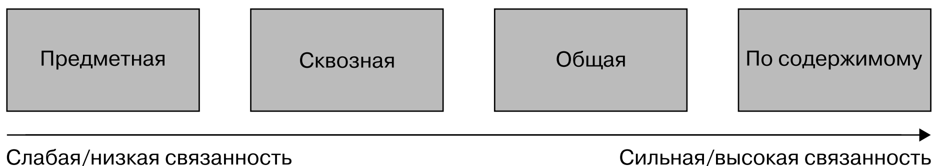


Рис. 2.1. Различные типы связанностей, от слабых (низких) до сильных (высоких)

Далее мы изучим каждую форму по очереди и рассмотрим примеры, показывающие, как эти формы могут проявляться в нашей микросервисной архитектуре.

Предметная связанность

Предметная (доменная) связь описывает ситуацию, в которой одному микросервису необходимо взаимодействовать с другим, поскольку первому требуется использовать функциональность, предоставляемую вторым микросервисом[1].

На рис. 2.2 показана часть того, как заказы на компакт-диски управляются внутри MusicCorp. В этом примере `Обработчик заказа` вызывает микросервис `Склад` для резервирования на складе, а микросервис `Оплата` принимает оплату. Следовательно, `Обработчик заказа` в этой операции зависит от микросервисов `Склад` и `Оплата` и связан с ними. Между сервисами `Склад` и `Оплата` нет такой связи, поскольку они не взаимодействуют.

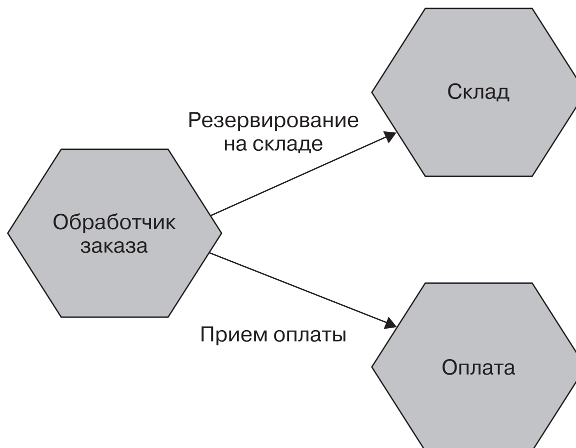


Рис. 2.2. Пример доменной связи, где сервису Обработчик заказов необходимо использовать функциональность, предоставляемую другими микросервисами

В микросервисной архитектуре данный тип взаимодействия практически неизбежен. Система, основанная на микросервисах, для выполнения своей работы полагается на взаимодействие нескольких микросервисов. Однако нам по-прежнему требуется свести это к минимуму. Ситуация, когда один микросервис зависит от нескольких нижестоящих микросервисов, означает, что он выполняет слишком много задач.

[1] Эта концепция аналогична протоколу предметных приложений, который определяет правила взаимодействий компонентов в системе, основанной на REST.

Как правило, доменная связанность считается слабой формой связи, хотя даже здесь вполне реально столкнуться с проблемами. Микросервис, которому необходимо взаимодействовать с большим количеством нижестоящих микросервисов, может указывать на то, что слишком много логики было централизовано. Предметная связанность также может стать проблематичной, поскольку между сервисами передаются все более усложняющиеся наборы данных. Обычно это говорит о менее благоприятных формах связи, которые мы рассмотрим в дальнейшем.

Помните о важности скрытия информации. Делитесь только тем, что необходимо, и отправляйте только минимальный требуемый объем данных.

ПРИМЕЧАНИЕ О ВРЕМЕННОЙ СВЯЗАННОСТИ

Другая достаточно известная форма связанности — *временная*. С точки зрения связи, ориентированной на код, временная связанность относится к ситуации, в которой понятия объединяются вместе только потому, что они происходят одновременно. Временная связь в контексте распределенной системы имеет немного другое значение: одному микросервису требуется, чтобы другой микросервис выполнял что-то в то же время.

Для завершения операции оба микросервиса должны быть запущены и доступны для одновременной связи друг с другом. Итак, на рис. 2.3, где `Обработчик заказов` MusicCorp выполняет синхронный HTTP-вызов сервиса `Склад`, сервис `Склад` должен быть запущен и доступен одновременно с вызовом.

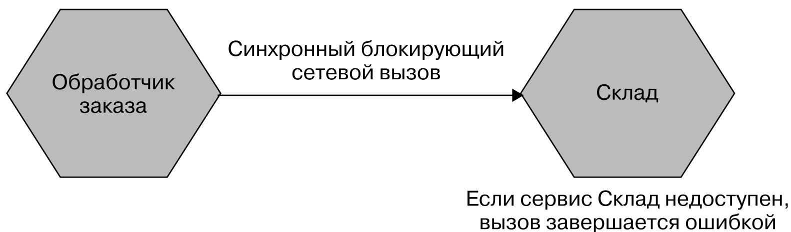


Рис. 2.3. Пример временной связи, при которой Обработчик заказов выполняет синхронный HTTP-вызов микросервиса Склад

Если по какой-либо причине `Обработчик заказов` не может связаться со складом, операция завершается ошибкой, так как не получается зарезервировать компакт-диски для отправки. Сервису `Обработчик заказов` также придется выполнять блокировку и ждать ответа от сервиса `Склад`, что может вызвать проблемы с точки зрения конкуренции за ресурсы.

Это не значит, что временная связь плоха. Просто об этом следует знать. По мере роста количества микросервисов с более сложными взаимодействиями проблемы временной связанности могут возрасти до такой степени, что становится все труднее масштабировать систему и поддерживать ее работоспособность. Один из способов избежать временной связанности — использовать некоторую форму асинхронной связи, такую как брокер сообщений.

Сквозная связанность

Сквозная связанность[1] описывает ситуацию, в которой один микросервис передает данные другому микросервису исключительно потому, что данные нужны какому-то третьему микросервису, находящемуся дальше по потоку. Во многих отношениях это одна из самых проблемных форм связи, поскольку она подразумевает, что вызывающий сервис должен знать, что вызываемый им сервис вызывает еще один. А также вызывающему микросервису необходимо знать, как работает удаленный от него микросервис.

В качестве примера сквозной связи более внимательно рассмотрим фрагмент системы обработки заказов MusicCorp. На рис. 2.4 показан `Обработчик заказов`, посылающий запрос к сервису `Склад` для подготовки заказа к отправке. В качестве полезной нагрузки запроса мы отправляем `Путевой лист`. Этот `Путевой лист` содержит не только адрес клиента, но и тип доставки. Сервис `Склад` просто передает эту информацию нижестоящему микросервису `Доставка`.

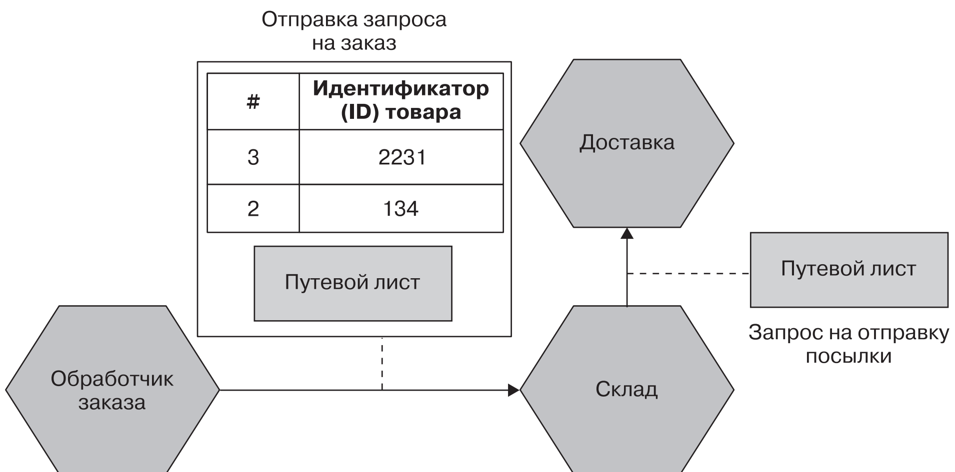


Рис. 2.4. Сквозная связанность, при которой данные передаются микросервису исключительно потому, что они нужны другому нижестоящему сервису

Основная проблема сквозной связи заключается в том, что изменение требуемых данных ниже по потоку может привести к более значительному изменению выше по потоку. В нашем примере если сервису `Доставка` в настоящее время

[1] Сквозная связанность — это мое название того, что первоначально было описано Мейлиром Пейдж-Джонсом как «блуждающая связь». Я решил использовать другой термин, так как первоначальный показался мне несколько неудобным для более широкой аудитории.

требуется изменение формата или содержания данных, то сервисам `Склад` и `Обработчик заказов`, скорее всего, тоже потребуется что-то изменить.

Есть несколько способов, с помощью которых это можно исправить. Первый — рассмотреть, имеет ли смысл для вызывающего микросервиса просто обходить посредника. В нашем примере это означает, что `Обработчик заказов` напрямую обращается к сервису `Доставка`, как показано на рис. 2.5. Однако это влечет за собой иные проблемы. Наш `Обработчик заказов` усиливает свою предметную связанность, поскольку `Доставка` станет еще одним микросервисом, с которым придется взаимодействовать. Если бы это была единственная проблема, такое решение все еще было бы допустимым, поскольку предметная связь представляет собой более слабую форму связи. Однако здесь это решение только усугубляет ситуацию, поскольку резервирование на складе должно происходить при помощи сервиса `Склад`, прежде чем посылка будет отправлена с помощью сервиса `Доставка`, и после завершения доставки необходимо соответствующим образом обновить перечень товаров на складе. Это добавляет больше сложности и логики в `Обработчик заказов`, которая ранее была скрыта внутри `Склада`.

Рис. 2.5. Один из способов обойти сквозную связь заключается в непосредственном взаимодействии с нижестоящим сервисом

Для этого конкретного примера я мог бы рассмотреть более простое (хотя и более тонкое) изменение, а именно — полностью скрыть запрос документа `Путевой лист` от сервиса `Обработчик заказов`. Идея делегировать работу по управлению запасами и по организации отправки посылки сервису `Склад` имеет

смысл, но не очень хорошо, что мы допустили утечку некоторых реализаций более низкого уровня, а именно тот факт, что микросервису `Доставка` требуется `Путевой лист`. Один из способов скрыть эту деталь — сделать так, чтобы `Склад` получал необходимую информацию в рамках своего контракта, и тогда он создаст `Путевой лист` локально, как показано на рис. 2.6. Теперь, если сервис `Доставка` меняет свой сервисный контракт, такое изменение будет невидимым для сервиса `Обработчик заказов`, пока `Склад` собирает необходимые данные.

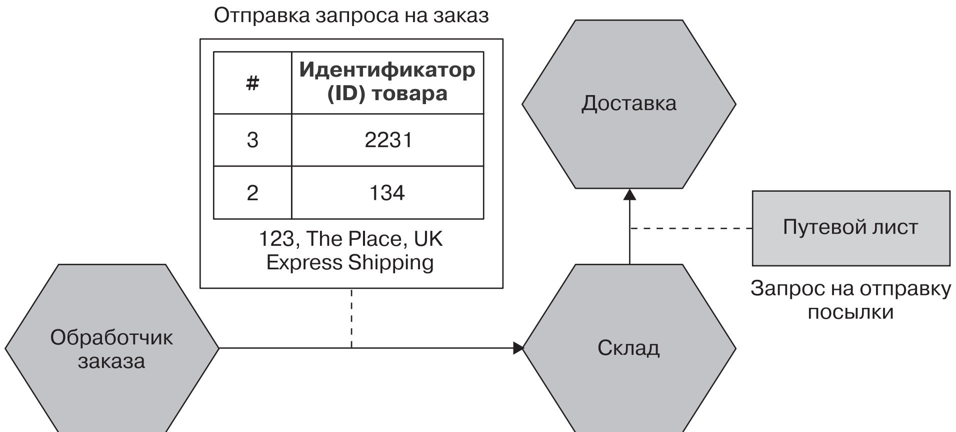


Рис. 2.6. Скрытие необходимости в Путевом листе от Обработчика заказов

Пример на рис. 2.7, условно говоря, довольно безобидный. Это связано с тем, что по самой своей природе справочные данные нечасто меняются, а также потому, что они доступны только для чтения. Поэтому я спокойно отношусь к совместному использованию справочных данных таким образом. Однако общая связанность становится более проблематичной, если структура данных меняется чаще или если несколько микросервисов читают и записывают одни и те же данные.

Хотя описанные действия помогут защитить микросервис `Склад` от некоторых изменений в сервисе `Доставка`, есть вещи, которые все равно потребуют преобразований от всех участников. Допустим, мы хотим осуществлять международные поставки. Сервису `Доставка` потребуется включить документ `Таможенная декларация` в перечень документов запроса `Путевой лист`. Если это необязательный параметр, то мы могли бы без проблем развернуть новую версию микросервиса `Доставка`. Однако если он обязательный, то сервису `Склад` необходимо будет создать его. Данный сервис может сделать это с помощью имеющейся у него информации или потребовать ее от `Обработчика заказов`.

В этом случае необходимость модификации всех трех микросервисов не исчезла, но нам было предоставлено гораздо больше возможностей в отношении того, когда и как эти изменения можно вносить. Если бы у нас была сильная (сквозная) связь из начального примера, добавление нового документа

Таможенная декларация могло бы потребовать поэтапного развертывания всех трех микросервисов. По крайней мере, скрыв эту деталь, мы могли бы гораздо легче пошагово осуществить развертывание.

Рис. 2.7. Несколько сервисов, получающих доступ к общим статическим справочным данным

Один из последних подходов, который мог бы помочь уменьшить сквозную связь, состоит в том, чтобы указать сервису Обработчик заказов продолжать отправлять Путевой лист в микросервис Доставка через Склад. Но Складу не нужно знать структуру самого Путевого листа. Обработчик заказов отправляет лист как часть запроса на заказ, но Склад не пытается просмотреть или обработать поле — он просто относится к нему как к неструктурированному объекту данных и не заботится о содержимом. Вместо этого он просто отправляет документ дальше. При изменении формата в Путевом листе все же придется внести изменения в микросервисы Обработчик заказов и Доставка, но, так как Склад не интересуется фактическим содержимым листа, его менять не надо.

Общая связанность

Общая связанность возникает, когда два или более микросервиса используют общий набор данных. Простым и распространенным примером такой формы связанности могут служить множественные микросервисы, использующие одну и ту же БД, но это также может проявляться в использовании общей памяти или файловой системы.

Основная проблема систем с общей связанностью заключается в том, что изменения в структуре данных могут повлиять на несколько микросервисов одновременно. Рассмотрим пример некоторых сервисов MusicCorp на рис. 2.7.

Как мы условились ранее, MusicCorp работает по всему миру, поэтому компании требуется различная информация о странах, в которых она зарегистрирована. Здесь все множество сервисов считывает статические справочные данные из общей БД. Если схема этой базы изменится обратно несовместимым образом, это потребует преобразований для каждого потребителя БД. На практике подобные общие данные, как правило, очень трудно изменить.

На рис. 2.8 показана ситуация, в которой сервисы `Обработчик заказов` и `Склад` считывают и записывают данные из общей таблицы `Заказ`, чтобы помочь управлять процессом отправки компакт-дисков клиентам MusicCorp. Оба микросервиса обновляют столбец `Статус`. `Обработчик заказов` может задать статусы `РАЗМЕЩЕН`, `ОПЛАЧЕН` и `ЗАВЕРШЕН`, в то время как `Склад` задает статусы `СОБИРАЕТСЯ` или `ОТПРАВЛЕН`.

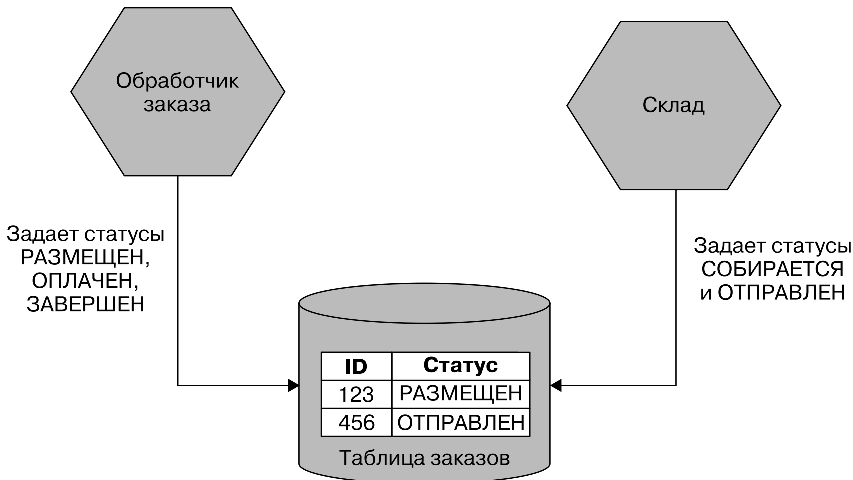


Рис. 2.8. Пример общей связанности, при которой сервисы Обработчик заказов и Склад обновляют одну и ту же запись заказа

Хотя вы можете счесть рис. 2.8 несколько надуманным, этот простой пример помогает проиллюстрировать основную проблему общей связанности. Концептуально у нас есть микросервисы `Обработчик заказов` и `Склад`, управляющие различными аспектами жизненного цикла заказа. Можно ли быть уверенными, что при внесении изменений в `Обработчик заказов` не возникнет конфликта интересов с сервисом `Склад`?

Гарантией того, что состояние чего-либо изменяется правильным образом, было бы создание конечного автомата. Конечный автомат может использоваться для управления переходом некоторого объекта из одного состояния в другое, запрещая недопустимые переходы состояний. На рис. 2.9 можно увидеть разрешенные переходы состояний для заказа в MusicCorp. Заказ может перейти непосредственно из статуса `РАЗМЕЩЕН` в статус `ОПЛАЧЕН`, но

не напрямую из статуса РАЗМЕЩЕН к СОБИРАЕТСЯ (этого конечного автомата, скорее всего, будет недостаточно для реальных бизнес-процессов, выполняемых при покупке и доставке товаров, но смысл этого простого примера — проиллюстрировать идею).

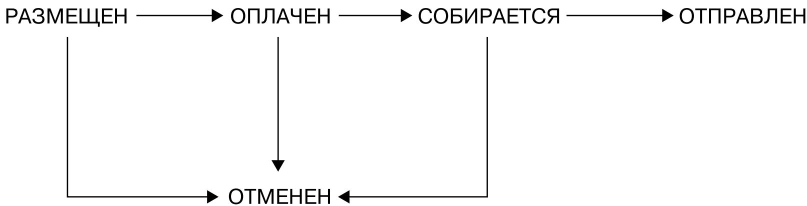


Рис. 2.9. Обзор допустимых переходов статусов для заказа в MusicCorp

Проблема этого конкретного примера в том, что и `Склад`, и `Обработчик заказов` разделяют ответственность за управление этим конечным автоматом. Как мы можем быть уверены, что они согласуют допустимые переходы? Существуют способы управления подобными процессами через границы микросервиса; мы вернемся к этой теме, когда будем обсуждать саги в главе 6.

Потенциальным решением здесь было бы обеспечить состояние, при котором один микросервис управлял бы заказом. На рис. 2.10 отправлять запросы на обновление статуса заказа в сервисе `Заказ` могут микросервисы `Склад` или `Обработчик заказов`. Здесь сервис `Заказ` будет источником истины для любого размещенного заказа. В этой ситуации действительно важно, чтобы мы рассматривали запросы от сервисов `Склад` и `Обработчик заказов` именно как *запросы*. Задачей сервиса `Заказ` станет управление допустимыми переходами статусов, целиком связанными с заказом. Таким образом, если сервис `Заказ` получит запрос от `Обработчика заказов` на изменение статуса с `РАЗМЕЩЕН` непосредственно на `ЗАВЕРШЕН`, сервис заказа вправе отклонить заявку, если такое изменение недопустимо.

Убедитесь, что у нижестоящего микросервиса есть возможность отклонить недействительный запрос, отправленный в микросервис.

Альтернативным подходом в данном случае будет реализация сервиса `Заказ` в виде чего-то большего, чем просто оболочка для операций CRUD с базой данных, где запросы сопоставляются непосредственно с обновлениями БД. Это похоже на объект, содержащий приватные поля, в котором публичные геттеры и сеттеры просочились из микросервиса к вышестоящим потребителям (снижение связности), и мы вернулись в мир управления приемлемыми переходами состояний между несколькими различными сервисами.

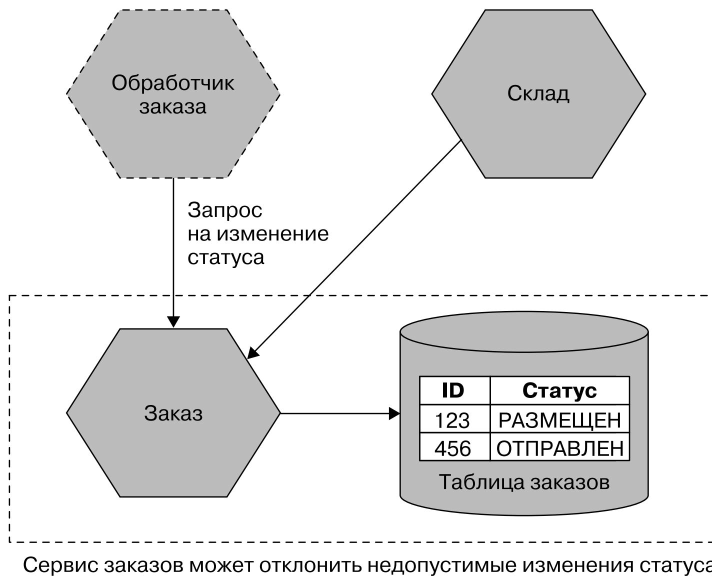
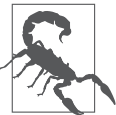


Рис. 2.10. Сервисы Обработчик заказов и Склад могут выполнить запрос на изменения в заказе, но решение о том, какие запросы допустимы, принимает микросервис Заказ



Если вы видите микросервис, который выглядит просто как тонкая оболочка для CRUD-операций с базой данных, — это признак слабой связности и более сильной связанности, поскольку логика, которая должна быть в этом сервисе для управления данными, распределена по другим местам вашей системы.

Источники общей связанности также являются потенциальными виновниками конкуренции за ресурсы. Множество микросервисов, использующих одну и ту же файловую систему или базу данных, могут перегружать этот ресурс, вызывая серьезные последствия при его замедлении или недоступности. Общая БД особенно подвержена такой проблеме из-за возможной подачи к ней произвольных запросов различной сложности. Я видел не одну базу данных, поставленную на колени ресурсоемким SQL-запросом, — возможно, я даже был виновником такой ситуации один или два раза[1].

Так что общая связанность *иногда* допустима, но чаще всего — нет. Даже если это неопасно, мы все равно ограничены в тех изменениях, которые можно внести в общие данные, что говорит об отсутствии связности в нашем коде. Это также может вызвать разногласия между сервисами. Именно по этим причинам мы считаем общую связанность одной из наименее желательных форм связи, но бывает и хуже.

[1] Хорошо, не один и не два раза. Гораздо чаще, чем раз или два...

Связанность по содержимому

Связанность по содержимому проявляется, когда вышестоящий сервис проникает во внутренние компоненты нижестоящего и изменяет его состояние. Наиболее распространенный вариант этого типа связанности представлен обращением внешнего сервиса к базе данных другого микросервиса и изменением ее напрямую. Различия между связанностью по содержимому и общей связанностью практически не заметны. В обоих случаях два или более микросервиса выполняют чтение и запись одного и того же набора данных. При общей связанности используется общая внешняя зависимость, которую вы не контролируете. При связанности по содержимому границы становятся менее четкими, а разработчикам все сложнее изменять систему.

Вернемся к нашему предыдущему примеру MusicCorp. На рис. 2.11 показан сервис `Заказ`, который управляет допустимыми изменениями статусов заказов в нашей системе. `Обработчик заказов` отправляет запросы сервису `Заказ`, делегируя не только право на изменение статуса, но и ответственность за принятие решения о том, какие переходы статусов допустимы. С другой стороны, `Склад` напрямую обновляет таблицу, в которой хранятся данные заказа, минуя любые функции сервиса `Заказ`, способные проверять допустимые преобразования. Мы должны надеяться, что сервис `Склад` содержит согласованный набор логики, гарантирующий внесение только разрешенных изменений. В лучшем случае логика продублируется. В худшем — мы можем получить заказы в очень необычных местах.

В этой ситуации у нас также возникает проблема, связанная с тем, что внутренняя структура данных нашей таблицы заказов общедоступна. Теперь при преобразовании сервиса `Заказ` стоит предельно осторожно вносить изменения в эту конкретную таблицу — даже когда для нас очевидно, что к ней напрямую обращаются сторонние пользователи. Исправить ситуацию с минимумом усилий поможет разрешение сервису `Склад` отправлять запросы в сервис `Заказ` напрямую. Здесь можно проверить запрос, а также скрыть внутренние детали, что сделает внесение последующих изменений в `Заказ` намного проще.

Если вы работаете над микросервисом, очень важно иметь четкое представление о том, что можно свободно изменять, а что нельзя. Для большей ясности скажем, что, как разработчик, вы должны знать, что сервис становится общедоступным при модификации функциональности, представляющей собой часть контракта. Необходимо убедиться, что при внесении изменений работа вышестоящих потребителей не будет нарушена. Функциональность, которая не влияет на контракт, предоставляемый вашим микросервисом, можно преобразовать без проблем.

Безусловно, сложности, возникающие при общей связанности, применимы и к связанности по содержимому, но во втором случае есть дополнительные источники головной боли, которые делают такой способ настолько проблематичным, что его называют в том числе *патологической связанностью*.

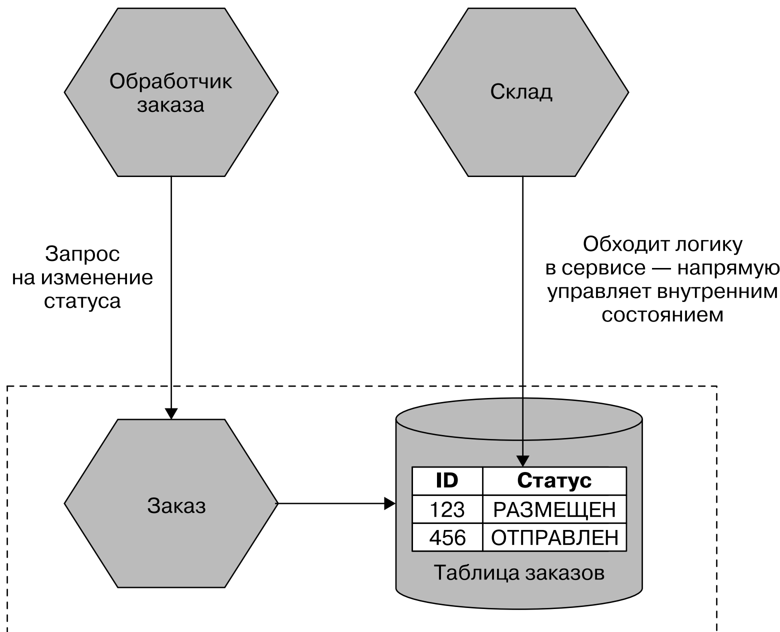


Рис. 2.11. Пример связанности по содержимому, при которой Склад получает прямой доступ к внутренним данным сервиса Заказ

Когда вы разрешаете внешней стороне прямой доступ к *своей* базе данных, она фактически становится частью внешнего контракта, и даже в таком случае вам будет сложно решить, что можно или нельзя изменить. Теряется способность определять, что относится к общим ресурсам (и, следовательно, не может быть без труда изменено), а что скрыто. Скрытие информации вышло из-под контроля. Короче говоря, избегайте связанности по содержимому.

Немного предметно-ориентированного проектирования

В главе 1 описан основной механизм, используемый для определения границ микросервисов. Процесс вращается вокруг самой предметной области, и предметно-ориентированное проектирование (DDD, domain-driven design) применяется, чтобы помочь создать ее модель. Давайте теперь расширим понимание того, как DDD работает в контексте микросервисов.

Разумеется, все разработчики стремятся к тому, чтобы их программы лучше отражали реальный мир, в котором они будут работать. Объектно-ориентированные языки программирования, такие как Simula, были разработаны именно для

моделирования реальных предметных областей. Но для того, чтобы идея отражения реального мира действительно обрела форму, требуется нечто большее, чем возможности языка программирования.

В книге Эрика Эванса[1] рассмотрен ряд важных идей, которые помогли нам лучше представить проблемную предметную область в своих программах. Полное изучение этих идей выходит за рамки данной книги, но есть несколько основных концепций DDD, которые стоит выделить.

Единый язык

Общий язык, определенный и принятый для использования в коде и при описании предметной области с целью облегчения коммуникации.

Агрегат

Набор объектов, управляемых как единое целое, обычно ссылающихся на концепции реального мира.

Ограниченный контекст

Четкая граница внутри предметной области бизнеса, которая обеспечивает функциональность более широкой системы, но также скрывает сложность. Часто соответствует границам организации.

Единый язык

Идея использования единого языка говорит, что мы должны стремиться употреблять в нашем коде те же термины, что и пользователи. Наличие общего языка между командой доставки и реальными людьми облегчит моделирование предметной области реального мира, а также улучшит коммуникацию.

В качестве контрпримера хочу поделиться опытом работы в крупном международном банке. Мы трудились в области корпоративной ликвидности — этот причудливый термин означает способность перемещать наличные деньги между различными счетами, принадлежащими одному и тому же юридическому лицу. С владелицей продукта оказалось приятно работать: она четко понимала, что хотела вывести на рынок. Работая с ней, мы обсуждали стрижки и уборку в конце дня — все, что имело большой смысл в ее мире и имело значение для ее клиентов.

С другой стороны, в коде не было ничего из этого. В какой-то момент было принято решение использовать стандартную модель данных для БД. В широком смысле эта модель характеризовалась термином «банковская модель IBM», но я не мог понять, был ли это стандартный продукт IBM или просто творение консультанта из IBM. Определяя расплывчатое понятие «соглашения»,

[1] *Эванс Э.* Предметно-ориентированное проектирование: Структуризация сложных программных систем.

теория утверждала, что можно смоделировать любую банковскую операцию. Взять кредит? Для этого было соглашение. Покупка акции? Для этого было соглашение! Подача заявки на получение кредитной карты? Вы, наверное, уже догадались!

Модель данных загрязнила код до такой степени, что кодовая база была лишена всякого реального понимания создаваемой системы. Мы не создавали универсальное банковское приложение. Мы создавали систему специально для управления корпоративной ликвидностью. Проблема заключалась в необходимости сопоставить богатый язык предметной области с общими концепциями кода, что означало большую работу по переводу. В результате наши бизнес-аналитики чаще всего тратили свое время на объяснение одних и тех же концепций снова и снова.

Благодаря внедрению в код языка реального мира все стало намного проще. Разработчик, взявший в руки историю, написанную с использованием терминов, с высокой вероятностью поймет их значение и приступит к работе.

Агрегат

В DDD *агрегат (aggregate)* — несколько запутанное понятие, имеющее множество различных определений. Что это? Просто произвольный набор объектов? Самая маленькая единица извлечения из базы данных? Модель, всегда работавшая в моем представлении, предполагает рассмотрение агрегата в качестве понятия из реальной предметной области — подумайте о чем-то вроде заказа, инвойса, единицы хранения и т. д. Жизненный цикл агрегатов обычно связан с этими понятиями, что позволяет реализовать агрегат в виде конечного автомата.

Например, в предметной области MusicCorp агрегат `Заказ` может содержать несколько позиций, представляющих товары в заказе. Эти позиции имеют значение только как часть общего агрегата заказов.

Стоит рассматривать агрегаты как автономные единицы — необходимо убедиться, что код, обрабатывающий переходы состояний агрегатов, сгруппирован вместе с самим состоянием. Таким образом, один агрегат должен управляться одним микросервисом, хотя один микросервис может управлять несколькими агрегатами.

В целом агрегат — нечто имеющее состояние, идентичность, жизненный цикл, которыми можно управлять как частью системы, — обычно относится к концепциям реального мира.

Один микросервис будет управлять жизненным циклом и хранением данных одного или нескольких различных типов агрегатов. Если функция в другом сервисе захочет изменить один из этих агрегатов, ей потребуется либо напрямую запросить изменение, либо заставить сам агрегат реагировать

на другие процессы в системе, чтобы инициировать свои собственные переходы состояний.

Здесь важно понимать, что, если внешняя сторона запрашивает переход состояния в агрегате, тот в свою очередь может сказать «нет». В идеале необходимо реализовать свои алгоритмы таким образом, чтобы недопустимые переходы состояний были невозможны.

Одни агрегаты могут взаимодействовать с другими. На рис. 2.12 агрегат `Покупатель` связан с одним или несколькими агрегатами `Заказ` и `Список избранного`. Эти агрегаты могут управляться одним и тем же микросервисом или разными.

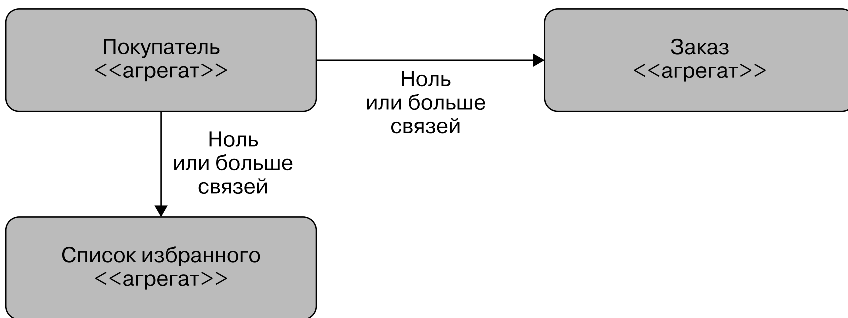


Рис. 2.12. Один агрегат Покупатель может быть связан с одним или несколькими агрегатами Заказ или Список избранного

Если эти отношения между агрегатами существуют внутри области одного микросервиса, их можно легко сохранить, используя что-то вроде внешнего ключа при работе с реляционной БД. Однако если отношения между этими агрегатами охватывают границы микросервиса, нам нужен какой-то способ их моделирования.

Теперь мы могли бы просто сохранить идентификатор агрегата непосредственно в локальной базе данных. Например, рассмотрим микросервис `Финансы`, управляющий гроссбухом, в котором содержатся транзакции покупателя. Локально, в БД микросервиса `Финансы`, может храниться столбец `CustID`, содержащий идентификатор этого покупателя. Если бы потребовалось получить больше информации об этом клиенте, нам пришлось бы выполнить поиск по микросервису `Покупатель`, используя данный идентификатор.

Проблема заключается в том, что концепция не очень ясна — на самом деле связь между столбцом `CustID` и покупателем, расположенным удаленно, полностью неявна. Чтобы узнать, как использовался идентификатор, нам пришлось бы взглянуть на код сервиса `Финансы`. Оптимальным решением является возможность хранить ссылку на внешний агрегат более очевидным способом.

На рис. 2.13 мы кое-что изменили, чтобы сделать взаимосвязь явной. Вместо обычного идентификатора для ссылки на клиента мы храним URI, который будет использоваться при создании системы на основе REST[1].

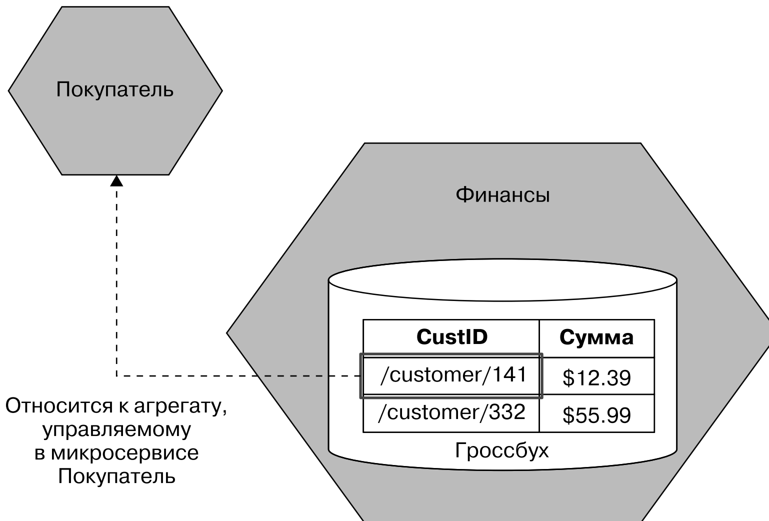


Рис. 2.13. Пример того, как может быть реализована взаимосвязь между двумя агрегатами в разных микросервисах

Преимущества такого подхода двояки. Связь явная, и в системе REST мы могли бы напрямую переназначить URI для поиска связанного ресурса. Но что, если вы не создаете систему REST? Фил Кальсадо описывает вариант этого подхода, используемый в SoundCloud[2], где они разработали схему псевдо-URI для межсервисных ссылок. Например, в `soundcloud:tracks:123` будет ссылка на композицию с идентификатором `123`. Эта запись более понятна для человека, смотрящего на идентификатор, и достаточно полезна для создания кода, который мог бы облегчить агрегированный поиск между микросервисами.

Существует множество способов разбить систему на агрегаты, причем некоторые варианты выбора весьма субъективны. Вы можете решить, по соображениям производительности или для простоты внедрения, изменять агрегаты с течением времени. Однако я считаю проблемы реализации второстепенными. Я начинаю с того, что при первоначальном проектировании отдаю все на откуп ментальной модели пользователей системы, пока в игру не вступят другие факторы.

---

[1] Я знаю, что некоторые люди возражают против использования шаблонных URI в системах REST, и понимаю почему — просто хочу упростить этот пример.

[2] *Calcado P.* Pattern: Using Pseudo-URIs with Microservices. https://oreil.ly/xOYMr.

Ограниченный контекст

Ограниченный контекст обычно представляет собой более широкую организационную границу. В рамках этой границы необходимо выполнять четкие обязанности. Все это немного запутанно, поэтому давайте рассмотрим на конкретном примере.

На складе MusicCorp происходит много всего: управление отгрузками (и непонятным возвратом), прием новых товаров, гонки на вилочных погрузчиках и т. д. Работа в финансовом отделе, возможно, менее увлекательна, но все же он выполняет важную функцию в нашей организации, занимаясь расчетом заработной платы, оплатой поставок и т. п.

Ограниченные контексты скрывают детали реализации. Существуют внутренние дилеммы, например, типы используемых вилочных погрузчиков мало интересуют кого-либо, кроме сотрудников склада. Эти проблемы должны быть скрыты от внешнего потребителя, которому не нужно о них знать, да и которого это не должно волновать в принципе.

С точки зрения реализации ограниченные контексты содержат один или несколько агрегатов. Некоторые из них могут быть доступны вне ограниченного контекста, другие — скрыты внутри. Ограниченные контексты могут иметь связи с другими ограниченными контекстами — при сопоставлении с сервисами эти зависимости становятся зависимостями между сервисами.

Вернемся к бизнесу MusicCorp. Наша предметная область — это весь бизнес, в котором мы работаем. Она охватывает все: от склада до приемной, от финансов до заказа. Мы можем как моделировать все это в нашем программном обеспечении, так и не моделировать, но тем не менее это предметная область, в которой мы работаем. Давайте подумаем об отдельных ее частях, выглядящих как ограниченные контексты, на которые ссылается Эрик Эванс.

Скрытые модели

Для MusicCorp можно рассматривать финансовый отдел и склад как два отдельных ограниченных контекста. Они оба поддерживают явный интерфейс связи с внешними потребителями (в виде отчетов об инвентаризации, платежных квитанций и т. д.), и у них есть детали, о которых знают только они (вилочные погрузчики, калькуляторы).

Финансовому отделу не нужно знать о подробностях внутренней работы склада. Однако ему все же необходима *кое-какая* информация, например об уровне запасов, чтобы поддерживать счета в актуальном состоянии. На рис. 2.14 показана примерная контекстная диаграмма. На ней видны внутренние складские концепции, такие как комплектовщик (тот, кто собирает заказы), стеллажи, тележки и т. д. Аналогичным образом записи в гроссбухе представляют собой неотъемлемую часть финансовых расчетов, но здесь они не сообщаются с внешним миром.

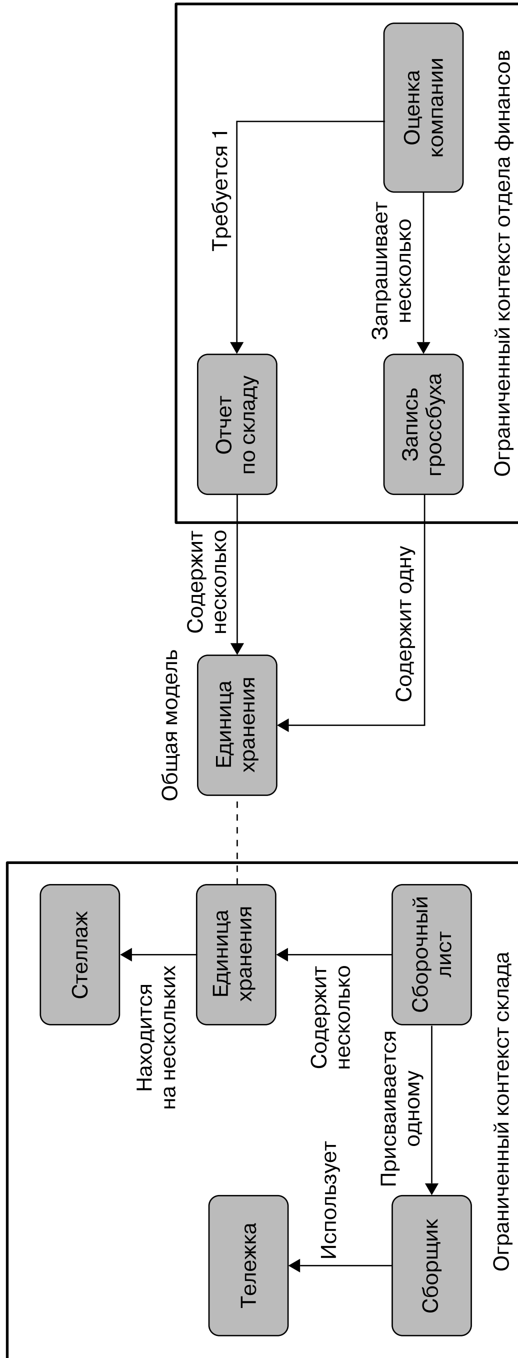


Рис. 2.14. Общая модель взаимодействия финансового отдела и склада

Однако, чтобы получить возможность оценить стоимость компании, сотрудникам финансового отдела нужна информация о складских запасах. Затем единица хранения становится общей моделью для двух контекстов. Обратите внимание, что нам не нужно слепо раскрывать все о единице хранения на складе из контекста склада. На рис. 2.15 показано, как `Единица хранения` внутри ограниченного контекста склада содержит ссылки на места хранения, имея при этом только сведения о количестве. Таким образом, существует исключительно внутреннее и внешнее представление, которое мы раскрываем. Часто, когда имеются оба вида представления, может быть полезно называть их по-разному, чтобы избежать путаницы, например назвать общую `Единицу хранения` — `Количеством запасов`.

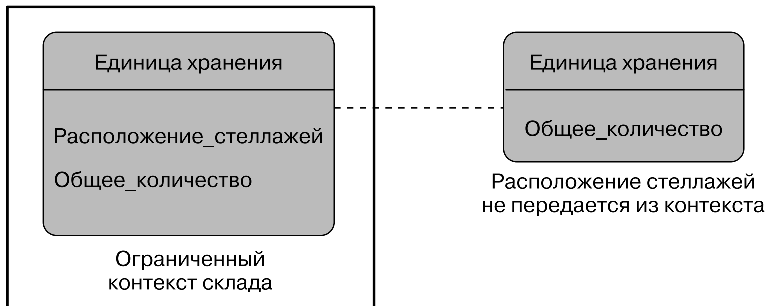


Рис. 2.15. Общая модель может решить скрыть информацию, которая не должна передаваться во внешний мир

Общая модель

Могут быть и понятия, появляющиеся более чем в одном ограниченном контексте. Склад и отдел финансов должны обладать информацией о нашем клиенте. Финансовому отделу необходимо знать о его платежах, в то время как склад для отслеживания поставок должен иметь представление, какие посылки были отправлены.

Когда возникает подобная ситуация, общая модель, такая как `Покупатель`, может иметь разные значения в различных ограниченных контекстах и, следовательно, может называться по-разному. Разумно сохранить название `Покупатель` в финансах, а для склада использовать имя `Получатель`, поскольку именно такую роль клиенты играют в этом контексте. Мы храним информацию о клиенте в обоих местах, но она различается. Финансовый контекст хранит сведения о платежах (или возвратах) клиента, склад — об отгруженных товарах. Нам все еще может понадобиться связать обе локальные концепции с глобальным покупателем и найти общую, совместно используемую информацию об этом клиенте, такую как его имя или адрес электронной почты. Для достижения этой цели допустимо использовать метод, подобный показанному на рис. 2.13.

Сопоставление агрегатов и ограниченных контекстов с микросервисами

Агрегаты и ограниченный контекст дают нам единицы связности с четко определенными интерфейсами с более широкой системой. Агрегат — это независимый конечный автомат, который фокусируется на концепции одной предметной области в нашей системе, при этом ограниченный контекст представляет собой набор связанных агрегатов, опять же с явным интерфейсом связи с внешними потребителями.

Следовательно, оба они могут хорошо работать в качестве границ сервисов. Как уже упоминалось, в начале работы всегда хочется сократить количество сервисов, с которыми предстоит работать. В результате следует ориентироваться на сервисы, охватывающие целые ограниченные контексты. Когда вы освоитесь и решите разбить эти сервисы на более мелкие, учтите, что агрегаты лучше не разделять — один микросервис может управлять разным количеством агрегатов, однако наиболее благоприятный вариант — когда одним агрегатом управляет один микросервис.

Черепахи — и нет им конца[1]

Сначала вы, вероятно, определите ряд общих ограниченных контекстов. Но они, в свою очередь, могут содержать другие ограниченные контексты. Например, можно разложить сервис `Склад` на функции, связанные с выполнением заказов, управлением запасами или получением товаров. При рассмотрении границ микросервисов в первую очередь подумайте о более крупных, общих контекстах, а затем разделите их по вложенным контекстам.

Вся хитрость в том, что, даже если вы решите позже разделить сервис, моделирующий весь ограниченный контекст, на более мелкие сервисы, вы все равно сможете скрыть это действие от внешних потребителей, например представив им общий крупномодульный API. Решение разбить сервис, возможно, станет шагом к реализации, поэтому по возможности его надо также скрыть. На рис. 2.16 показан такой пример. Мы разделили `Склад` на `Запасы` и `Доставку`. Что касается внешних потребителей, то для них по-прежнему существует только микросервис `Склад`. Однако внутренне мы еще больше разбили процессы, чтобы позволить сервису `Запасы` управлять `Единицами хранения` и позволить `Доставке` управлять `Отгрузками`. Помните, что мы хотим сохранить принадлежность одного агрегата одному микросервису.

Это еще одна форма скрытия информации — мы скрыли решение о внутренней реализации таким образом, что, если эта деталь реализации снова изменится в будущем, наши потребители не узнают.

---

[1] Отсылка к одноименной книге Джона Грина или к кантри-песне. Еще так называется стиль программирования с глубоким стеком вызовов. — *Примеч. пер.*

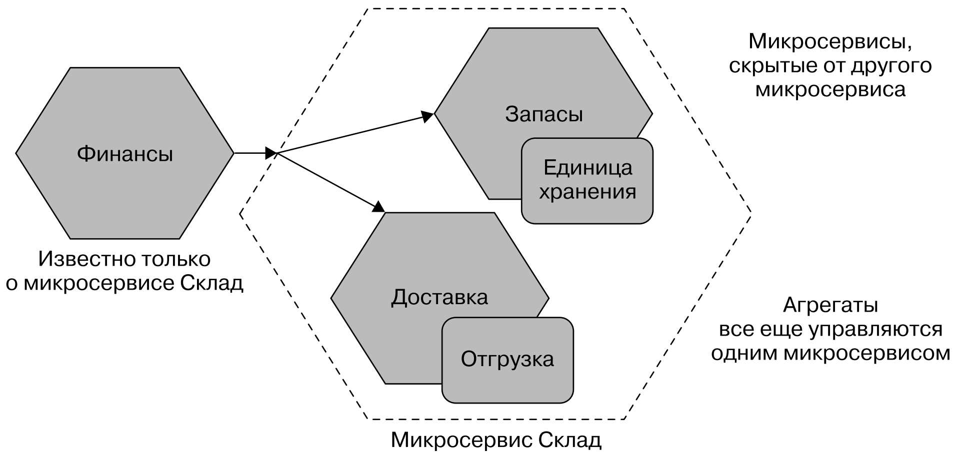


Рис. 2.16. Сервис Склад внутренне был разделен на микросервисы Запасы и Доставка

Вложенный подход стоит также предпочесть при разделении архитектуры на фрагменты для упрощения тестирования. Например, при тестировании сервисов, использующих склад, не требуется заключать каждый сервис внутри контекста хранилища — только более общий API. Это может дать вам единицу изоляции при рассмотрении крупномасштабных тестов. Я могу, например, решить провести сквозные тесты, в которых будут запущены все сервисы внутри контекста хранилища, но для остальных сотрудников я могу их отключить. Подробнее о тестировании и изоляции мы поговорим в главе 9.

Метод Event Storming

Event Storming (иногда называют *штурмом событий*) — метод, разработанный Альберто Брандолини и представляющий собой практику мозгового штурма, призванную помочь выявить модель предметной области. Метод Event Storming позволяет привлечь к совместному обсуждению технических и нетехнических специалистов. Идея заключается в превращении разработки модели предметной области в совместную деятельность, в результате которой вы получаете общий, объединенный взгляд на задачу.

На текущем этапе стоит упомянуть, что, хотя модели предметной области, определенные в процессе Event Storming, могут применяться для реализации систем, управляемых событиями, — и действительно, сопоставление очень простое, — вы также можете использовать такую модель предметной области для построения системы, более ориентированной на принцип «запрос — ответ».

Логистика

У Альберто есть несколько очень интересных идей относительно порядка проведения Event Storming, и с некоторыми из них я полностью согласен. Сначала соберите всех сотрудников в одном помещении. Часто это самый трудный шаг: согласование расписания, поиск достаточно большого помещения. Все эти проблемы были актуальны в мире и до COVID, но во время карантина эта задача может стать еще более сложной. Однако присутствие всех сотрудников одновременно — ключевой момент. Необходимы представители всех сфер предметной области, которую вы планируете моделировать: пользователи, эксперты по предметной области, владельцы продуктов и т. д.

Как только люди соберутся в одном помещении, Альберто предлагает убрать все стулья, чтобы никто не смог отсидеться в стороне. Я понимаю эту стратегию, но, как человек с больной спиной, признаю, что она не всем пойдет на пользу. В чем я действительно согласен с Альберто, так это в необходимости подготовить большое пространство, где можно было бы заниматься моделированием. Популярное решение — клеить стикеры на стены, что позволяет занять все пространство для сбора информации, причем разные цвета заметок должны соответствовать различным концепциям.

Процесс

Упражнение начинается с определения *событий предметной области*. Этим событиям, происходящим в системе, уделяется больше всего внимания. «Заказ размещен» могло бы стать событием, к которому мы проявляли интерес в контексте MusicCorp, как и «Оплата получена». Они запечатлены на оранжевых стикерах. Именно в этот момент я снова не согласен с Альберто. События — это чуть ли не самое многочисленное, что необходимо фиксировать, а оранжевых стикеров не так уж и много[1].

Затем участники определяют команды, вызывающие эти события. Команда — это решение, принятое человеком (пользователем ПО), за которым следует выполнение какого-то действия. В этот момент предпринимается попытка понять границы системы и определить ключевых людей, действующих в ней. Команды записываются на синих стикерах. Технари, присутствующие на встрече, должны прислушиваться к тому, что здесь предлагают их нетехнические коллеги. Ключевая часть этого упражнения — не позволить какой-либо текущей реализации исказить восприятие предметной области (это будет позже). На текущем этапе предпринимается попытка создать пространство, в котором все присутствующие должны изложить свои идеи и видение системы.

После того как события и команды определены, приходит очередь агрегатов. Этот этап полезен для обмена информацией о происходящем в системе

[1] Я имею в виду, почему не желтые? Это самый распространенный цвет!

и о потенциальных агрегатах. Подумайте о вышеупомянутом событии предметной области «Заказ размещен». Существительное — «Заказ» — вполне может быть потенциальным агрегатом, а слово «размещен» вполне может быть частью ее жизненного цикла. Агрегаты представлены желтыми стикерами, а связанные с ними команды и события перемещаются и группируются вокруг них. Это также поможет понять, как агрегаты связаны между собой — события из одного агрегата могут повлиять на поведение в другом.

После определения агрегатов они группируются в ограниченные контексты, которые чаще всего соответствуют организационной структуре компании, а участники упражнения получают полное понимание того, кто, как и какие агрегаты должен использовать.

Конечно, я привел лишь краткий обзор метода Event Storming. Для более подробного ознакомления я бы посоветовал вам прочитать одноименную книгу Альберто Брандолини[1].

Аргументы в пользу предметно-ориентированного проектирования микросервисов

Мы рассмотрели, как DDD может работать в контексте микросервисов, поэтому давайте подведем итог тому, насколько этот подход полезен для нас.

Во-первых, основная причина эффективности подхода DDD заключается в ограниченных контекстах, предназначенных для скрытия информации. Их применение дает возможность представить четкую границу модели предметной области с точки зрения более широкой системы, скрывая внутреннюю сложность реализации. Это также позволяет вносить изменения, не затрагивающие другие части системы. Таким образом, следуя подходу DDD, мы используем скрытие информации, что жизненно важно для определения стабильных границ микросервиса.

Во-вторых, акцент на общем, едином языке очень помогает, когда дело доходит до определения конечных точек микросервиса. Это дает нам общий словарь, на который можно опираться при разработке API, форматов событий и т. п. Это также помогает решить проблему стандартизации API с точки зрения возможности изменения языка в ограниченных контекстах — изменения внутри границы, влияющего на саму границу.

Преобразования, которые мы вносим в систему, часто связаны с изменениями, которые бизнес хочет внести в поведение системы. Мы меняем функциональность — возможности, доступные нашим клиентам. Если наши системы разложены по ограниченным контекстам, которые представляют нашу предметную область, любые желаемые модификации с большей вероятностью будут изо-

[1] *Brandolini A.* Introducing EventStorming. — Victoria, BC: Leanpub, forthcoming.

лированы в пределах одной границы микросервиса. Это сокращает количество мест, в которых требуется внести изменения, и позволяет быстро их внедрять.

По сути, DDD ставит бизнес-сферу в центр создаваемого нами приложения. Использование языка бизнеса в нашем коде помогает улучшить знания предметной области среди разработчиков ПО. Это, в свою очередь, позволяет лучше понимать пользователей и улучшает коммуникацию между техподдержкой, разработчиками и конечными клиентами. Если вы заинтересованы в переходе к потоковым командам, DDD идеально подойдет в качестве механизма, помогающего согласовать техническую архитектуру с более широкой организационной структурой. В мире, в котором мы все чаще пытаемся разрушить барьеры между ИТ и «бизнесом», это не так уж плохо.

Альтернативы границам предметной области бизнеса

Как я уже говорил, подход DDD может быть невероятно полезен при построении микросервисных архитектур, однако это не единственный метод, применяемый при определении границ микросервиса. Я часто использую несколько методов в сочетании с DDD, чтобы определить, как система должна быть разделена. Давайте рассмотрим некоторые другие факторы, которые нужно учитывать при определении границ.

Волатильность

Я все чаще слышу о неприятии предметно-ориентированной декомпозиции, обычно от сторонников того, что волатильность представляет собой основную движущую силу декомпозиции. Декомпозиция на основе волатильности позволяет определить, какие части вашей системы часто изменяются, а затем извлечь эту функциональность в отдельные сервисы и таким образом более эффективно работать с ней. Я не возражаю против этого метода, но продвигать его как единственный способ реализации бесполезно, особенно если учесть различные факторы, подталкивающие нас к внедрению микросервисов. Например, если самая большая проблема связана с необходимостью масштабирования приложения, декомпозиция на основе волатильности вряд ли принесет большую пользу.

Идея, лежащая в основе декомпозиции на базе волатильности, проявляется и в бимодальных ИТ-моделях. Концепция бимодальной ИТ-модели, предложенная компанией Gartner, четко разделяет мир на категории с краткими названиями «режим 1» (или «системы учета») и «режим 2» (или «системы инноваций») в зависимости от скорости работы различных систем. Системы режима 1 мало меняются и не требуют серьезного участия бизнеса, а в режиме 2 происходит действие с системами, требующими быстрых изменений и тесного вовлечения

бизнеса. Если отбросить резкое упрощение, такая схема подразумевает очень фиксированный взгляд на мир и противоречит очевидным во всей отрасли трансформациям, поскольку компании стремятся «перейти на цифровые технологии». У различных компаний есть системы обработки бизнес-процессов. Некоторые части этих систем ранее не нуждались в значительных преобразованиях, а теперь внезапно меняются для соответствия потребностям рынка.

Давайте вернемся к компании MusicCorp. Ее первым шагом на пути к цифровым технологиям было просто создание веб-страницы. Все, что она предлагала еще в середине девяностых, — список выставленного на продажу. Однако для оформления заказа необходимо было звонить в MusicCorp. Напоминало объявление в газете. Затем онлайн-заказы стали обычным делом, и весь ассортимент, который до этого момента обрабатывался только на бумаге, пришлось оцифровывать. Кто знает, возможно, MusicCorp на каком-то этапе придется подумать и о продажах музыки в электронном виде! Можете оценить масштаб потрясений, через которые проходят фирмы во время технологических преобразований.

Мне не нравится бимодальная ИТ-модель как концепция, поскольку она дает людям возможность упаковать то, что трудно изменить, в красивую аккуратную коробку и сказать: «Нам не нужно разбираться с проблемами там — это режим 1». Это еще одна модель, которую компании могут принять, чтобы объяснить, почему они не меняются. Ведь довольно часто изменения в функциональности требуют преобразований в системах учета (режим 1), чтобы можно было учесть изменения в системах инноваций (режим 2). По моему опыту, организации, внедряющие бимодальные ИТ-модели, в конечном счете получают два режима — медленный и еще медленнее.

Справедливости ради отмечу, что многие из сторонников декомпозиции на основе волатильности не всегда рекомендуют такие упрощенные концепции, как бимодальные ИТ-модели. Я считаю, что этот метод очень полезен для определения границ, если основной запрос — быстрое время выхода на рынок. В таком случае извлечение часто изменяемой или требующей изменения функциональности имеет смысл. Но опять же наиболее подходящий механизм нужно определять, ориентируясь на цель.

Данные

Характер данных, которые вы храните и которыми управляете, подталкивает к различным формам декомпозиции. Например, может потребоваться внести ограничения, чтобы определить, какие сервисы могут обрабатывать личную информацию (PII). Это позволит снизить риск утечки и упростит контроль данных, а также поспособствует соблюдению таких регламентов, как GDPR.

Для одного из моих недавних заказчиков — платежной компании, назовем ее PaymentCo, — использование определенных типов данных напрямую повлияло на принимаемые решения в отношении декомпозиции. PaymentCo обрабатывает данные кредитных карт. Поэтому ее система должна соответствовать различным

требованиям, установленным стандартами индустрии платежных карт (PCI, payment card industry) в отношении управления этими данными. У PaymentCo была необходимость обрабатывать полные данные данные кредитных карт в объеме, соответствующем PCI уровня 1. Это самый строгий уровень, и он требует ежеквартальной внешней оценки систем и методов, связанных с управлением данными.

Многие требования PCI логичны, но сложно обеспечить соответствие всей системы этим условиям, а необходимость проведения аудита кажется довольно обременительной. В результате компания захотела выделить часть системы, обрабатывающую полные данные кредитных карт, чтобы только ее обеспечить дополнительным уровнем контроля. На рис. 2.17 показана упрощенная форма придуманного нами проекта. Сервисы, работающие в зеленой зоне, не видят информации о кредитной карте — эти данные ограничены процессами (и сетями) в красной зоне. Шлюз перенаправляет вызовы на соответствующие сервисы (и в соответствующую зону). Так как информация о кредитной карте проходит через этот шлюз, она фактически также находится в красной зоне.

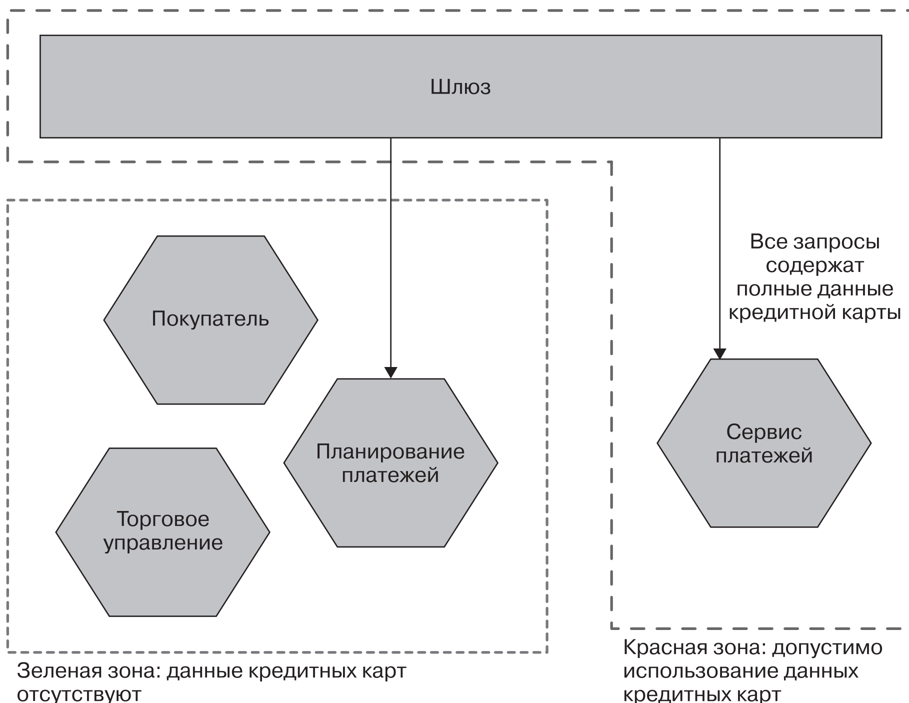


Рис. 2.17. Компания PaymentCo, разделяющая процессы на основе использования информации о кредитной карте, чтобы уменьшить объем требований PCI

Поскольку информация о кредитной карте никогда не попадает в зеленую зону, все сервисы в этой области могут быть освобождены от полного аудита

PCI, в отличие от сервисов красной зоны. При разработке проекта мы сделали все возможное, чтобы ограничить то, что должно находиться в красной зоне. Необходимо было убедиться, что информация о кредитной карте вообще никогда не поступает в зеленую зону: если микросервис в зеленой зоне может запросить эту информацию или если эта информация может быть отправлена обратно в зеленую зону микросервисом в красной зоне, то четкие разделительные линии будут нарушены.

Разделение данных часто обусловлено различными соображениями конфиденциальности и безопасности. Мы вернемся к этой теме и к примеру с PaymentCo в главе 11.

Технологии

Необходимость использования различных технологий также может быть фактором, влияющим на определение границ. Допускается размещать разные базы данных в одном запущенном микросервисе, но, если требуется смешивать несколько моделей среды выполнения, будьте готовы к проблемам. Если вы для улучшения производительности решите реализовать часть функциональности в среде выполнения, подобной Rust, это в итоге станет основным влияющим фактором.

Конечно, необходимо отдавать себе отчет, к чему может привести принятие технологий в качестве основы декомпозиции. Классическая трехуровневая архитектура, описанная в главе 1 и снова показанная на рис. 2.18, представляет собой пример объединения связанных технологий. Как мы уже выяснили, часто это далеко не идеальная архитектура.

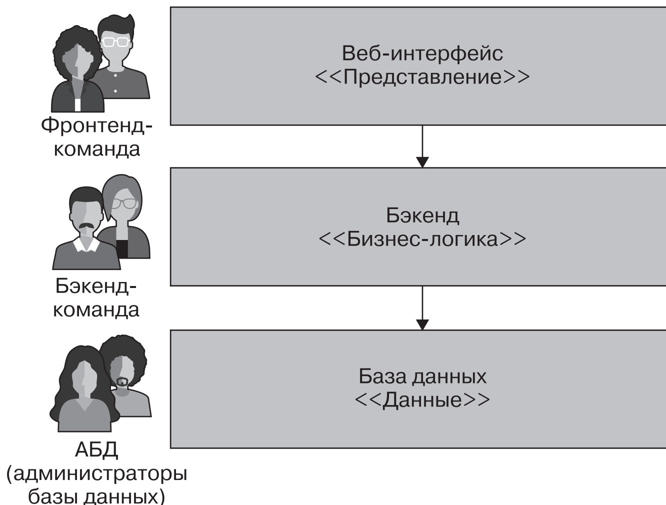


Рис. 2.18. Традиционная трехуровневая архитектура часто определяется технологическими границами

Организационный подход

Рассматривая закон Конвея в главе 1, мы установили, что существует неотъемлемая взаимосвязь между организационной структурой и получаемой в итоге архитектурой системы. На своем собственном опыте я убедился, что самоорганизация в конечном счете определяет архитектуру вашей системы, хорошо это или плохо. Когда дело доходит до определения границы сервисов, стоит рассматривать это как ключевую часть процесса принятия решений.

Определение границы сервиса, владение которым будет распространяться на несколько разных команд, вряд ли приведет к желаемым результатам (подробнее — в главе 15). Совместное владение микросервисами сопряжено с большими трудностями. Отсюда вытекает необходимость принимать во внимание существующую организационную структуру при рассмотрении вопроса об определении границ. А в некоторых ситуациях даже следует рассмотреть возможность преобразования организационной структуры для поддержки желаемой архитектуры.

Конечно, важно учитывать и изменения организационной структуры. Предстоит ли нам в связи с этим перестраивать ПО? В худшем случае это может привести к пересмотру требующего разделения существующего микросервиса, поскольку он содержит функциональность, принадлежащую теперь двум отдельным командам, тогда как раньше за сервис отвечала одна. С другой стороны, часто организационные изменения требуют только смены владельца существующего микросервиса. Рассмотрим ситуацию, когда команда, отвечающая за складские операции, ранее также выполняла функции, связанные с определением количества товаров, которые следует заказать у поставщиков. Допустим, было принято решение передать эту ответственность команде по прогнозированию. Такой команде необходимо получить информацию о текущих продажах и планируемых рекламных акциях, чтобы определить, что нужно заказать. Если бы у команды, отвечающей за работу склада, имелся выделенный микросервис `Заказ поставщику`, его можно было бы просто перенести в новую команду прогнозирования. С другой стороны, если эта функциональность ранее была интегрирована в систему с более широким охватом, принадлежащую команде, владеющей сервисом `Склад`, то ее, возможно, потребуется отделить.

Даже когда мы работаем в рамках существующей организационной структуры, не исключен риск неверно установить границы. Много лет назад мы с несколькими коллегами работали с клиентом в Калифорнии, помогая компании внедрить некоторые более «чистые» методы кодирования и перейти к автоматизированному тестированию. Мы начали с простого — декомпозиции сервиса, но заметили кое-что странное. Я не могу вдаваться в подробности, но это было общедоступное приложение с большой глобальной клиентской базой.

Команда и система увеличились. Первоначально задуманная одним человеком система все больше и больше обрастала функциями и пользователями.

В конце концов, организация решила усилить потенциал команды, пригласив новую группу разработчиков из Бразилии, которые взяли бы на себя часть работы. Система была разделена, причем пользовательская половина приложения, по сути, не имела состояния и реализовывала сайт, как показано на рис. 2.19. Серверная же часть системы представляла собой просто интерфейс удаленного вызова процедур (RPC, remote procedure call) через хранилище данных. Представьте, что вы взяли уровень репозитория в своей кодовой базе и сделали его отдельным сервисом.

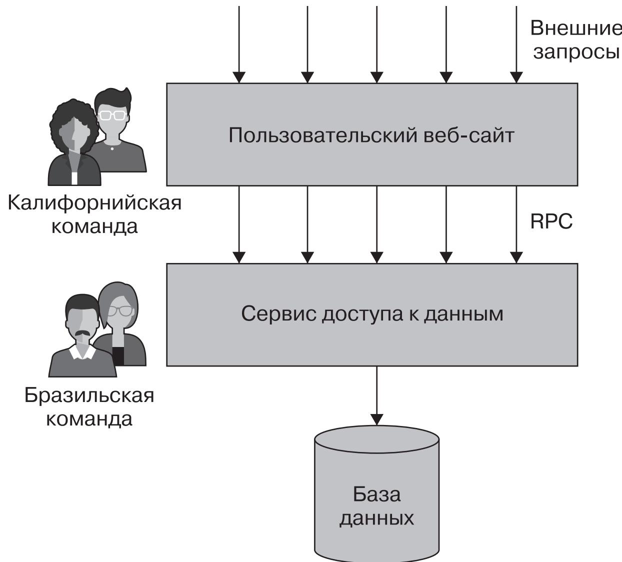


Рис. 2.19. Граница сервисов, образованная техническими стыками

Оба сервиса часто приходилось модифицировать. Они использовали низкоуровневые вызовы методов в стиле RPC, которые были слишком уязвимыми (мы обсудим это в главе 4). Интерфейс сервиса также очень интенсивно обменивался информацией, что приводило к проблемам с производительностью. Нам пришлось создать сложные механизмы дозирования RPC. Я назвал это «луковой архитектурой», так как в ней было много слоев и я плакал, когда нам приходилось сквозь них прорезаться.

Теперь, на первый взгляд, идея разделения ранее монолитной системы по географическому/организационному признаку обрела смысл, о чем мы подробнее поговорим в главе 15. Здесь, однако, вместо того, чтобы делать вертикальный, бизнес-ориентированный срез стека, команда выделила то, что ранее являлось встроенным API, и сделала горизонтальный срез. Лучшим вариантом было бы, если бы команда в Калифорнии сделала один сквозной вертикальный срез, со-

стоящий из связанных частей пользовательского интерфейса и функций доступа к данным, а команда в Бразилии сделала другой срез.

> **ВНУТРЕННЕЕ И ВНЕШНЕЕ НАСЛОЕНИЕ**
>
> Надеюсь, вы уже поняли, что я не поклонник горизонтально-слоистой архитектуры. Однако многослойность может иметь смысл. В пределах границ микросервиса вполне разумно разделять различные уровни, чтобы упростить управление кодом. Проблема возникает, когда это расслоение становится механизмом определения границ вашего микросервиса и определения владения.

Смешивание моделей и исключений

Надеюсь, к настоящему моменту вы поняли, что я не навязываю конкретный подход к поиску границ. Если вы будете следовать рекомендациям по скрытию информации и учитывать взаимодействие связанности и связности, то, скорее всего, сможете избежать некоторых недостатков любого выбранного механизма. Мне кажется, что, сосредоточившись на этих идеях, вы с *большей* вероятностью получите предметно-ориентированную архитектуру. Со временем. Однако факт в том, что часто могут возникать причины для смешивания моделей, даже если вы решите выбрать «предметно-ориентированную» модель в качестве основного механизма определения границ микросервиса.

Между различными механизмами, описанными ранее, существует множество потенциальных взаимосвязей. Если не взглянуть на выбор механизмов широко, придется следовать сценарию, определенному для конкретного механизма, а не делать то, что лучше для системы. Декомпозиция на основе волатильности не бессмысленна, если вы сосредоточены на повышении скорости доставки. Но если это заставляет вас извлекать сервис, выходящий за рамки организационных границ, то будьте готовы, что темпы изменений снизятся из-за проблем с доставкой.

Можно было бы определить хороший сервис `Склад`, основываясь на своем понимании предметной области бизнеса, но, если одна часть этой системы реализована, например, на C++, а другая — на Kotlin, вам придется проводить декомпозицию дальше по линиям, проведенным на основе технической составляющей системы.

Организационные и предметно-ориентированные границы сервисов — это моя собственная отправная точка, мой подход по умолчанию. Как правило, в игру вступает ряд описанных здесь факторов, и то, какие из них повлияют на ваши собственные решения, будет зависеть от решаемых вами задач. Вам необходимо ориентироваться на свои конкретные условия, чтобы определить, что лучше всего подходит именно для вас, и надеюсь, я дал вам несколько вариантов для рассмотрения. Просто помните, что если кто-то говорит: «Единственный способ сделать это — X!» — он, скорее всего, пытается убедить вас

использовать конкретный сценарий, в пределах которого вам придется работать. Но вы можете найти решение получше.

Учитывая все сказанное, давайте глубже погрузимся в тему моделирования предметной области, изучив предметно-ориентированное проектирование немного подробнее.

Резюме

В этой главе вы узнали, какими должны быть хорошие границы микросервиса и как находить в проблемном пространстве стыки, которые позволят получить двойные преимущества как слабой связанности, так и сильной связности. Детальное понимание предметной области может стать жизненно важным инструментом, помогающим найти эти стыки, и, согласовывая микросервисы с этими границами, мы гарантируем, что результирующая система сохранит эти свойства нетронутыми. Также вы узнали, как можно еще больше разделить микросервисы.

Идеи, изложенные в книге Эрика Эванса «Предметно-ориентированное проектирование», очень полезны для поиска разумных границ сервисов. Я же только прошелся по верхам — в книге Эрика гораздо больше подробностей. Если вы хотите углубиться в эту тему, я могу порекомендовать книгу Вона Вернона «Реализация методов предметно-ориентированного проектирования»[1], — она поможет вам понять практические аспекты этого подхода, в то время как другая книга Вернона[2] про основы предметно-ориентированного программирования отлично подойдет, если вы ищете что-то более конкретное.

Большая часть этой главы посвящена описанию поиска границы для микросервисов. Но что произойдет, если у вас уже есть монолитное приложение и вы хотите перейти на микросервисную архитектуру? Далее мы рассмотрим такой случай.

[1] *Вернон В.* Реализация методов предметно-ориентированного проектирования.

[2] *Вернон В.* Предметно-ориентированное проектирование. Самое основное.

ГЛАВА 3

Разделение монолита на части

У многих из вас, вероятно, нет возможности начать разработку системы с «чистого листа», и, даже если есть, начинать с микросервисов может оказаться не очень хорошей идеей по тем причинам, которые мы рассмотрели в главе 1. У некоторых из вас уже есть существующая система, возможно какой-то вариант монолитной архитектуры, которую вы хотите превратить в микросервисную.

В этой главе я расскажу, с чего начать и какие шаблоны лучше использовать.

Осознайте цель

Микросервисы не должны быть самоцелью. Вы ничего не выиграете просто от их присутствия в системе. Выбор микросервисной архитектуры — это осознанное, рациональное решение. Думать о переходе следует только в том случае, если вы не можете найти более простого способа достичь своей конечной цели имеющимися средствами.

Без четкого понимания целей создания системы можно попасть в ловушку, перепутав деятельность с результатом. Я видел команды, одержимые идеей создания микросервисов, которые никогда задавались вопросом, зачем они им. И это крайне неразумно, учитывая сложности, которые могут привнести микросервисы.

Зацикленность на микросервисах, а не на конечной цели грозит тем, что вы, скорее всего, перестанете рассматривать другие способы добиться желаемых изменений. Например, микросервисы, безусловно, способны помочь масштабировать систему, но есть и ряд альтернативных методов, на которые следует обратить внимание в первую очередь. Развертывание еще нескольких копий существующей монолитной системы за балансировщиком нагрузки вполне может помочь вам масштабировать систему гораздо эффективнее, чем сложная и длительная декомпозиция на микросервисы.

Микросервисы — это не самый легкий вариант. Попробуйте для начала более простые подходы.

Какой микросервис необходимо создать в первую очередь? Без всеобъемлющего понимания конечной цели ответить на этот вопрос практически невозможно.

Поэтому четко определите, каких изменений вы пытаетесь добиться, и поищите более простые способы их реализации, прежде чем рассматривать микросервисы. И если микросервисы все же окажутся лучшим вариантом, то отслеживайте свой прогресс и при необходимости меняйте курс.

Постепенный переход

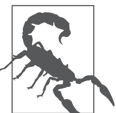
> Если вы занимаетесь чем-то хаотично, единственное, что вы гарантированно получите, — это хаос.
>
> *Мартин Фаулер*

Если вы все же придете к выводу, что разделять существующую монолитную систему на части — это правильно, я настоятельно рекомендую вам делать это постепенно. Поэтапный подход позволит изучать микросервисы постепенно, по ходу работы, а также ограничит отрицательное влияние ошибочных изменений (а они точно будут!). Представьте себе монолит в виде мраморной глыбы. Можно было бы ее взорвать, но подобные действия редко заканчиваются благополучно. Гораздо разумнее откалывать от монолита по кусочку.

Разбейте свою большую работу по переходу к микросервисам на множество маленьких этапов — из выполнения каждого из них можно будет извлечь уроки. Если это и окажется шагом назад, то лишь небольшим. В любом случае вы учитесь, и каждый следующий предпринимаемый вами шаг будет основан на опыте предыдущих.

Разбиение фрагментов системы на более мелкие части также позволяет вам не застрять на месте и постепенно набрать обороты. Отделяя микросервисы по одному, вы получаете возможность постепенно раскрывать приносимую ими пользу вместо ожидания какого-то масштабного развертывания.

Все это сводится к основному совету для людей, изучающих микросервисы: если вы считаете, что их внедрение — это хорошая идея, начните с малого. Выберите одну или две функции, реализуйте их как микросервисы и внедрите в ваш продукт, а затем подумайте, помогло ли вам это приблизиться к конечной цели.



Вы не сможете оценить весь масштаб проблем, которые способна принести микросервисная архитектура, пока не запустите систему в работу.

Монолит не всегда плохой вариант

Хотя в начале книги я уже приводил доводы в пользу того, что та или иная форма монолитной архитектуры может быть абсолютно правильным выбором, стоит повторить, что монолитная архитектура не является *плохой* по своей сути и поэтому не должна восприниматься враждебно. Не зацикливайтесь на отказе от монолита. Лучше сосредоточьтесь на преимуществах, которые должны принести изменения вашей архитектуры.

Обычно существующая монолитная архитектура сохраняется после перехода к микросервисам, хотя часто ее функциональность оказывается урезанной. Например, улучшить стабильность приложения можно, если извлечь 10 % функциональности, которая в настоящее время является узким местом, а 90 % оставить в монолитной системе.

Многие люди считают сосуществование монолита и микросервисов «запутанным», но архитектура реально работающей системы никогда не бывает чистой или простой. Если вам нужна чистая архитектура, обязательно заламинируйте распечатку идеальной версии архитектуры системы, которая могла бы у вас быть, если бы вы обладали даром предвидения и безграничными средствами. Архитектура реальной системы постоянно развивается, требует адаптации по мере изменения потребностей и получения новых знаний. Настоящее мастерство заключается в том, чтобы привыкнуть к этой идее, к чему я вернусь в главе 16.

Благодаря тому, что работа по переходу на микросервисы будет выполняться поэтапно, вы можете «откалывать» фрагменты, попутно внося улучшения, а также, что немаловажно, зная, когда остановиться.

В удивительно редких случаях полное разделение монолита может оказаться труднодостижимым требованием. По моему опыту, так часто бывает, когда существующий монолит основан на устаревшей или умирающей технологии, привязан к инфраструктуре, которую необходимо удалить, или эта инфраструктура представлена дорогостоящей сторонней системой, от которой вы предпочли бы отказаться. Даже в этих ситуациях по изложенным мной причинам рекомендуется использовать поэтапный подход к декомпозиции.

Опасность преждевременной декомпозиции

Не спешите создавать микросервисы, если у вас неясное представление о предметной области. В моей практике такое было, когда я работал в компании Thoughtworks. Одним из ее продуктов был Snap CI — размещенный на хостинге инструмент непрерывной интеграции и непрерывной доставки (мы обсудим эти концепции в главе 7). Команда ранее работала над аналогичным приложением — GoCD — инструментом непрерывной доставки с открытым исходным кодом, который можно развертывать локально, а не размещать в облаке.

Хотя на самом раннем этапе определенная часть кода в проектах Snap CI и GoCD повторялась, в итоге у Snap CI оказалась совершенно новая кодовая база. Тем не менее предыдущий опыт команды в области инструментов CD (непрерывной доставки) позволил разработчикам быстрее определить границы и построить свою систему как набор микросервисов.

Однако через несколько месяцев стало ясно, что варианты использования Snap CI настолько сильно различаются, что первоначальное представление о границах сервисов было не совсем правильным. Это привело к большому количеству изменений, внесенных во все сервисы, и к возросшим издержкам. В конце концов разработчики объединили сервисы обратно в одну монолитную систему, дав членам команды время для более четкого определения границ сервисов. Год спустя они смогли разделить монолитную систему на микросервисы, границы которых оказались гораздо более стабильными. Это далеко не единственный пример подобной ситуации. Преждевременная декомпозиция системы может оказаться дорогостоящей, особенно если вы новичок в этой области. Во многих отношениях работать с существующей кодовой базой, требующей разделения на микросервисы, намного проще, чем пытаться создавать микросервисы с самого начала.

Что отделить в первую очередь

Как только у вас появится четкое представление о необходимости внедрения микросервисов, вам потребуется определить, какие микросервисы создавать в первую очередь. Хотите масштабировать приложение? Функции, в настоящее время ограничивающие способность системы справляться с нагрузкой, будут занимать первое место в списке. Хотите ускорить время выхода на рынок? Посмотрите на изменчивость системы, чтобы определить наиболее часто изменяющиеся части функциональности, и поймите, будут ли они работать как микросервисы. Для быстрого поиска наиболее изменчивых частей кодовой базы можно использовать инструменты статического анализа, такие как CodeScene (https://www.codescene.com). Пример работы CodeScene показан на рис. 3.1, где изображены хот-споты в проекте Apache Zookeeper с открытым исходным кодом.

Но также необходимо учитывать, какие варианты декомпозиции будут жизнеспособными. Некоторые функции могут оказаться настолько глубоко встроены в существующее монолитное приложение, что будет невозможно понять, как их извлечь. Или, возможно, рассматриваемая функциональность настолько важна для приложения, что любые модификации связаны с высоким риском. Или, наоборот, функциональность, которую вы хотите перенести, уже может быть автономной, и поэтому извлечение покажется очень простым.

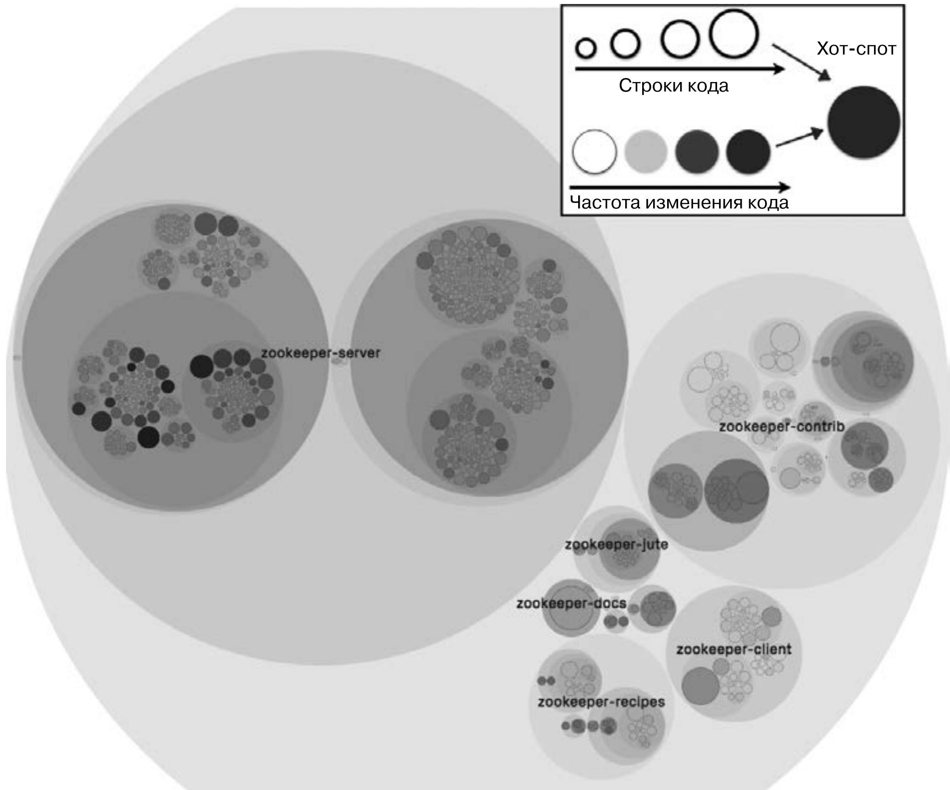


Рис. 3.1. Хот-споты в CodeScene, помогающие идентифицировать часто изменяющиеся части кодовой базы

Решение, какую функцию превратить в микросервис, в конечном счете будет представлять собой баланс между двумя факторами: насколько просто извлечение и насколько оно выгодно.

Мой совет для первой пары микросервисов: выбирайте что-то попроще — нечто оказывающее определенное влияние на достижение конечной цели, но в то же время это должен быть достаточно простой и доступный вариант. При длительном переходе, особенно таком, который может занять месяцы или годы, важно на раннем этапе ощутить динамику, получить результат своей работы.

С другой стороны, если вы пытаетесь извлечь для создания микросервиса то, что считаете самым простым, и не можете заставить это работать, возможно, стоит подумать, действительно ли микросервисы подходят вам и вашей организации.

Добившись некоторых успехов и усвоив несколько уроков, вы сможете гораздо эффективнее справляться с более сложными извлечениями функциональности.

Декомпозиция по слоям

Итак, вы определили, какой микросервис будете извлекать первым. Что дальше? Разобьем эту декомпозицию на дальнейшие, более мелкие этапы.

Если взять в расчет традиционные три уровня стека веб-сервисов, то можно рассмотреть функциональность, которую мы хотим извлечь, с точки зрения ее пользовательского интерфейса, серверной части кода приложения и данных.

Сопоставление микросервиса с пользовательским интерфейсом часто не соответствует соотношению 1:1 (подробнее об этом — в главе 14). Таким образом, извлечение функциональности UI, связанной с микросервисом, можно рассматривать как отдельный шаг. Хотелось бы предостеречь вас от игнорирования части уравнения, касающейся пользовательского интерфейса. Я видел слишком много организаций, думающих только о преимуществах декомпозиции функциональности серверной части (бэкенда), что часто приводило к нарушению целостности подхода к любой архитектурной реструктуризации. Иногда самые большие преимущества можно получить от декомпозиции UI, так что игнорируйте этот шаг на свой страх и риск. Часто декомпозиция UI имеет тенденцию отставать от декомпозиции бэкенда, поскольку до тех пор, пока микросервисы не будут доступны, трудно увидеть возможности декомпозиции пользовательского интерфейса. Просто убедитесь, что процесс не слишком сильно отстает.

Если мы рассмотрим код бэкенда и связанное с ним хранилище, то при извлечении микросервиса жизненно важно, чтобы оба они находились в области видимости. Давайте взглянем на рис. 3.2. На нем изображена попытка извлечь функциональность, связанную с управлением клиентским списком избранного. Существует некий код приложения, который расположен в монолите, и некое связанное с ним хранилище данных в БД. Итак, какую часть следует извлечь в первую очередь?

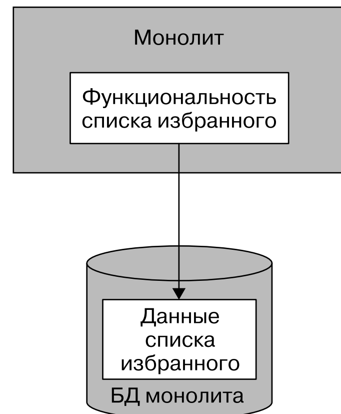


Рис. 3.2. Код и данные списка избранного в существующем монолитном приложении

Сначала код

На рис. 3.3 код, связанный с функциональностью списка избранного, извлечен в новый микросервис. На текущем этапе сведения для списка избранного остаются в базе данных монолита — декомпозиция не завершена, пока не перемещены данные, относящиеся к новому микросервису `Список избранного`.

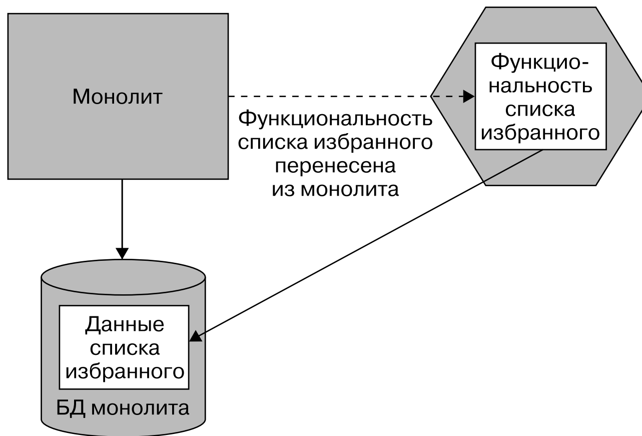


Рис. 3.3. Сначала переместите код списка избранного в новый микросервис, оставив данные в монолитной базе данных

По моему опыту, это самый распространенный первый шаг, так как он дает наибольшую краткосрочную выгоду. Если оставить данные в монолитной БД, в будущем накопится много проблем, которые тоже придется решать, однако мы уже многое выиграли от появления нового микросервиса.

Извлечение кода приложения обычно проще, чем извлечение данных из БД. Если выяснится, что извлечь код аккуратно невозможно, можно прервать любую дальнейшую работу и избежать необходимости распутывать базу данных. Однако если код приложения извлекается без ошибок, но извлечение информации оказывается невозможным, возникают проблемы. Таким образом, очень важно, чтобы вы заранее изучили соответствующее хранилище данных и пришли к выводу, реально ли произвести извлечение данных и как это осуществить. Поэтому перед началом декомпозиции продумайте, как будут извлекаться код приложения и данные.

Сначала данные

На рис. 3.4 показана ситуация, когда сначала извлекаются данные, а затем код приложения. Такой подход встречается реже, но он может быть полезен, когда нет уверенности в возможности четкого разделения данных.

Основное преимущество такого подхода (в краткосрочной перспективе) заключается в снижении рисков при полном извлечении микросервиса. Это вынуждает заранее решать такие проблемы, как потеря целостности данных в вашей БД или отсутствие транзакционных операций с обоими наборами данных. Мы кратко коснемся последствий обеих проблем позже в этой главе.

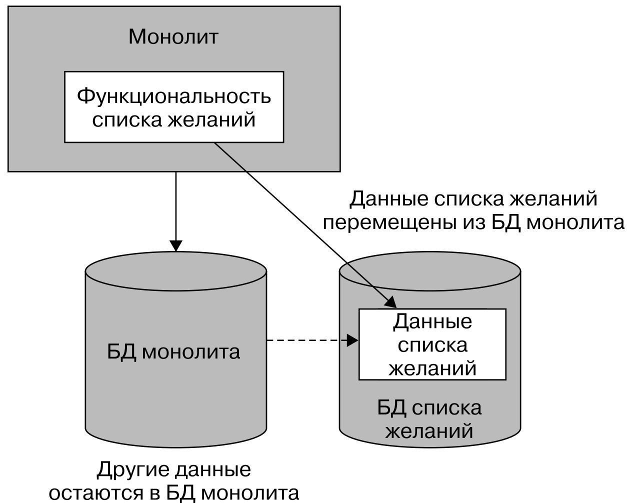


Рис. 3.4. Сначала извлекаются таблицы, связанные с функциональностью списка избранного

Полезные шаблоны декомпозиции

Для разделения существующей системы можно использовать ряд шаблонов. Многие из них подробно рассмотрены в моей книге «От монолита к микросервисам»[1]. Чтобы лишний раз не повторяться, поделюсь обзором лишь некоторых из них, давая вам представление об их возможностях.

Шаблон «Душитель»

Техника, которая часто используется при переписывании системы, — это *шаблон «Душитель»* (Strangler Fig Pattern), придуманный Мартином Фаулером (https://oreil.ly/u33bI). Данный шаблон описывает процесс объединения старой и новой систем с течением времени, позволяя актуальной версии постепенно перенимать все больше и больше функций старой системы.

Этот подход достаточно прост, он показан на рис. 3.5. Выполняется перехват вызовов существующей системы — в нашем случае монолитного приложения. Если вызов этой части функциональности реализован в новой микросервисной архитектуре, он перенаправляется на микросервис. Если функциональность по-прежнему обеспечивается монолитом, вызову разрешается продолжить выполнение до самого монолита.

[1] *Ньюмен С.* От монолита к микросервисам. Эволюционные шаблоны для трансформации монолитной системы.

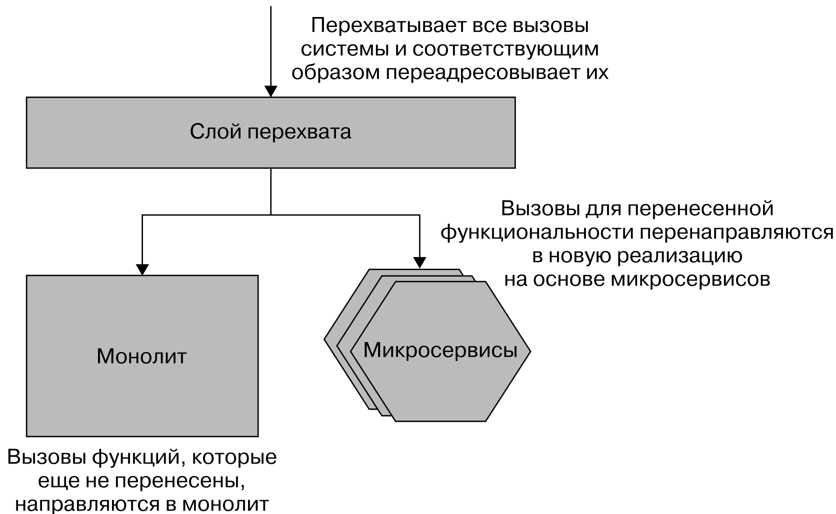


Рис. 3.5. Обзор шаблона «Душитель»

Прелесть этого шаблона заключается в возможности реализовать его без внесения каких-либо изменений в базовое монолитное приложение. Монолиту неизвестно, что был «обернут» в более новую систему.

Параллельное выполнение

Переход от старой функциональности к новой, основанной на микросервисах, — достаточно напряженный для разработчиков процесс, особенно если переносимая функциональность представляет решающее значение для вашей организации.

Один из способов убедиться в корректности работы новой функциональности, не подвергая риску поведение существующей системы, — использовать шаблон параллельного выполнения (parallel run): одновременное выполнение монолитной реализации функциональности и микросервисной, обслуживание одних и тех же запросов и сравнение результатов. Мы рассмотрим этот шаблон более подробно в подразделе «Параллельное выполнение» главы 8.

Шаблон переключаемых функций

Переключатель функций (feature toggle) — это механизм, позволяющий выключать или включать функцию или переключаться между двумя различными ее реализациями. У данного шаблона хорошая применимость во многих случаях, однако он особенно полезен при переходе к микросервисам.

Как и в случае с шаблоном «Душитель», во время перехода мы часто оставляем существующую функциональность в монолите, но рассматриваемый шаблон позволяет нам переключаться между версиями функциональности — в монолите и в новом микросервисе. В примере применения шаблона «Душитель» с использованием HTTP-прокси можно было бы реализовать переключение функций на уровне прокси, чтобы обеспечить простой элемент управления.

Для более широкого ознакомления с переключателями функций я рекомендую статью Пита Ходжсона[1].

Проблемы декомпозиции данных

Во время разбиения БД на части может возникнуть ряд проблем, о которых мы поговорим ниже.

Производительность

Базы данных, особенно реляционные, очень хороши для объединения данных в разных таблицах. Однако нередко при разделении БД на части нам в конечном счете приходится перемещать операции объединения (JOIN) с уровня данных в сами микросервисы. И как бы мы ни старались, вряд ли их выполнение будет столь же быстрым.

Рассмотрим рис. 3.6. Он иллюстрирует ситуацию, в которой мы оказались с компанией MusicCorp. Мы решили извлечь из каталога функциональность — нечто, что может предоставлять информацию об исполнителях, треках и альбомах и управлять ею. В настоящее время код, связанный с каталогом внутри монолита, использует таблицу `Альбомы` для хранения информации о компакт-дисках, доступных для продажи. Эти альбомы в конечном счете попадают в таблицу `Гроссбух`, где мы отслеживаем весь товарооборот. В таблицу `Гроссбух` записывается дата реализации товара вместе с идентификатором, ссылающимся на проданный продукт. Идентификатор в нашем примере называется единицей учета запасов (SKU, stock keeping unit) — это обычная практика в системах розничной торговли.

В конце каждого месяца необходимо составлять отчет с описанием самых продаваемых компакт-дисков. Таблица `Гроссбух` помогает нам понять, копий какого артикула продано больше всего, но информация об этом артикуле находится в таблице `Альбомы`. Отчеты должны быть понятными и удобными для чтения, поэтому вместо того, чтобы говорить: «Мы продали 400 копий

[1] *Hodgson P.* Feature Toggles (aka Feature Flags). martinfowler.com, 9 октября 2017 года. https://oreil.ly/XiU2t.

артикула 123 и заработали 1596 долларов», мы бы добавили больше информации. Например: «Мы продали 400 копий Now That's What I Call Death Polka и заработали 1596 долларов». Этот запрос к базе данных инициируется кодом финансовой части, что требует добавления информации из таблицы `Гроссбух` в таблицу `Альбомы`, как показано на рис. 3.6.

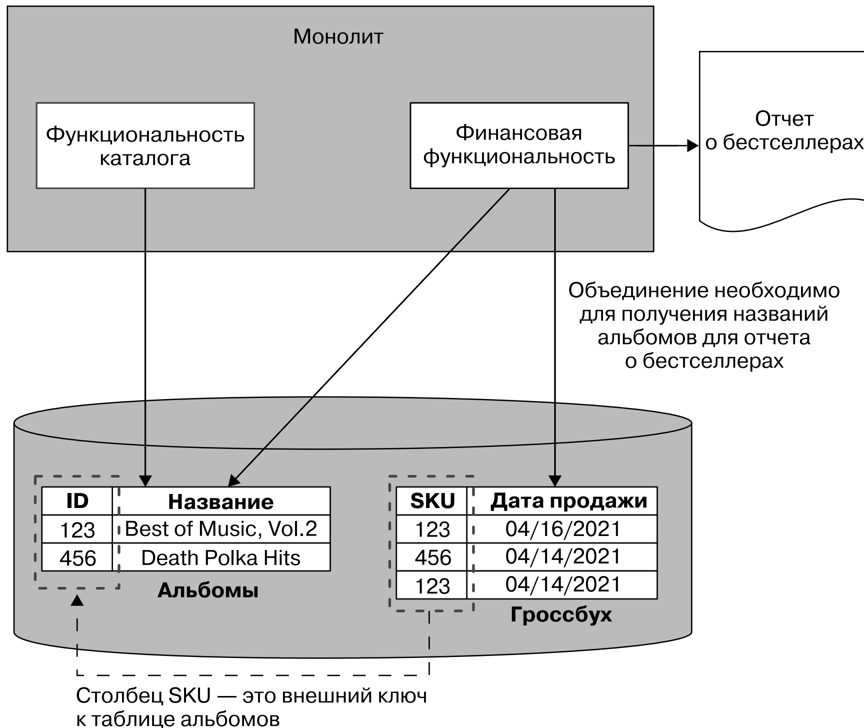


Рис. 3.6. Операция объединения в монолитной базе данных

В нашем новом мире, основанном на микросервисах, сервис `Финансы` отвечает за создание отчета о бестселлерах, но не содержит в себе данных об альбомах. Поэтому ему нужно будет извлечь эти данные из микросервиса `Каталог`, как показано на рис. 3.7. При создании отчета микросервис `Финансы` сначала запрашивает таблицу `Гроссбух`, извлекая список самых продаваемых артикулов за последний месяц. На данный момент единственная информация, которую содержит мокросервис, — это список артикулов и количество проданных копий каждого артикула.

Далее нам нужно вызвать микросервис `Каталог`, запросив информацию об этих артикулах. Такой запрос приведет к тому, что микросервис `Каталог` локально выполнит `SELECT` в своей базе данных.

Логически операция объединения все еще выполняется, но теперь она происходит внутри микросервиса `Финансы`, а не в БД. Объединение перешло с уровня данных на уровень кода приложения. К сожалению, эта операция будет далеко не такой эффективной, как объединение в базе данных. Мы ушли из мира, в котором у нас было единственный оператор `SELECT`, в новый, где есть запрос `SELECT` к таблице `Гроссбух`. За ним следует вызов микросервиса `Каталог`, который, в свою очередь, запускает оператор `SELECT`, адресованный таблице `Альбомы`, как показано на рис. 3.7.

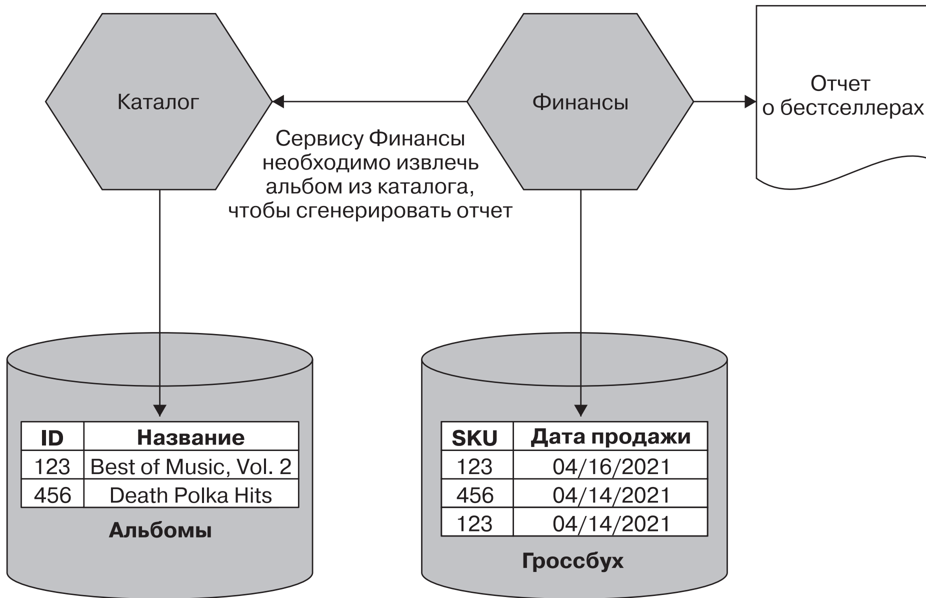


Рис. 3.7. Замена операции объединения с базой данных вызовами сервисов

В этой ситуации я был бы *очень* удивлен, если бы общее время ожидания этой операции не увеличилось. В данном случае это не представляет серьезной проблемы, поскольку отчет генерируется ежемесячно и может быть принудительно кэширован (более подробно об этом — в разделе «Кэширование» главы 13). Но если операция часто повторяется, задержка может стать более серьезной проблемой.

Для того чтобы снизить влияние задержки, нужно разрешить массовый поиск артикулов в микросервисе `Каталог` или, возможно, даже локально кэшировать необходимую информацию об альбоме.

Целостность данных

Базы данных могут быть полезны для обеспечения целостности наших данных. Возвращаясь к рис. 3.6, где таблицы `Альбомы` и `Гроссбух` находились в одной БД, мы могли бы определить (и, вероятно, определили бы) связь внешнего ключа между строками в таблицах `Гроссбух` и `Альбомы`. Это гарантирует, что мы всегда сможем перейти от записи в таблице `Гроссбух` к информации о проданном альбоме, так как, если записи из таблицы `Альбомы` упоминаются в `Гроссбухе`, их нельзя удалить.

Поскольку эти таблицы теперь находятся в разных базах данных, у нас больше нет возможности обеспечить целостность модели данных. Ничто не мешает нам удалить строку в таблице `Альбомы`, что вызовет проблему, когда мы попытаемся определить, какой именно товар был продан.

В какой-то степени вам просто нужно привыкнуть, что больше нет возможности полагаться на свою БД в вопросе обеспечения целостности взаимосвязей между сущностями. Очевидно, что для данных, остающихся внутри одной БД, это не проблема.

Существует ряд обходных путей, хотя словосочетание «модели преодоления» могло бы стать лучшим термином для описания того, как можно справиться с этой проблемой. Мы могли бы использовать мягкое удаление в таблице `Альбомы`, чтобы на самом деле не избавляться от записи, а просто пометить ее как удаленную. Другим вариантом может быть копирование названия альбома в таблицу `Гроссбух` при совершении продажи, но придется решить, как обрабатывать синхронизацию изменений в названии альбома.

Транзакции

Многие из нас привыкли полагаться на гарантии, получаемые при управлении данными в транзакциях. Мы знаем, что БД могут автоматически решать ряд определенных задач, и создаем приложения, полагаясь на эти знания. Однако, как только мы начинаем разделять данные по нескольким БД, утрачивается безопасность транзакций ACID, к которым мы привыкли (ACID мы рассмотрим в главе 6).

Людям, привыкшим к системам, в которых все изменения состояния управляются в рамках границы одной транзакции, может быть сложно перейти к распределенным системам. И часто их реакцией на это становится попытка внедрить распределенные транзакции, чтобы вернуть гарантии, которые ACID-транзакции давали нам с более простыми архитектурами. В разделе «Транзакции базы данных» главы 6 будет подробно описано, что, к сожалению, распределенные транзакции не только сложны в реализации, но и на самом деле не дают нам тех

гарантий, которые мы привыкли ожидать от транзакций с более узким охватом базы данных.

Вскоре, в разделе «Саги» главы 6, вы узнаете, что существуют альтернативные (и предпочтительные) механизмы распределенных транзакций для управления изменениями состояния между несколькими микросервисами, но они сопряжены с новыми сложностями. Как и в случае с целостностью данных, мы должны смириться с тем фактом, что, разбивая наши БД, мы столкнемся с новым перечнем проблем.

Инструментарий

Преобразование баз данных затруднено по многим причинам, одна из которых заключается в ограниченном количестве доступных инструментов, позволяющих легко вносить изменения. Что касается кода, существует инструментарий рефакторинга, встроенный в среды IDE, и есть дополнительное преимущество: изменяемые системы в принципе не имеют состояния. В случае с базой данных у изменяемых сущностей есть состояние, и здесь также не хватает хороших инструментов для рефакторинга.

Существует множество инструментов, которые помогут управлять процессом изменения схемы реляционной базы данных, но большинство из них работают по одному и тому же шаблону. Каждое изменение схемы определяется в дельта-скрипте с контролем версий. Затем эти скрипты выполняются в строгом порядке идемпотентным образом. Миграции Rails работают так же, как и DBDeploy — инструмент, в создании которого я участвовал много лет назад.

В настоящее время для достижения того же результата я советую людям Flyway (https://flywaydb.org) или Liquibase (https://www.liquibase.org), если у них еще нет подобного инструмента.

База данных отчетов

В рамках извлечения микросервисов из монолита мы также разделяем базы данных, поскольку хотим скрыть доступ к внутреннему хранилищу данных. Убирая прямой доступ к базам данных, можно создавать стабильные интерфейсы, позволяющие осуществлять независимое развертывание. К сожалению, это вызывает проблемы, когда доступ к данным осуществляется из более чем одного микросервиса или когда эти данные доступны в БД, а не через что-то вроде REST API.

С помощью базы данных отчетов мы создаем выделенную базу, предназначенную для внешнего доступа, и возлагаем на микросервис ответственность за передачу информации из внутреннего хранилища в эту базу данных отчетов, как показано на рис. 3.8.

База данных отчетов позволяет скрыть внутреннее управление состоянием, сохраняя при этом предварительную отправку данных в БД, что может быть очень полезно. Например, можно разрешить пользователям выполнять специальные SQL-запросы, запускать крупномасштабные операции объединения или использовать существующие цепочки инструментов, у которых должен быть доступ к конечной точке SQL. База данных отчетов будет хорошим решением этой проблемы.

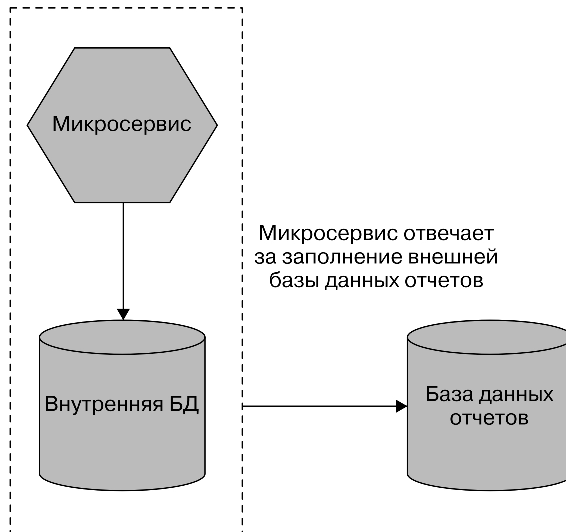


Рис. 3.8. Обзор шаблона базы данных отчетов

Здесь следует выделить два ключевых момента. Во-первых, нам все еще требуется больше практики в скрытии информации. Поэтому в базе данных отчетов следует отображать только минимальный объем данных. Это означает, что находящиеся в БД отчетов данные могут быть только подмножеством данных, хранящихся в микросервисе. Так как здесь не выполняется прямое сопоставление данных, у нас появляется возможность разработать для базы отчетов схему, которая точно соответствует требованиям потребителей: с помощью радикально другой схемы или, возможно, даже совершенно другого типа технологии БД.

Во-вторых, база данных отчетов должна обрабатываться как любая другая конечная точка микросервиса, и задача сопровождающего микросервиса заключается в обеспечении совместимости этой конечной точки с системой, даже если внутренние детали реализации микросервиса будут изменены. Отображение внутреннего состояния в базе данных отчетов относится к обязанностям людей, разрабатывающих сам микросервис.

Резюме

Итак, приступая к работе по переносу функциональности с монолитной архитектуры на микросервисную, необходимо иметь четкое представление своей конечной *цели*. Она определит, как вы будете выполнять свою работу, а также поможет вам понять, в правильном ли направлении вы движетесь.

Переход должен быть постепенным: внесение изменения, развертывание, оценка — затем повтор этих действий. Даже сам процесс отделения одного микросервиса от монолита может быть разбит на ряд небольших шагов.

Если вы хотите более глубоко погрузиться в изучение какой-либо из концепций этой главы, рекомендую ознакомиться с другой моей книгой — «От монолита к микросервисам».

В следующей главе мы перейдем к более техническим вопросам и рассмотрим взаимодействие между микросервисами.

ГЛАВА 4

Стили взаимодействия микросервисов

Для многих наладить правильное взаимодействие между микросервисами представляется проблематичным, так как люди тяготеют к выбранному технологическому подходу, не рассматривая альтернативы. В этой главе я попытаюсь разобрать различные стили взаимодействия, чтобы помочь вам понять плюсы и минусы каждого из них, а также определить, какой подход лучше всего подойдет для решения ваших задач.

Мы рассмотрим механизмы синхронной блокировки и асинхронной неблокирующей связи, а также сравним взаимодействие «запрос — ответ» и событийную архитектуру связи.

К концу главы вы сможете гораздо глубже понять различные доступные варианты и получите базовые знания, помогающие разобраться с проблемами, описанными в следующих главах.

От внутрипроцессного к межпроцессному

Итак, давайте сначала разберемся с легкими вопросами, точнее, с тем, что, с моей точки зрения, *кажется* легким. Вызовы *между* различными процессами по сети (межпроцессные) *сильно* отличаются от вызовов *внутри* одного процесса (внутрипроцессные). На каком-то уровне мы можем игнорировать это различие. Например, легко представить, что один объект вызывает метод другого объекта, а затем просто сопоставить этот процесс с двумя микросервисами, взаимодействующими по сети. Если оставить в стороне тот факт, что микросервисы — это не просто объекты, такое мышление может привести нас к большим неприятностям.

Давайте рассмотрим некоторые из этих различий и то, как они могут изменить ваше представление о взаимодействии между микросервисами.

Производительность

Производительность внутрипроцессного вызова принципиально отличается от межпроцессного. Когда выполняется внутрипроцессный вызов, базовый компилятор и среда выполнения могут произвести целый ряд оптимизаций, чтобы уменьшить влияние вызова на производительность, включая встраивание вызова в процесс выполнения, будто его никогда и не было. При межпроцессных вызовах такая оптимизация невозможна. Пакеты должны быть отправлены. Накладные расходы на такой вызов ожидаемо будут выше, чем при внутрипроцессном вызове. Первое очень легко измерить — обыкновенная передача одного пакета в центре обработки данных длится миллисекунды, в то время как накладные расходы, связанные с вызовом метода, не стоят вашего внимания.

Это часто приводит к желанию переосмыслить API. API, который разумно применять внутри процесса, может оказаться нецелесообразным в межпроцессных ситуациях. Я могу спокойно сделать тысячу вызовов через границу API внутри процесса. Захочу ли я сделать то же самое между двумя микросервисами? Возможно, и нет.

Когда параметр передается в метод, структура данных обычно не перемещается — более вероятно, что в запросе будет передан указатель на ячейку памяти. Передача объекта или структуры данных другому методу не требует выделения дополнительной памяти для копирования данных.

С другой стороны, данные фактически должны быть сериализованы в некую форму, которую можно передавать по сети при выполнении вызовов между микросервисами. Затем данные необходимо отправить и десериализовать принимающей стороне. Поэтому нам, возможно, потребуется более внимательно относиться к размеру полезных нагрузок, отправляемых между процессами. Когда вы в последний раз думали про размер структуры данных, передаваемой внутри процесса? Скорее всего, никогда. Однако теперь стоит задуматься. Это может привести к уменьшению объема отправляемых или принимаемых данных (возможно, это неплохо, если речь о скрытии информации), выбору более эффективных механизмов сериализации или даже выгрузке данных в файловую систему и передаче ссылки на местоположение файла.

Эти различия могут не сразу вызвать проблемы, но, безусловно, стоит о них знать. Я много раз сталкивался с попытками утаить от разработчика факт наличия сетевого вызова. Желание создавать абстракции, чтобы скрыть детали,

позволяет выполнять больше задач и делать это эффективнее. Но иногда в этом стремлении мы переусердствуем настолько, что абстракции скрывают слишком много. Разработчик должен осознавать, что выполняет операцию, приводящую к сетевому вызову. В противном случае не стоит удивляться, если вы столкнетесь с уменьшением производительности, вызванным странными взаимодействиями между сервисами, которые не были видны разработчику.

Изменение интерфейсов

Если рассматривать преобразования в интерфейсе внутри процесса, развертывание изменений представляет собой достаточно простой набор действий. Код, реализующий интерфейс, и код, вызывающий интерфейс, упаковываются в один и тот же процесс. На самом деле, если сигнатура метода изменяется при помощи IDE с возможностью рефакторинга, часто сама среда IDE автоматически выполняет рефакторинг вызовов этого изменяемого метода. Развертывание такого изменения может быть выполнено атомарным способом — обе стороны интерфейса упаковываются в один процесс.

Сервис, предоставляющий интерфейс, и сервис-потребитель являются отдельно развертываемыми микросервисами, когда мы говорим о коммуникации между микросервисами. При внесении обратно несовместимых изменений в микросервис интерфейса необходимо либо выполнить одновременное развертывание с потребителями, убедившись, что они готовы к использованию нового интерфейса, либо найти способ поэтапного развертывания нового контракта микросервиса. Мы рассмотрим эту концепцию более подробно позже в этой главе.

Обработка ошибок

При вызове метода внутри процесса природа ошибок, как правило, довольно проста: сбой либо ожидаем и с ним легко справиться, либо он распространяется вверх по стеку вызовов. Ошибки в целом детерминированы.

В распределенной системе характер ошибок может быть разным. Система уязвима для множества сбоев, находящихся вне вашего контроля. Например, тайм-аут сети. Нижестоящие микросервисы могут быть временно недоступны. Сети отключаются, контейнеры отключаются из-за потребления слишком большого объема памяти, а в экстремальных ситуациях части центра обработки данных могут загореться[1].

[1] Реальная история.

В книге «Распределенные системы»[1] Эндрю Таненбаум и Мартен Стин описывают пять типов отказов, встречающихся при рассмотрении межпроцессного взаимодействия. Вот упрощенная версия.

Аварийный отказ

Все было хорошо, пока сервер не вышел из строя. Перезагрузка!

Пропуск при отказе

Вы что-то отправили, но не получили отклика. Сюда же относятся ситуации, в которых ожидается, что нижестоящий микросервис будет отправлять сообщения (возможно, включая события), но он просто останавливается.

Сбой синхронизации

Что-то случилось с опозданием (нужно было к определенному моменту) или, наоборот, преждевременно!

Ошибка ответа

Вы получили ответ, но он кажется неправильным. Например, вы запросили сводку заказа, но в ответе отсутствуют необходимые данные.

Произвольный отказ

Иначе известен как «византийская ошибка». Это когда что-то пошло не так, но участники не могут прийти к соглашению, произошел ли сбой (или почему). Это звучит как «ничего страшного, время такое».

Многие из этих ошибок часто носят нерегулярный характер — это кратковременные проблемы, которые могут исчезнуть. Рассмотрим ситуацию, в которой мы отправляем запрос в микросервис, но не получаем ответа (тип ошибки «пропуск при отказе»). В первую очередь это может означать, что нижестоящий микросервис не получал запрос, поэтому нам нужно отправить его снова. Существуют проблемы, требующие вмешательства человека-оператора. В результате важно иметь богатый набор семантических выражений для возврата ошибок таким образом, чтобы клиенты могли предпринять соответствующие действия.

HTTP — пример протокола, показывающего важность этого процесса. Каждый HTTP-ответ содержит код, при этом коды серий 400 и 500 зарезервированы для ошибок. Серия 400 представляет собой ошибки запроса — по сути, сервис ниже по потоку выполнения сообщает клиенту, что с исходным запросом что-то не так. Следовательно, нет смысла повторять попытку, например, при ответе `404 Not Found`. Коды ответов серии 500 относятся к последующим проблемам. Их подмножество указывает клиенту, что сбой может быть кратковременным. Например, ошибка `503 Service Unavailable` говорит нам, что нижестоящий сервер пока что не в состоянии обработать запрос, но, немного подождав, вышестоящий клиент может повторить запрос. С другой

[1] *Стин ван М., Таненбаум Э. С.* Распределенные системы. 3-е изд.

стороны, если клиент получает ответ `501 Not Implemented`, очередная попытка вряд ли что-то изменит.

Неважно, выбираете вы протокол на основе HTTP для связи между микросервисами или нет, — если у вас есть богатый набор семантики, учитывающий природу ошибки, вы облегчите клиентам выполнение действий, направленных на устранение последствий ошибок. Это, в свою очередь, должно помочь вам создавать более надежные системы.

Технология межпроцессного взаимодействия: так много вариантов выбора

> В мире, где слишком много вариантов выбора и слишком мало времени, разумным будет просто игнорировать некоторые вещи.
>
> *Сет Годин*

Спектр доступных технологий для межпроцессного взаимодействия огромен. В результате мы часто оказываемся перегружены выбором. Я нередко замечаю, что люди тяготеют к незнакомым технологиям или, возможно, просто к новейшим, о которых они узнали на какой-нибудь выставке. Проблема заключается в том, что, покупая конкретную технологию, вы приобретаете набор не только идей, но и ограничений, возникающих в процессе работы. Эти ограничения могут оказаться неподходящими для вас, и задумка, лежащая в основе технологии, может на самом деле не соответствовать проблеме, которую вы стараетесь решить.

Если вы пытаетесь создать веб-сайт, то технологии одностраничных приложений, такие как Angular или React, вам не подойдут. Аналогично попытка использовать Kafka для реализации стиля взаимодействия «запрос — ответ» — не очень хорошая идея, поскольку это приложение было разработано для большего количества событийных взаимодействий (темы, к которым мы перейдем чуть позже). И все же я снова и снова сталкиваюсь с тем, что технологии используются не по назначению. Люди выбирают «крутую» новенькую разработку (например, микросервисы!), не задумываясь о том, действительно ли это необходимо.

Таким образом, когда дело доходит до выбора технологий для организации связи между микросервисами, я считаю, что сначала важно определиться с желаемым стилем взаимодействия и только потом искать правильную технологию для реализации. Имея это в виду, давайте взглянем на модель, которую я использую уже несколько лет. Она помогает различать всевозможные подходы к взаимодействию между микросервисами, что, в свою очередь, может помочь отфильтровать варианты доступных технологий.

Стили взаимодействия микросервисов

На рис. 4.1 показана схема модели, которую я использую во время размышлений о различных стилях взаимодействия. Эта модель не претендует на звание эталонной (я не пытаюсь представить здесь общую теорию межпроцессного взаимодействия), но она обеспечивает хороший высокоуровневый обзор для рассмотрения различных стилей взаимодействия, наиболее широко используемых в микросервисных архитектурах.

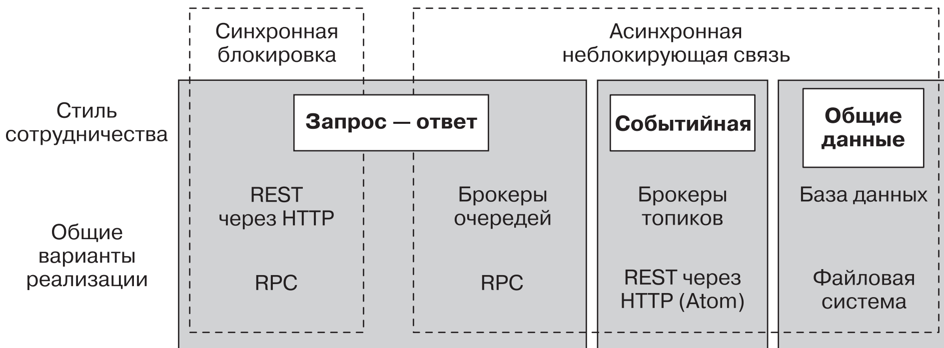


Рис. 4.1. Различные стили межмикросервисной коммуникации наряду с примерами технологий внедрения

Вскоре мы рассмотрим различные элементы этой модели более подробно, но сначала я хотел бы кратко описать их.

Синхронная блокировка

Микросервис выполняет вызов другого микросервиса и блокирует операцию в ожидании ответа.

Асинхронная неблокирующая связь

Микросервис, отправляющий вызов, способен продолжать работу независимо от того, принят вызов или нет.

Запрос — ответ

Микросервис отправляет другому микросервису запрос на какое-то действие и ожидает получить ответ, информирующий его о результате.

Событийный стиль

Микросервисы генерируют события, которые другие микросервисы потребляют, и реагируют на них соответствующим образом. Микросервис, выпускающий события, не знает их конечного потребителя и имеется ли таковой вообще.

Общие данные

Не часто рассматриваются как стиль коммуникации. При таком стиле микросервисы взаимодействуют через какой-то общий источник данных.

Командам, использующим данную модель, для выбора правильного подхода приходится тратить много времени на понимание контекста, в котором они работают. При выборе технологии определенную роль будут играть требования в отношении надежной связи, приемлемой задержки. Но в целом я склонен начинать с принятия решения о том, какой стиль взаимодействия более подходит для данной ситуации: «запрос — ответ» или событийный стиль. Если рассматривать стиль «запрос — ответ», то будут доступны как синхронные, так и асинхронные реализации, поэтому возникает необходимость сделать второй выбор. Однако при выборе событийного стиля взаимодействия способы реализации будут ограничены неблокирующим асинхронным вариантом.

При выборе правильной технологии в игру вступает множество других соображений, не касающихся стиля взаимодействия, — например, необходимость связи с меньшей задержкой, вопросы, связанные с безопасностью, или возможность масштабирования. Маловероятно, что вы сможете сделать обоснованный выбор технологии без учета требований (и ограничений) вашей конкретной предметной области. В главе 5 мы обсудим некоторые из этих вопросов.

Смешивание и сочетание

Важно отметить, что микросервисная архитектура в целом может поддерживать различные стили взаимодействия, и это считается нормой. Одни виды взаимодействия имеет смысл организовать просто в стиле «запрос — ответ», в то время как другие — в событийном стиле. На самом деле для одного микросервиса обычно реализуется более одной формы взаимодействия. Рассмотрим микросервис `Заказ`, предоставляющий API «запрос — ответ», который позволяет размещать или изменять заказы, а затем инициирует события при внесении этих изменений.

С учетом сказанного разберем эти различные стили взаимодействия более подробно.

Шаблон: синхронная блокировка

При синхронном блокирующем вызове микросервис отправляет какой-либо вызов нижестоящему процессу (вероятно, другому микросервису) и блокируется до завершения вызова и, возможно, до получения ответа. На рис. 4.2 `Обработчик заказов` отправляет вызов микросервису `Лояльность`, чтобы сообщить ему о необходимости добавить несколько баллов в учетную запись клиента.

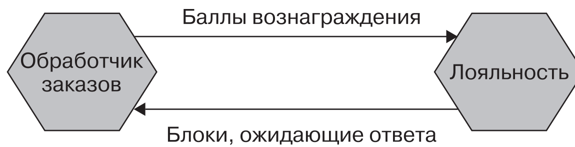


Рис. 4.2. Обработчик заказов отправляет синхронный вызов микросервису Лояльность, блокируется и ожидает ответа

Как правило, синхронный блокирующий вызов — это вызов, ожидающий ответа от нижестоящего процесса. Такой подход связан с необходимостью использовать результат вызова для какой-то дальнейшей операции или, если отклик не получен, предпринять повторную попытку. В результате практически каждый встречающийся мне синхронный блокирующий вызов на деле оказывается вызовом типа «запрос — ответ».

Преимущества

В блокирующем синхронном вызове есть что-то простое и знакомое. Многие из нас научились программировать в фундаментально синхронном стиле, читая фрагмент кода как сценарий, где строки выполняются по очереди. Большинство ситуаций, в которых используются межпроцессные вызовы, вероятно, исполнялись в синхронном, блокирующем стиле — например, выполнение SQL-запроса к базе данных или HTTP-запроса к нижестоящему API.

При переходе от монолитной архитектуры с одним процессом имеет смысл придерживаться знакомых идей, даже когда вокруг столько нового.

Недостатки

Основная проблема при синхронных вызовах — возникающая временная связанность (эту тему мы кратко рассмотрели в главе 2). Когда `Обработчик заказов` вызывал сервис `Лояльность` в предыдущем примере, микросервис `Лояльность` оставался доступным для вызова. Если микросервис `Лояльность` недоступен, то вызов завершается неудачей и `Обработчику заказов` необходимо выполнить компенсирующее действие: немедленную повторную попытку, буферизацию вызова для повторной попытки позже или, возможно, полный отказ.

Это двухсторонняя связанность. При таком стиле интеграции ответ обычно отправляется по тому же входящему сетевому подключению к вышестоящему микросервису. Таким образом, если микросервис `Лояльность` захочет отправить ответ обратно в `Обработчик заказов`, когда вышестоящий экземпляр отсутствует, ответ будет потерян. Временная связанность здесь возникает не только между двумя микросервисами — она существует между двумя конкретными экземплярами этих микросервисов.

Отправитель вызова блокируется и ожидает ответа нижестоящего микросервиса, из чего следует, что если нижестоящий микросервис отвечает медленно или если существует проблема задержки сети, то отправитель вызова будет заблокирован в течение длительного периода времени в ожидании ответа. Если микросервис `Лояльность` находится под значительной нагрузкой и долго отвечает на запросы, это, в свою очередь, приводит к тому, что и `Обработчик заказов` замедлится.

Таким образом, использование синхронных вызовов может сделать систему более уязвимой для каскадных проблем, вызванных сбоями ниже по потоку, чем применение асинхронных вызовов.

Где использовать

В простых микросервисных архитектурах серьезных проблем с использованием синхронных блокирующих вызовов не возникает. Если вы будете разбираться в таких вызовах, это станет преимуществом при работе с распределенными системами.

Для меня использование этих типов вызовов становится спорным, когда появляется больше цепочек вызовов. На рис. 4.3 приведен пример потока из MusicCorp, где платеж проверяется на предмет потенциально мошеннических действий. Сервис `Обработчик заказов` вызывает микросервис `Оплата` для получения платежа. Сервис `Оплата`, в свою очередь, с помощью микросервиса `Обнаружение мошенничества` должен проверить, следует давать разрешение или нет. Микросервису `Обнаружение мошенничества`, в свою очередь, необходимо получить информацию от микросервиса `Покупатель`.

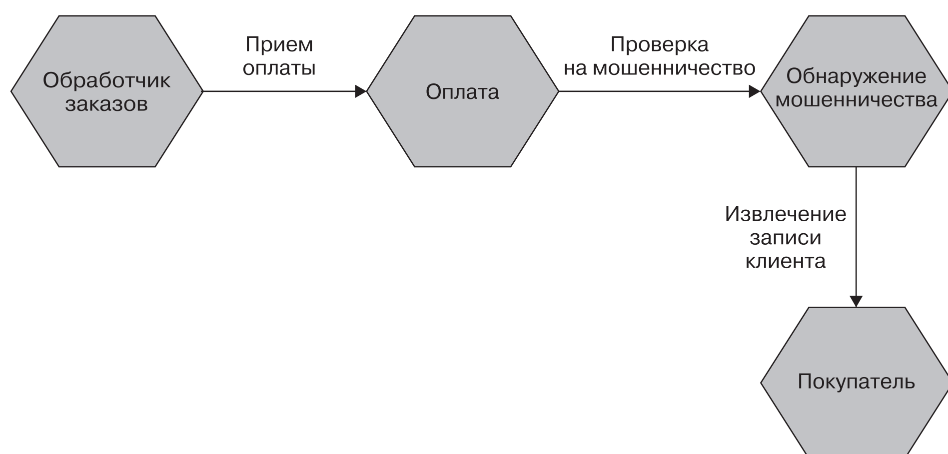


Рис. 4.3. Проверка на потенциально мошенническое поведение в рамках процесса обработки заказов

Если все эти вызовы будут синхронными и блокирующими, вы столкнетесь с рядом сложностей. Проблема в любом из четырех задействованных микросервисов или в сетевых вызовах между ними может привести к сбою всей операции. И это помимо *конкуренции за ресурсы*, которую могут вызвать такие длинные цепочки. За кулисами `Обработчик заказов`, вероятно, поддерживает открытое сетевое соединение, ожидающее ответа от сервиса `Оплата`, у которого, в свою очередь, имеется открытое сетевое соединение, ожидающее ответа от сервиса `Обнаружение мошенничества`, и т. д. Наличие большого количества подключений, которые необходимо держать открытыми, может повлиять на работу системы: либо доступные подключения закончатся, либо произойдет перегрузка сети.

Чтобы нивелировать влияние этих проблем, стоит в первую очередь пересмотреть взаимодействие между микросервисами. Например, исключить использование функции `Обнаружение мошенничества` из основного потока покупок, как показано на рис. 4.4, и запустить ее в фоновом режиме. Если обнаруживается проблема с конкретным клиентом, его записи соответствующим образом обновляются, что можно было бы проверить ранее в процессе оплаты. Фактически часть работы выполняется параллельно. При сокращении длины цепочки вызовов общая задержка операции уменьшится и мы уберем один из микросервисов (`Обнаружение мошенничества`) из критического пути для потока покупок. В итоге у нас станет на одну критическую операцию меньше.

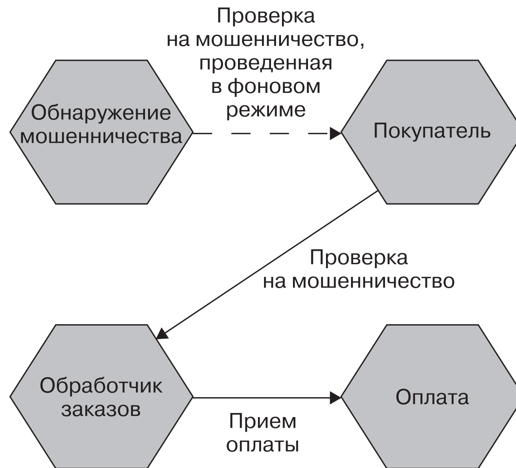


Рис. 4.4. Перенос функции Обнаружение мошенничества в фоновый процесс может снизить влияние проблем, связанных с длиной цепочки вызовов

Конечно, мы могли бы также заменить использование блокирующих вызовов некоторым стилем неблокирующего взаимодействия, не меняя рабочий процесс. Этот подход мы рассмотрим далее.

Шаблон: асинхронная неблокирующая связь

При асинхронной связи процесс отправки вызова по сети не блокирует микросервис, отправляющий вызов. Тот способен продолжить любую другую обработку, не дожидаясь ответа. Неблокирующая асинхронная связь существует во многих формах, но мы рассмотрим три наиболее распространенных варианта.

Связь через общие данные

Вышестоящий микросервис изменяет кое-какие общие данные, которые позже использует один или несколько сервисов.

Запрос — ответ

Микросервис отправляет другому микросервису запрос на какое-то действие. Когда запрошенная операция завершается успешно (или нет), вышестоящий микросервис получает ответ. В частности, *любой* экземпляр вышестоящего микросервиса должен быть в состоянии обработать ответ.

Событийное взаимодействие

Микросервис транслирует событие, которое можно рассматривать как фактическое утверждение о чем-то, что произошло. Другие микросервисы могут прослушивать интересующие их события и реагировать соответствующим образом.

Преимущества

При неблокирующей асинхронной связи микросервис, выполняющий первоначальный вызов, и микросервис (или микросервисы), принимающий вызов, временно утрачивают связанность. Последний не обязательно должен быть доступен одновременно с вызовом. Это означает, что мы можем избежать проблем временного отсутствия связанности, которые обсуждались в главе 2 (см. «Примечание о временной связанности»).

Данный стиль связи также полезен, если для обработки запроса требуется много времени. Вернемся к нашему примеру с MusicCorp и, в частности, к процессу отправки посылки. На рис. 4.5 `Обработчик заказов` принял оплату и отправил вызов микросервису `Склад`. Процесс поиска компакт-дисков на стеллажах, упаковки и доставки может занять от пары часов до нескольких дней. Следовательно, имеет смысл `Обработчику заказов` выполнить неблокирующий асинхронный вызов сервиса `Склад`, чтобы затем получить от него информацию о продвижении заказа. Это форма асинхронной связи «запрос — ответ».

При попытке сделать что-то подобное с помощью синхронной блокировки вызовов пришлось бы перестроить взаимодействие между сервисами `Обработчик заказов` и `Склад`. Для `Обработчика заказов` было бы невозможно открыть

соединение, отправить запрос, заблокировать любые дальнейшие операции при вызове потока и ждать ответа в течение нескольких часов или дней.

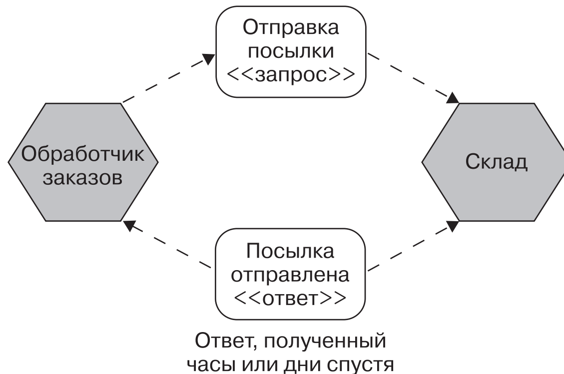


Рис. 4.5. Обработчик заказов асинхронным способом запускает процесс упаковки и отправки заказа

Недостатки

Основными недостатками неблокирующей асинхронной связи являются уровень сложности и диапазон выбора. Как уже отмечалось, существуют различные стили асинхронной связи на выбор — какой из них подходит именно вам? Когда мы начинаем копаться в реализациях этих различных стилей взаимодействия, вырисовывается огромный список технологий, на которые можно было бы обратить внимание.

Если асинхронная связь не соответствует вашим ментальным моделям вычислений, переход на асинхронный стиль связи поначалу будет непростой задачей. Дальнейшее подробное рассмотрение данного стиля покажет нам, что существует множество различных интересных способов навлечь на себя *внушительное* количество неприятностей.

ASYNC/AWAIT И КОГДА АСИНХРОННАЯ СВЯЗЬ ВСЕ ЕЩЕ БЛОКИРУЕТСЯ

Как и во многих областях информатики, здесь можно использовать один и тот же термин в разных контекстах, чтобы получились очень разные значения. Стиль программирования, который, по-видимому, особенно популярен, — это использование конструкций типа async/await для работы с потенциально асинхронным источником данных, но в блокирующем, синхронном стиле.

В примере 4.1 показана очень простая иллюстрация такой реализации на JavaScript. Курсы обмена валюты часто колеблются в течение дня, и мы получаем их через брокер сообщений. Мы определяем промис (promise). В целом промис — это то, что в какой-то момент в будущем разрешится. В нашем случае `eurToGbp` станет следующим обменным курсом евро к фунту стерлингов.

Пример 4.1. Пример работы блокирующим, синхронным способом с потенциально асинхронным вызовом

```
async function f() {

  let eurToGbp = new Promise((resolve, reject) => {
    // код для получения последних данных по курсу валют EUR/GBP
    ...
  });

  var latestRate = await eurToGbp; ❶
  process(latestRate); ❷
}
```

❶ Ожидает, пока не будут получены последние данные по курсу валют Евро/Фунт стерлингов.

❷ Не будет работать, пока промис не выполнен.

При ссылке на `eurToGbp` с помощью `await` мы блокируем связь до тех пор, пока состояние `latestRate` не будет успешно выполнено, а функция `process` не будет выполняться, пока мы не разрешим состояние `eurToGbp`[1].

Несмотря на то что обменные курсы принимаются асинхронно, использование `await` в этом контексте означает, что *блокирование* происходит до тех пор, пока не будет разрешено состояние `latestRate`. Таким образом, даже если базовую технологию для получения курса можно считать асинхронной по своей природе (например, ожидание курса), то с точки зрения нашего кода это по своей сути синхронное, блокирующее взаимодействие.

Где использовать

В конечном счете необходимо учитывать, какой *тип* асинхронной связи вы хотите выбрать, поскольку у каждого из них есть свои компромиссы. Однако имеются конкретные варианты использования, при которых необходимо обратиться к той или иной форме асинхронной связи. Очевидным кандидатом представляются длительные процессы, как показано на рис. 4.5. Кроме того, хорошими претендентами могут быть ситуации с длинными цепочками вызовов, которые нелегко реструктурировать. Мы углубимся в это, когда рассмотрим три наиболее распространенные формы асинхронной связи: вызовы «запрос — ответ», событийную связь и взаимодействие через общие данные.

[1] Обратите внимание, что пример очень упрощен — я полностью опустил код обработки ошибок. Если вы хотите узнать больше об async/await, в частности, в JavaScript, ознакомьтесь с Modern JavaScript Tutorial (https://java script.info). Это отличный вариант для начала.

Шаблон: связь через общие данные

Стиль взаимодействия, охватывающий множество реализаций, — это связь через общие данные. Данный шаблон используется, когда один микросервис помещает данные в определенное место, а другой (или, возможно, несколько) позже использует их. Представьте, что один микросервис помещает файл в определенное место, а в какой-то момент позже другой микросервис берет этот файл и что-то с ним делает. Такой стиль интеграции принципиально асинхронен по своей природе.

Пример рассматриваемого стиля показан на рис. 4.6. Здесь сервис `Импортер нового продукта` создает файл, позже доступный для микросервисов `Запасы` и `Каталог`, расположенных ниже по потоку.

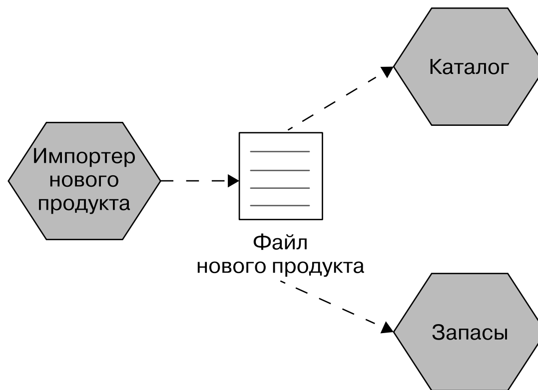


Рис. 4.6. Один микросервис записывает файл, используемый другими микросервисами

Этот шаблон является наиболее распространенным общим паттерном межпроцессного взаимодействия. И все же мы иногда вообще не рассматриваем его как шаблон коммуникации, так как связь между процессами часто настолько косвенна, что ее трудно заметить.

Реализация

Чтобы реализовать такой шаблон, вам нужно постоянное хранилище данных. Файловой системы во многих случаях будет достаточно. Я создал много систем, которые просто периодически сканируют файловую систему, отмечают наличие нового файла и реагируют на него соответствующим образом. Конечно, можно было бы использовать какое-то надежное распределенное хранилище памяти. Стоит отметить, что любому нижестоящему микросервису, которому предстоит

работать с этими данными, потребуется собственный механизм для определения доступности новых данных — поллинг (polling) часто применяется в качестве решения этой проблемы.

Два распространенных примера рассматриваемого шаблона представлены озером данных и хранилищем данных. В обоих случаях эти решения обычно предназначены для обработки больших объемов данных, но, возможно, они существуют на противоположных концах спектра в отношении связанности. При использовании озера данных источники загружают необработанные данные в любом удобном для них формате, а расположенные ниже по потоку потребители этих данных будут знать, как обрабатывать информацию. Хранилище данных представляет собой структурированную систему хранения данных. Микросервисы, передающие данные в хранилище, должны знать его структуру. Если она изменяется обратно несовместимым образом, то эти поставщики данных необходимо обновить.

Как для хранилища данных, так и для озера данных предполагается, что поток информации идет в одном направлении. Один микросервис публикует данные в общем хранилище данных, а нижестоящие потребители считывают их и выполняют соответствующие действия. Проблемой станет реализация совместно используемой БД, в которой множество микросервисов будут считывать и записывать данные в одно и то же хранилище. Пример такой организации обсуждался в главе 2 при изучении общей связанности. На рис. 4.7 показаны микросервисы `Обработчик заказов` и `Склад`, обновляющие одну и ту же запись.

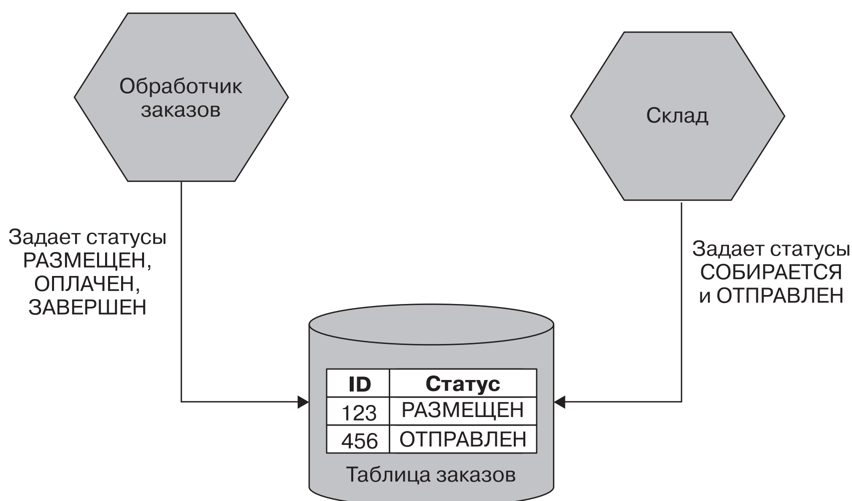


Рис. 4.7. Пример общей связанности, в которой сервисы Обработчик заказов и Склад обновляют одну и ту же запись заказа

Преимущества

Рассматриваемый шаблон можно без проблем реализовать, используя общепринятую технологию. Если у вас есть возможность выполнять чтение/запись файла или базы данных, вы можете использовать этот шаблон. Применение распространенных и хорошо понятных технологий также обеспечивает взаимодействие между различными типами систем, включая старые приложения для мейнфреймов или настраиваемые готовые программные продукты (COTS). Объемы данных также не вызывают здесь особого беспокойства: если вы отправляете много данных за один большой подход — этот шаблон вполне сгодится.

Недостатки

Нижестоящие потребляющие микросервисы обычно узнают о наличии новых данных для обработки с помощью какого-либо механизма поллинга или периодически запускаемого процесса. Это означает, что такой механизм вряд ли будет полезен в ситуациях, требующих минимального отклика. Конечно, вы можете комбинировать данный шаблон с каким-либо другим видом вызова, информирующим нижестоящий микросервис о доступности новых данных. Например, можно записать файл в общую файловую систему, а затем отправить вызов заинтересованному микросервису, сообщив ему о новых данных. Это сократит разрыв между публикацией и обработкой данных. Однако если вы заинтересованы в отправке больших объемов данных и их обработке в режиме реального времени, то лучше использовать какую-либо потоковую технологию, такую как Kafka.

Еще один большой недостаток заключается в том, что общее хранилище данных становится потенциальным источником связанности. Если это хранилище каким-либо образом изменит структуру, это может нарушить связь между микросервисами.

Надежность связи также зависит от отказоустойчивости базового хранилища данных. Строго говоря, это не является недостатком, но об этом следует помнить. При перемещении файла в файловую систему необходимо убедиться, что сама файловая система не выйдет из строя необычным образом.

Где использовать

Где эта модель действительно хороша, так это в обеспечении взаимодействия между процессами, имеющими ограничения на доступные к использованию технологии. Наличие существующей системы, взаимодействующей с интерфейсом GRPC вашего микросервиса или подписывающейся на его топик Kafka, вполне может быть более удобным вариантом с точки зрения микросервиса, но не с точ-

ки зрения потребителя. У старых систем возможны ограничения в отношении поддерживаемых технологий, а также могут оказаться высокие затраты на изменение. С другой стороны, даже старые системы мейнфреймов должны иметь возможность считывать данные из файла. Конечно, все это зависит от использования широко поддерживаемой технологии хранения данных — я также мог бы реализовать этот шаблон, используя что-то вроде кэша Redis. Но может ли ваша старая система мейнфреймов взаимодействовать с Redis?

Еще одним важным преимуществом этой модели стало совместное использование больших объемов данных. Если требуется отправить очень объемный файл в файловую систему или загрузить несколько миллионов строк в базу данных, то использование этого шаблона станет разумным выходом из ситуации.

Шаблон: связь «запрос — ответ»

При использовании модели «запрос — ответ» микросервис отправляет запрос на какое-либо действие нижестоящему сервису и ожидает получить ответ с результатом запроса. Это взаимодействие можно осуществить с помощью синхронного блокирующего вызова или асинхронным неблокирующим методом. Простой пример подобной кооперации показан на рис. 4.8. Здесь микросервис `Чарт`, собирающий самые продаваемые компакт-диски с музыкой разных жанров, отправляет запрос в сервис `Запасы` для получения текущих уровней запасов для некоторых компакт-дисков.

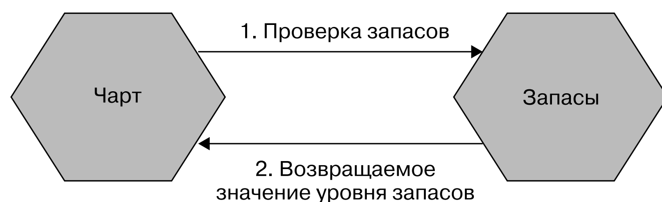


Рис. 4.8. Микросервис Чарт отправляет запрос к сервису Запасы

Извлечение данных из других микросервисов, подобных этому, — распространенный вариант использования для вызова «запрос — ответ». На рис. 4.9 микросервис `Склад` получает запрос на резервирование запасов от сервиса `Обработчик заказов`. Сервису `Обработчик заказов` необходима информация об успешном резервировании товара, прежде чем он сможет продолжить прием оплаты. Если товар на складе нельзя зарезервировать, например, этот товар больше не доступен, тогда платеж может быть отменен. Использование вызовов типа «запрос — ответ» в ситуациях, когда вызовы должны выполняться в определенном порядке, — обычное явление.

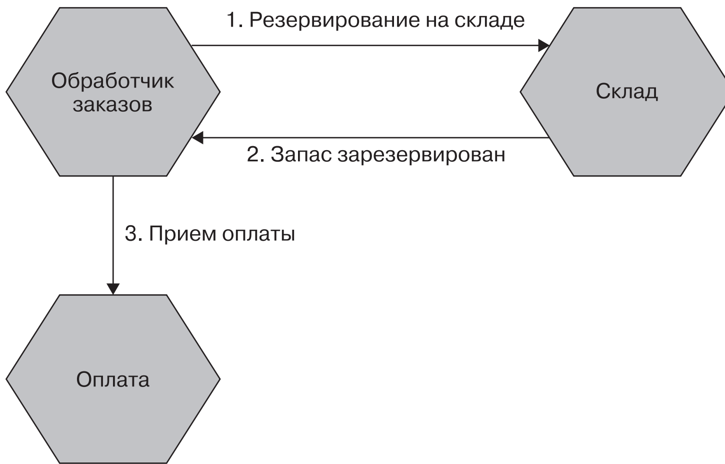


Рис. 4.9. Обработчик заказов должен убедиться, что запасы можно зарезервировать, до того как будет произведена оплата

КОМАНДЫ ИЛИ ЗАПРОСЫ

Иногда люди говорят об отправке команд, а не запросов, особенно в контексте асинхронной связи «запрос — ответ». Смысл термина «команда», возможно, тот же, что и у запроса, а именно: вышестоящий микросервис просит нижестоящий что-то сделать.

Однако лично я предпочитаю термин «запрос». Команда подразумевает директиву, которой необходимо подчиняться, и это может привести к ситуации, когда людям будет казаться, что команда обязательна к выполнению. Запрос подразумевает что-то, что может быть отклонено. Это правильно, потому что микросервис рассматривает каждый запрос по существу и, основываясь на собственной внутренней логике, решает, следует ли выполнять его. Если отправленный запрос нарушает внутреннюю логику, микросервис должен отклонить его. Хотя разница небольшая, я не уверен, что термин «команда» передает то же значение.

Я буду придерживаться термина «запрос», но, какой бы термин вы ни решили использовать, помните, что микросервис может отклонить запрос/команду, если это необходимо.

Реализация: синхронная или асинхронная

Подобные вызовы «запрос — ответ» можно реализовать либо в блокирующем синхронном, либо в неблокирующем асинхронном стиле. При синхронном вызове, как правило, открывается сетевое соединение с микросервисом ниже по потоку, а запрос отправляется по этому соединению. Соединение остается открытым, пока вышестоящий микросервис ожидает ответа нижестоящего. В этом случае сервису, отправляющему ответ, на самом деле не нужно ничего

знать о сервисе, отправившем запрос, — он просто отправляет данные обратно по входящему соединению. Если это соединение обрывается, например, если какой-либо из экземпляров микросервиса удален, тогда у нас может возникнуть проблема.

С асинхронным вызовом в стиле «запрос — ответ» все не так просто. Вернемся к процессу, связанному с резервированием складских запасов. На рис. 4.10 запрос на резервирование отправляется в виде сообщения через своего рода брокер сообщений (мы рассмотрим брокеры сообщений позже в этой главе). Вместо того чтобы сообщение отправлялось непосредственно в микросервис `Запасы` из `Обработчика заказов`, оно помещается в очередь. Сервис `Запасы` по возможности считывает сообщения из этой очереди и выполняет связанную с ними работу по резервированию запасов. Микросервису `Запасы` необходимо знать, куда направить ответ. В нашем примере он отправляет его обратно по другой очереди, используемой `Обработчиком заказов`.

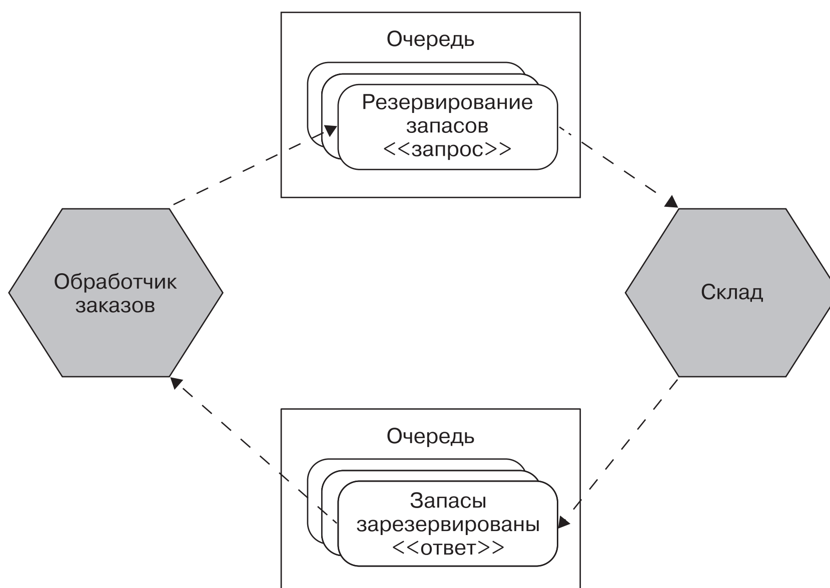


Рис. 4.10. Использование очередей для отправки запросов на резервирование запасов

Таким образом, при неблокирующем асинхронном взаимодействии микросервис, получающий запрос, должен либо неявно знать, куда направить ответ, либо получить указание, куда его послать. При использовании очереди появляется дополнительное преимущество, заключающееся в том, что несколько запросов могут быть буферизованы в очереди, ожидающей обработки. Это поможет

в ситуациях, когда запросы невозможно обработать достаточно быстро. Микросервис может использовать следующий запрос по готовности, вместо того чтобы перегружаться слишком большим количеством вызовов. Конечно, многое зависит от очереди, принимающей эти запросы.

Когда микросервис получает ответ таким образом, ему требуется связать его с исходным запросом. Это непросто, так как может пройти много времени, и в зависимости от характера используемого протокола ответ рискует не дойти до отправившего его экземпляра микросервиса. В нашем примере резервирования запасов в рамках размещения заказа необходимо знать, как связать ответ «запас зарезервирован» с данным заказом для его дальнейшей обработки. Проще всего было бы сохранить любое состояние, связанное с исходным запросом, в базе данных, чтобы при поступлении ответа принимающий экземпляр мог загрузить какое угодно связанное состояние и действовать соответствующим образом.

Последнее замечание: все типы взаимодействия «запрос — ответ», вероятно, потребуют некоторой формы обработки тайм-аута, чтобы избежать проблем, когда система будет заблокирована в ожидании чего-то, что может никогда не произойти. Способ реализации этой функции тайм-аута будет варьироваться в зависимости от применяемой технологии реализации. Мы рассмотрим тайм-ауты более подробно в главе 12.

ПАРАЛЛЕЛЬНЫЕ ИЛИ ПОСЛЕДОВАТЕЛЬНЫЕ ВЫЗОВЫ

При работе со связями типа «запрос — ответ» вы часто будете сталкиваться с ситуацией, в которой вам потребуется выполнить несколько вызовов, прежде чем появится возможность продолжить выполнение программы.

Рассмотрим случай, в котором MusicCorp необходимо проверить цену на определенный товар у трех разных поставщиков с помощью API-вызовов. Требуется получить информацию о ценах от всех дилеров, прежде чем решить, кому из них мы хотим направить заказ на пополнение запасов. Мы могли бы принять решение сделать три вызова последовательно, ожидая завершения каждого из них, прежде чем приступить к следующему. В такой ситуации мы бы ждали время, равное сумме ожидания выполнения каждого из вызовов. Если бы возврат вызова API к каждому поставщику занимал одну секунду, пришлось бы ждать три секунды, прежде чем появится возможность решить, у кого следует оформить заказ.

Оптимально было бы выполнить эти три запроса параллельно. Тогда общая задержка операции будет равна самому медленному вызову API, а не сумме задержек каждого вызова.

Реактивные расширения и механизмы, такие как async/await, могут быть очень полезны для параллельного выполнения вызовов, и это должно привести к значительному снижению времени ожидания некоторых операций.

Где использовать

Вызовы типа «запрос — ответ» идеально подходят для ситуации, в которой результат запроса необходим для дальнейшей обработки. Они также очень хорошо вписываются, когда микросервису требуется знать, не сработал ли вызов, чтобы выполнить какое-то компенсирующее действие, например повторить попытку. Если любой из них соответствует вашей ситуации, то подход «запрос — ответ» будет для вас разумным выбором. Единственный оставшийся вопрос — что выбрать: синхронную или асинхронную реализацию с теми же компромиссами, которые мы обсуждали ранее.

Шаблон: событийное взаимодействие

Событийное взаимодействие выглядит довольно странно по сравнению с вызовами «запрос — ответ». Вместо того чтобы инициировать в другом сервисе какое-либо действие, микросервис выдает события, которые могут быть получены или не получены другими микросервисами. Это по своей сути асинхронное взаимодействие, поскольку прослушиватели событий будут работать в своем собственном потоке выполнения.

Событие — это утверждение о чем-то, что произошло внутри мира микросервиса, выдающего событие. Сервис, его отправляющий, может не знать ни о намерении других сервисов использовать это событие, ни даже об их существовании. Он выдает событие по необходимости, и на этом его обязанности заканчиваются.

На рис. 4.11 показано, что сервис `Склад` генерирует события, связанные с процессом упаковки заказа. Эти события принимаются двумя микросервисами, `Уведомления` и `Запасы`, которые реагируют соответствующим образом. Микросервис `Уведомления` отправляет клиенту электронное письмо, чтобы информировать его об изменениях в статусе заказа, в то время как микросервис `Запасы` может обновлять уровни запасов по мере того, как товары собираются в заказ клиента.

Сервис `Склад` просто транслирует события, ожидая соответствующей реакции заинтересованных сторон. Он не знает, кто будет получателем событий, что делает событийные взаимодействия в целом слабо связанными. При работе с моделью «запрос — ответ» можно было ожидать, что сервис `Склад` сообщит сервису `Уведомления`, когда необходимо отправлять электронные письма. В такой модели `Складу` нужна информация, какие события требуют создания уведомления для клиента. При событийном взаимодействии ответственность за это переносится на микросервис `Уведомления`.

Цель, стоящую за событием, можно рассматривать как противоположность запросу. Отправитель событий оставляет за получателями право решать, что делать. При использовании модели «запрос — ответ» микросервис, отправляющий запрос, ожидает определенной реакции и сообщает другому сервису, что должно

произойти дальше. Это означает, что при работе с моделью «запрос — ответ» отправитель запроса должен знать, что может сделать нижестоящий получатель. В итоге мы получаем большую степень предметной связанности. При событийном взаимодействии отправителю событий не обязательно знать о нижестоящих микросервисах и об их действиях, в результате чего связанность значительно снижается.

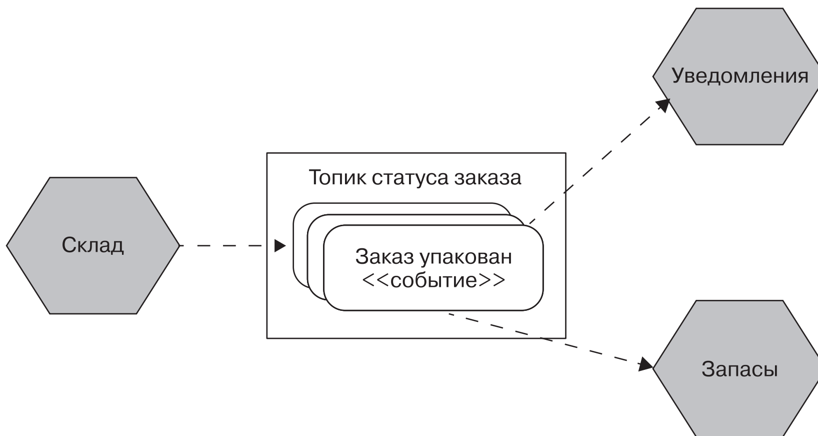


Рис. 4.11. Сервис Склад выдает события, на которые подписываются отдельные нижестоящие микросервисы

Распределение ответственности в наших событийных взаимодействиях похоже на распределение, наблюдаемое в организациях, пытающихся создать более автономные команды. Вместо того чтобы нести всю ответственность централизованно, мы перекладываем ее на сами команды для их более автономной работы (к этой концепции вернемся в главе 15). В нашем конкретном примере ответственность переходит от сервиса `Склад` в сервисы `Уведомления` и `Оплата`. Это может помочь уменьшить сложность сервиса `Склад` и привести к более равномерному распределению «умных» функций в нашей системе. Мы рассмотрим эту идею более подробно, когда сравним хореографию и оркестрацию в главе 6.

СОБЫТИЯ И СООБЩЕНИЯ

Иногда встречаются ситуации, когда термины «сообщения» и «события» путаются. *Событие* — это факт, уведомление о случившемся наряду с некоторой информацией о том, что именно произошло. *Сообщение* — это то, что мы отправляем через асинхронный механизм связи, например через брокер сообщений.

При событийном сотрудничестве мы транслируем это событие, помещая его в сообщение. Сообщение — это средство, а событие — полезная нагрузка.

Точно так же может потребоваться отправить запрос в качестве полезной нагрузки сообщения. В этом случае мы бы реализовали асинхронную форму «запрос — ответ».

Реализация

Здесь необходимо рассмотреть два основных способа: способ, которым микросервисы производят события, и способ, которым потребители узнают, что эти события произошли.

Традиционно брокеры сообщений, такие как RabbitMQ, пытаются справиться с обеими проблемами. Производители используют API для публикации события брокеру, который, в свою очередь, обрабатывает подписки, позволяя потребителям получать информацию о наступлении события. Эти брокеры могут даже управлять состоянием потребителей, например помогая отслеживать ранее поступавшие сообщения. Обычно эти системы проектируются с расчетом на масштабируемость и отказоустойчивость, и этих качеств не так просто достичь. Стремление их обеспечить может усложнить процесс разработки, поскольку вам, возможно, потребуется запустить еще одну систему для создания и тестирования ваших сервисов. Для поддержания этой инфраструктуры в рабочем состоянии также могут потребоваться дополнительные машины и особые знания. Но если удастся это осуществить, то такой способ реализации слабо связанных событийных архитектур может стать невероятно эффективным.

Однако будьте осторожны с миром промежуточного программного обеспечения, в котором брокер сообщений является лишь небольшой частью. Очереди сами по себе вполне разумная и полезная вещь. Однако поставщики, как правило, хотят упаковать в один пакет большое количество ПО, и это может привести к тому, что все больше и больше интеллектуальных функций будут внедряться в промежуточное программное обеспечение, о чем свидетельствует появление сервисной шины предприятия. Убедитесь, что знаете, что получаете в пакете поставки ПО: сохраните свое промежуточное ПО «глупым», а конечные точки — «умными».

Другой подход заключается в использовании HTTP для распространения событий. Atom — это спецификация, совместимая с REST, она определяет семантику (среди прочего) для публикации каналов ресурсов. Существует множество клиентских библиотек, позволяющих создавать и использовать эти каналы. Таким образом, сервис поддержки клиентов может публиковать событие в такой ленте всякий раз, когда меняется сервис поддержки клиентов. Потребители просто просматривают ленту в поисках изменений. С одной стороны, возможность повторно использовать существующую спецификацию Atom и любые связанные с ней библиотеки полезна и мы знаем, что HTTP очень хорошо справляется с масштабированием. Однако такое использование HTTP не совсем эффективно при минимальной задержке (в чем преуспевают некоторые брокеры сообщений). Также необходимо иметь в виду, что потребители должны отслеживать, какие сообщения они видели, и управлять своим собственным расписанием поллинга.

Иногда люди тратят целую вечность на реализацию все большего и большего количества моделей поведения, получаемых «из коробки» с соответствующим брокером сообщений, чтобы заставить Atom работать в некоторых ситуациях. Например, *шаблон конкурирующих потребителей* описывает метод, при котором

вызываются несколько экземпляров воркеров (worker, или тот, кто осуществляет процедуру; рабочий процесс) для конкуренции за сообщения. Это хорошо работает при обработке списка независимых заданий (мы вернемся к этому в следующей главе). Однако хотелось бы избежать случая, когда два или более воркера выполняют одну и ту же задачу несколько раз из-за одинакового сообщения. С помощью брокера сообщений стандартная очередь справится с этой проблемой. С применением Atom потребуется управлять нашим собственным общим состоянием среди всех воркеров, чтобы попытаться уменьшить шансы на повторное выполнение.

Если у вас уже есть хороший, надежный брокер сообщений, используйте его для обработки публикации и подписки на события. Если нет, рассмотрите вариант использования Atom, но не забывайте о требующихся затратах. Если вам нужно все больше и больше поддержки, предоставляемой брокером сообщений, в определенный момент вы, возможно, захотите изменить свой подход.

Фактически пересылки по этим асинхронным протоколам осуществляются по тем же принципам, что и при синхронной связи. Если в настоящее время вас устраивает кодирование запросов и ответов с использованием формата JSON, придерживайтесь его.

Что входит в событие

На рис. 4.12 показано событие, которое транслируется из микросервиса `Покупатель` и информирует заинтересованные стороны о регистрации в системе нового клиента. Два нижестоящих микросервиса, `Лояльность` и `Уведомления`, заинтересованы в этом событии. Микросервис `Лояльность` реагирует на получение события, создавая учетную запись для нового клиента, чтобы он мог начать зарабатывать баллы, в то время как микросервис `Уведомления` отправляет электронное письмо вновь зарегистрированному клиенту, приветствуя его от лица MusicCorp.

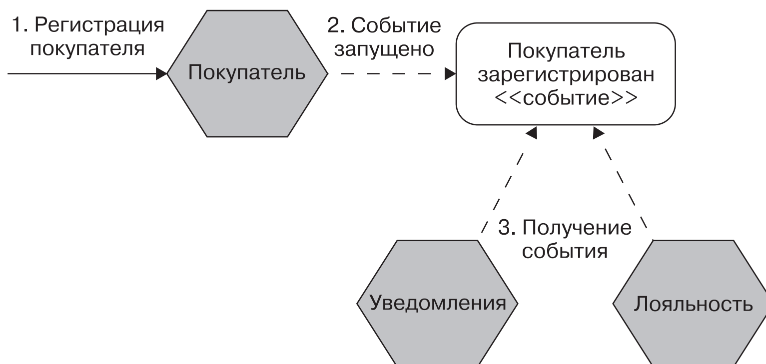


Рис. 4.12. Микросервисы Уведомления и Лояльность получают событие при регистрации нового клиента

Своим запросом мы просим микросервис что-то сделать и предоставляем необходимую информацию для выполнения запрошенной операции. С помощью событий мы транслируем факт, *способный* заинтересовать другие стороны. Но если сервис, отправляющий событие, не может и не должен знать, кто получает событие, как мы узнаем, какая информация может понадобиться другим сторонам? Что именно должно быть внутри события?

Просто идентификатор

Один из вариантов заключается в том, чтобы событие просто содержало идентификатор для вновь зарегистрированного клиента, как показано на рис. 4.13. Микросервису `Лояльность` нужен только этот идентификатор для создания соответствующей учетной записи лояльности, поэтому у него есть вся необходимая информация. Однако, хотя микросервис `Уведомления` знает, что при получении такого события ему необходимо отправить приветственное электронное письмо, для выполнения своей работы ему потребуется дополнительная информация, по крайней мере адрес электронной почты и, возможно, имя клиента, чтобы придать письму индивидуальный характер. Поскольку эта информация не содержится в событии, которое сервис `Уведомления` получает, у него нет другого выбора, кроме как извлечь эту информацию из микросервиса `Покупатель` (см. рис. 4.13).

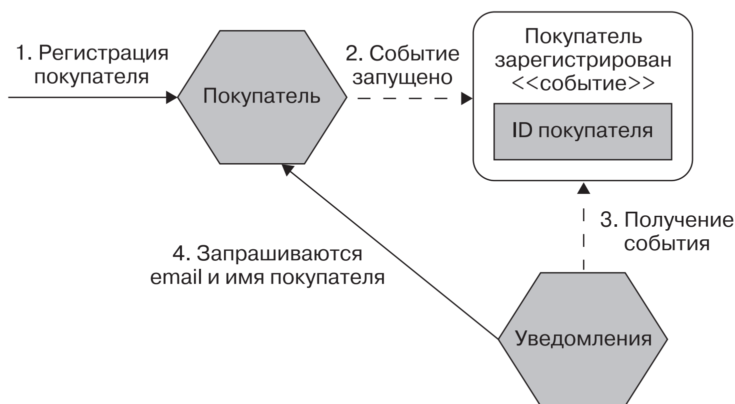


Рис. 4.13. Микросервису Уведомления необходимо запросить у микросервиса Покупатель дополнительные сведения, которые не включены в событие

У такого подхода есть некоторые недостатки. Микросервису `Уведомления` теперь необходимо знать о микросервисе `Покупатель`, что добавляет дополнительную предметную связанность. Хотя она, как обсуждалось в главе 2, по-прежнему слабая, все же хотелось бы по возможности избежать ее. Если бы событие, полученное микросервисом `Уведомления`, содержало всю необходимую

информацию, то этот обратный вызов не потребовался бы. Обратный вызов от принимающего микросервиса может привести еще к одному серьезному недостатку: в ситуации с большим количеством принимающих микросервисов сервис, отправляющий событие, в результате может получить шквал запросов. Представьте, что все пять разных микросервисов получили одно и то же событие по созданию клиента и всем им нужно немедленно запросить дополнительную информацию у микросервиса `Покупатель`. По мере увеличения числа микросервисов, заинтересованных в конкретном событии, влияние этих вызовов может стать значительным.

Полностью детализированные события

Альтернатива, которую я предпочитаю, заключается в том, чтобы поместить все в событие, которым вам было бы удобно поделиться через API. Если позволить микросервису `Уведомления` запрашивать адрес электронной почты и имя определенного клиента, почему бы просто не поместить эту информацию в событие сразу? На рис. 4.14 показано, что сервис `Уведомления` теперь более самодостаточен и способен выполнять свою работу без необходимости общаться с микросервисом `Покупатель`. На самом деле ему, возможно, никогда не понадобится знать о существовании микросервиса `Покупатель`.

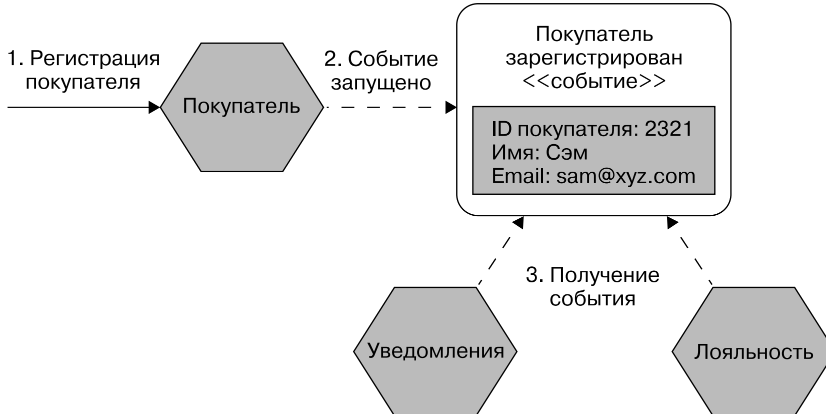


Рис. 4.14. Событие, содержащее больше информации, может позволить принимающим микросервисам действовать, не требуя дальнейших обращений к источнику события

В дополнение к тому, что события с большим количеством информации могут обеспечить более слабую связанность, такие события также могут использоваться в качестве архивной справки о произошедшем с определенной сущностью. Это поможет вам при внедрении системы аудита или даже даст возможность воссоздать сущность в конкретный момент времени — это означает, что данные

события допускается использовать как часть шаблона источников событий, концепции, которую мы кратко рассмотрим далее.

Хотя этот подход, безусловно, для меня предпочтителен, он не лишен недостатков. Для начала, если объем связанных с событием данных внушителен, у нас могут возникнуть опасения по поводу размера события. У современных брокеров сообщений (при условии, что вы используете один из них для реализации механизма трансляции событий) довольно жесткие ограничения по размеру сообщения. Максимальный размер сообщения по умолчанию в Kafka составляет 1 Мбайт, а последняя версия RabbitMQ поддерживает теоретический верхний предел 512 Мбайт для одного сообщения (по сравнению с предыдущим 2 Гбайт!). Хотя вполне вероятно, что при таких больших размерах сообщений возникнут некоторые проблемы с производительностью. Но даже сообщение размером 1 Мбайт на Kafka предоставляет достаточно возможностей для отправки довольно большого объема данных. В конечном счете, если вы углубляетесь в область, в которой приходится уделять особое внимание размеру событий, я бы рекомендовал применить гибридный подход, при котором часть информации содержится в событии, но при необходимости можно просмотреть другие (более объемные) данные.

На рис. 4.14 сервису `Лояльность` не нужно знать адрес электронной почты или имя клиента, и тем не менее он получает эти данные через событие. Это может вызвать проблемы при попытке ограничить область, в которой микросервисы могут видеть личную информацию (или PII), данные платежных карт или аналогичные конфиденциальные данные. Решить проблему поможет отправка событий двух разных типов: одно содержит PII и доступно для просмотра некоторыми микросервисами, а второе не хранит PII и транслируется более широко. Это добавляет сложности с точки зрения управления видимостью различных событий и обеспечения их запуска. Что происходит, когда микросервис отправляет событие первого типа, но разрушается до отправки второго?

Еще один момент: как только мы помещаем данные в событие — оно становится частью нашего контракта с потребителем. Необходимо понимать, что удаление поля из события может нарушить работу сторонних потребителей. Скрытие информации по-прежнему является важной концепцией в событийном сотрудничестве — чем больше данных мы помещаем в событие, тем больше предположений о событии будет у потребителей. Я могу помещать информацию в событие при условии готовности поделиться этими данными через API запроса — ответа.

Где использовать

Событийное взаимодействие лучше всего использовать в ситуациях, когда информацию требуется транслировать, и в ситуациях, когда вы инвертируете цель. Большую привлекательность обретает переход от модели указания другим блокам, что делать, к предоставлению нижестоящим микросервисам возможности решать такие вопросы самостоятельно.

В ситуации, когда вы уделяете больше внимания слабой связанности, чем другим факторам, событийное взаимодействие будет более привлекательным.

Однако нельзя не отметить, что при таком стиле связи часто возникают новые источники проблем, особенно если у вас мало опыта. Если вы не уверены в этой форме коммуникации, помните, что наша микросервисная архитектура может (и, скорее всего, будет) сочетать различные стили взаимодействия. Нет необходимости идти ва-банк с этим шаблоном. Возможно, стоит начать с одного события и развивать модель дальше.

Лично я заметил, что тяготею к событийному взаимодействию почти по умолчанию. Мой мозг, похоже, перестроил себя так, что такие типы коммуникации кажутся мне просто *очевидными*. Однако из-за подобной субъективной оценки ничего не остается, кроме как сказать, что это *кажется* правильным, чем попытаться объяснить *почему*. Но это всего лишь мое собственное предубеждение — я, естественно, тяготею к тому, что знаю, основываясь на своем собственном опыте. Велика вероятность, что влечение к этой форме взаимодействия почти полностью обусловлено моим предыдущим неудачным опытом работы с чрезмерно связанными системами. Я мог бы быть просто генералом, переживающим битву снова и снова, не задумываясь о том, что, возможно, на этот раз все на самом деле по-другому.

Отбросив в сторону свои собственные предубеждения, могу сказать, что проще встретить команду, выполняющую замену взаимодействия «запрос — ответ» на событийные типы взаимодействия, чем наоборот.

Действуйте с осторожностью

Некоторые из этих асинхронных штучек кажутся забавными, правда? Есть ощущение, что событийные архитектуры приведут к созданию значительно менее связанных и масштабируемых систем. И это правда. Но подобные стили взаимодействия на самом деле приводят к повышению общей сложности системы. Эта сложность связана не только с управлением публикацией и подпиской на сообщения, как мы только что обсуждали, но и с другими встречающимися проблемами. Например, рассматривая длительный асинхронный процесс «запрос — ответ», необходимо продумать, что делать, когда ответ вернется. Возвращается ли он к тому же узлу, который инициировал запрос? Если да, то что произойдет, если данный узел выйдет из строя? Если нет, нужно ли где-то хранить информацию, чтобы иметь возможность реализовать соответствующую реакцию? Кратковременной асинхронностью иногда проще управлять, если у вас есть правильные API, но даже в таком случае это другой способ мышления для программистов, привыкших к внутрипроцессным синхронным вызовам сообщений.

Пришло время для поучительной истории. Еще в 2006 году я работал над созданием системы ценообразования для банка. Мы смотрели на рыночные

события и решали, какие товары в портфеле нуждаются в переоценке. После определения списка задач, над которыми нужно поработать, мы поместили их все в очередь сообщений. Была использована сетка для создания пула воркеров по ценообразованию, что позволяло увеличивать и уменьшать масштаб блока ценообразования по запросу. В этом случае использовался шаблон конкурирующих потребителей, каждый из которых поглощал сообщения как можно быстрее, пока не заканчивались данные для обработки.

Система была запущена и работала, и мы были вполне довольны собой. Но однажды, сразу после релиза, мы столкнулись с неприятной проблемой: наши воркеры продолжали умирать. И умирать. И умирать.

В конце концов мы отыскали проблему. Вкралась ошибка, из-за которой запрос определенного типа приводил к сбою воркера. Мы использовали очередь транзакций: когда воркер умирал, время ожидания его блокировки запроса истекало и запрос на ценообразование помещался обратно в очередь — только для того, чтобы другой воркер забрал его и завершился. Это был классический пример того, что Мартин Фаулер называет *катастрофической отказоустойчивостью* (https://oreil.ly/8HwcP).

Помимо самой ошибки, нам не удалось указать максимальный предел количества повторных попыток для задания в очереди. Поэтому мы исправили недочет и настроили максимальное количество повторных попыток. Но пришли к выводу, что нужен способ просматривать и потенциально воспроизводить эти сообщения. В итоге нам пришлось внедрить «лазарет» сообщений (или очередь недоставленных сообщений), куда они отправлялись в случае сбоя. Мы также создали пользовательский интерфейс для просмотра этих сообщений и повторения их при необходимости. Такого рода проблемы не сразу бросаются в глаза, если вы знакомы только с синхронной связью типа «точка — точка».

Сложность, связанная с событийной архитектурой и асинхронным программированием, в целом говорит, что следует ограничить внедрение этих идей. Убедитесь, что у вас налажен хороший мониторинг, и внимательно рассмотрите возможность использования идентификаторов корреляции, позволяющих отслеживать запросы через границы процесса (подробнее — в главе 10).

Я также настоятельно рекомендую ознакомиться с книгой «Шаблоны интеграции корпоративных приложений» Грегора Хоупа и Бобби Вулфа[1], в которой содержится гораздо больше подробностей о различных шаблонах обмена сообщениями.

Однако стоит быть честными в отношении стилей интеграции, которые мы могли бы считать «более простыми», — проблемы, связанные с пониманием того, работает что-то или нет, не ограничиваются асинхронными формами интеграции. Из-за чего мог произойти тайм-аут при синхронном блокирующем

[1] *Хоуп Г., Вулф Б.* Шаблоны интеграции корпоративных приложений.

вызове? Из-за потери запроса? Или ответа? Если вы повторите попытку, но первоначальный запрос на самом деле прошел, что тогда? Вот тут-то и возникает идемпотентность — тема, которую мы рассмотрим в главе 12.

Что касается обработки сбоев: возможно, синхронные блокирующие вызовы создадут нам столько же проблем, сколько выяснение, а произошло ли вообще что-то. Просто подобные проблемы могут быть нам лучше известны!

Резюме

В этой главе мы проанализировали некоторые ключевые стили взаимодействия микросервисов и обсудили различные компромиссы. Не всегда существует единственно *правильный* вариант, но, надеюсь, я собрал достаточно подробной информации о синхронных и асинхронных вызовах, о стилях событийной взаимосвязи и моделях «запрос — ответ», чтобы помочь выбрать стиль вызова, подходящий под ваш конкретный случай. Мое собственное пристрастие к асинхронному событийному взаимодействию обусловлено не только личным опытом, но и моим неприятием связанности в целом. Однако подобный стиль связи сопряжен со значительной сложностью, которую нельзя игнорировать, ведь каждая ситуация уникальна.

В этой главе я кратко упомянул несколько конкретных технологий, которые можно использовать для реализации описанных стилей. Теперь мы готовы приступить ко второй части книги — реализации. В следующей главе более подробно рассмотрим реализацию микросервисной коммуникации.

ЧАСТЬ II

Реализация

ГЛАВА 5

Реализация коммуникации микросервисов

Как обсуждалось в предыдущей главе, технологию нужно выбирать исходя из желаемого стиля коммуникации. Выбор между блокирующими синхронными и неблокирующими асинхронными вызовами, между коммуникацией в событийном стиле и стиле «запрос — ответ» поможет вам сократить очень длинный список доступных технологий. В этой главе мы рассмотрим некоторые технологии, обычно используемые для коммуникации микросервисов.

В поисках идеальной технологии

Существует огромное множество вариантов взаимодействия микросервисов. Но какой из них правильный: SOAP, XML-RPC, REST, gRPC? И всегда появляются новые. Поэтому, прежде чем мы обсудим конкретную технологию, давайте подумаем, чего мы хотим от нее.

Упростите обратную совместимость

При внесении изменений в микросервисы необходимо убедиться, что мы не нарушаем совместимость с любыми потребляющими микросервисами. Таким образом, мы гарантируем, что любая выбираемая технология позволит легко вносить обратно совместимые изменения. Простые операции, вроде добавления новых полей, не должны нарушать работу клиентов. В идеале мы хотим получить возможность проверять внесенные изменения на предмет обратной совместимости и иметь возможность получить эту обратную связь до того, как запустим микросервис в эксплуатацию.

Сделайте свой интерфейс выразительным

Важно, чтобы интерфейс, который микросервис предоставляет внешнему миру, был выразительным, то есть понятным для потребителя. Но это также означает, что разработчику микросервиса должно быть ясно, какая функциональность останется неизменной для внешних потребителей, ведь мы хотим избежать ситуации, в которой изменение микросервиса приводит к случайному нарушению совместимости.

Подобные схемы могут иметь большое значение для обеспечения выразительности интерфейса, предоставляемого микросервисом. Некоторые из рассматриваемых технологий требуют использования схемы, для остальных это необязательно. В любом случае я настоятельно рекомендую использовать выразительную схему, а также подтверждающую документацию, чтобы было ясно, какую функциональность потребитель может ожидать от микросервиса.

Следите за тем, чтобы ваши API не зависели от технологий

Если вы долго работаете в ИТ-индустрии, вам не нужно объяснять, что это быстро меняющаяся сфера. Единственная имеющаяся уверенность — это *наличие* перемен. Постоянно появляются новые инструменты, фреймворки и языки, реализующие новые идеи, способные помочь работать быстрее и эффективнее. Прямо сейчас у вас может быть магазин, написанный на платформе .NET. Но что будет через год или через пять лет? Что, если вы захотите поэкспериментировать с альтернативным технологическим стеком, который может повысить производительность?

Я думаю, что надо всегда оставаться открытыми для новых возможностей, вот почему я такой поклонник микросервисов. Очень важно сделать так, чтобы API, используемые для связи между микросервисами, не зависели от технологий. Это означает отказ от технологии интеграции, диктующей, какие технологические стеки можно применять для реализации микросервисов.

Сделайте свой сервис простым для потребителей

Необходимо упростить потребителям использование нашего микросервиса. Прекрасно оформленный микросервис не представляет большого значения, если его стоимость заоблачная! Итак, давайте подумаем, что позволит клиентам легко пользоваться нашим замечательным новым сервисом. Было бы идеальным предоставить клиентам полную свободу в выборе технологий. С другой стороны, предоставление клиентской библиотеки может облегчить внедрение. Однако часто такие библиотеки несовместимы с другими функциями, которые мы хотим получить. Например, мы могли бы использовать клиентские библиотеки, чтобы упростить работу для потребителей, но это может усилить связанность.

Скройте детали внутренней реализации

Необходимо, чтобы наши потребители не были привязаны к внутренней реализации, поскольку это приводит к усилению связанности. При попытке что-то преобразовать внутри микросервиса мы рискуем нарушить работу потребителей, ожидая от них внесения изменений. Это увеличивает стоимость, чего мы и пытаемся избежать. Нам с меньшей вероятностью захочется вносить изменения из-за страха обновления потребителей, что может привести к увеличению технического долга в рамках сервиса. Поэтому следует избегать любых технологий, подталкивающих нас к раскрытию внутренних деталей представления.

Выбор технологий

Существует целый ряд технологий, которые можно использовать, но вместо того, чтобы целиком рассматривать длинный список вариантов, я выделю некоторые из наиболее популярных и интересных.

Удаленный вызов процедур

Фреймворки, позволяющие применять локальные вызовы методов в удаленном процессе. Распространенные варианты включают SOAP и gRPC.

REST

Архитектурный стиль, где вы предоставляете ресурсы (`Клиент`, `Заказ` и т. д.), к которым можно получить доступ с помощью общего набора команд (GET, POST). В REST немного больше возможностей, и мы скоро вернемся к этой теме.

GraphQL

Относительно новый протокол, позволяющий потребителям определять пользовательские запросы, которые будут извлекать информацию из нескольких нижестоящих микросервисов, фильтруя результаты, чтобы возвращать только требующиеся данные.

Брокеры сообщений

Промежуточное ПО, позволяющее осуществлять асинхронную связь через очереди или топики.

Удаленные вызовы процедур

Удаленный вызов процедур (remote procedure call, RPC) относится к технике реализации локального вызова и его выполнения где-то на удаленном сервисе. Существует несколько различных реализаций RPC. Большая часть технологий в этой области требует явной схемы, применяемой, например, в системах SOAP или gRPC. В контексте RPC схема часто упоминается как язык описания интер-

фейса (interface definition language, IDL), а SOAP ссылается на свой формат схемы как на язык описания веб-сервисов (web service definition language, WSDL). Использование отдельной схемы упрощает создание клиентских и серверных заглушек для разных технологических стеков. Так, например, у меня мог бы быть сервер Java, предоставляющий интерфейс SOAP, и клиент .NET, созданный на основе одного и того же определения интерфейса WSDL. Другие технологии, такие как Java RMI, требуют более тесной связи между клиентом и сервером, где нужно, чтобы оба использовали одну и ту же базовую технологию, но избегали необходимости в явном определении сервиса, поскольку неявное определение предоставляется определениями типов Java. Однако все эти технологии имеют общую особенность: они делают удаленный вызов похожим на локальный.

Как правило, использование технологии RPC означает, что вы приобретаете протокол сериализации. Фреймворк RPC определяет, как данные сериализуются и десериализуются. Например, gRPC использует для этой цели формат сериализации Protocol buffers. Некоторые реализации привязаны к определенному сетевому протоколу (например, SOAP, номинально использующий HTTP), в то время как другие могут позволить вам применить различные типы сетевых протоколов, предоставляющих дополнительные функции. Например, TCP предлагает гарантии доставки, а UDP — нет, хотя требует гораздо меньших накладных расходов. Таким образом, вы можете выбирать разные сетевые технологии для различных сценариев.

RPC-фреймворки с явной схемой очень упрощают генерацию клиентского кода, что позволяет избежать необходимости в клиентских библиотеках, поскольку любой клиент может запросто сгенерировать свой собственный код в соответствии со спецификацией сервиса. Однако для того, чтобы генерация кода на стороне клиента работала, ему нужен какой-то способ вывести схему. Другими словами, потребитель должен иметь доступ к схеме, прежде чем планировать совершить вызов. Платформа Avro RPC является исключением, поскольку она дает возможность отправлять полную схему вместе с полезной нагрузкой, позволяя клиентам динамически интерпретировать схему.

Простота генерации клиентского кода — одно из основных преимуществ RPC. Возможность вызвать обычный метод и теоретически проигнорировать все остальное — настоящая находка.

Проблемы

Как вы уже поняли, у технологии RPC есть ряд преимуществ, но она не лишена и недостатков. И некоторые реализации RPC могут быть более проблематичными, чем другие. Многие из проблем вполне возможно решить, но они заслуживают дальнейшего изучения.

Технологическая связанность. Некоторые механизмы RPC, такие как Java RMI, сильно привязаны к конкретной платформе, что может ограничить использование технологии на стороне клиента и сервера. Системы Thrift и gRPC

обладают впечатляющей поддержкой альтернативных языков, что несколько уменьшает озвученный недостаток. Тем не менее технология RPC иногда имеет ограничения по совместимости.

В некотором смысле эта технологическая связанность может быть формой раскрытия внутренних технических деталей реализации. Например, использование RMI связывает с JVM не только клиент, но и сервер.

Однако стоит отметить, что существует ряд реализаций RPC, не имеющих подобного ограничения, — gRPC, SOAP и Thrift. Все это примеры, обеспечивающие совместимость между различными стеками технологий.

Локальные и удаленные вызовы различаются. Основная идея RPC — скрыть сложность удаленного вызова. Однако не стоит переусердствовать. Необходимо помнить, что удаленные и локальные вызовы методов — это разные вещи, хоть и некоторые формы RPC стремятся их уравнять и скрыть этот факт. Можно совершать большое количество локальных вызовов в процессе, не слишком беспокоясь о производительности. При использовании RPC затраты на маршалинг и анмаршалинг полезных нагрузок могут быть значительными, не говоря уже о времени, необходимом на отправку данных по сети. Это означает, что разработка API для удаленных интерфейсов потребует подхода, отличного от разработки локальных версий. Если вы просто возьмете локальный API и не задумываясь попытаетесь сделать его границей сервиса, скорее всего, у вас возникнут проблемы. В некоторых случаях, если абстракция слишком непрозрачна, разработчики могут использовать удаленные вызовы, даже не подозревая об этом.

Вам нужно подумать о самой сети. Одно из заблуждений о распределенных вычислениях — «Сеть надежна» (https://oreil.ly/8J4Vh). Сети *ненадежны*. Они могут не сработать, даже если общающиеся клиент и сервер работоспособны. Ожидать стоит чего угодно: моментального выхода из строя, постепенной деградации, даже искажения ваших пакетов. Исходите из того, что ваши сети кишат злонамеренными сущностями, готовыми в любой момент все сломать. В итоге вы столкнетесь с такими типами сбоев, с которыми, возможно, никогда не имели дело в более простом монолитном ПО. Вызвать сбой может как возвращенное удаленным сервером сообщение об ошибке, так и неправильно совершенный вызов. И как в таком случае поступить? А что вы будете делать, если удаленный сервер просто начнет медленно отвечать? Мы рассмотрим эту тему, когда будем говорить об отказоустойчивости в главе 12.

Уязвимость. Некоторые из самых популярных реализаций RPC могут привести к отдельным неприятным формам уязвимости. Java RMI является очень хорошим примером. Давайте рассмотрим довольно простой Java-интерфейс, который мы решили сделать удаленным API для нашего сервиса `Покупатель`. В примере 5.1 объявлены методы, которые мы собираемся использовать удаленно. Затем Java RMI генерирует клиентские и серверные заглушки для нашего метода.

Пример 5.1. Определение конечной точки сервиса с помощью Java RMI

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface CustomerRemote extends Remote {
  public Customer findCustomer(String id) throws RemoteException;

  public Customer createCustomer(
    String firstname, String surname, String emailAddress)
    throws RemoteException;
}
```

В этом интерфейсе функция `createCustomer` принимает имя, фамилию и адрес электронной почты. Что произойдет, если мы разрешим создание объекта `Customer` (Покупатель), используя только адрес электронной почты? На этом этапе можно довольно легко добавить новый метод, например:

```
  ...
  public Customer createCustomer(String emailAddress) throws RemoteException;
  ...
```

Проблема в том, что теперь клиентам, желающим использовать новый метод, понадобятся новые серверные заглушки. И в зависимости от характера изменений в спецификации, потребителям, которым новый метод не нужен, также придется их использовать. Конечно, с этим можно справиться, но только до определенного момента. Подобные изменения происходят довольно часто. Конечные точки RPC нередко содержат большое количество методов для различных способов создания объектов или взаимодействия с ними. Отчасти это связано с тем, что мы все еще думаем об этих удаленных вызовах как о локальных.

Однако есть и другой вид уязвимости. Давайте посмотрим, как выглядит объект `Customer`:

```
  public class Customer implements Serializable {
    private String firstName;
    private String surname;
    private String emailAddress;
    private String age;
  }
```

Что, если поле `age` (возраст) в наших объектах `Customer` окажется невостребованным ни одним из наших потребителей? Логично было бы удалить его. Но если в серверной реализации убрать поле `age` из определения этого типа, и не сделать то же самое для всех потребителей, то даже если они никогда не использовали это поле, код, связанный с десериализацией объекта `Customer` на стороне потребителя, будет нарушен. Чтобы внедрить данное изменение, нам потребуется преобразовать код клиента для поддержки нового определения и развернуть эти обновленные клиенты одновременно с развертыванием новой

версии сервера. Это ключевая проблема любого механизма RPC, который поощряет использование двоичной генерации заглушек: вы не можете разделить развертывание клиента и сервера. Если вы используете эту технологию, в будущем станете заложником жестко регламентированного порядка выпуска релизов.

Похожие проблемы возникают при необходимости перестроить объект `Customer`, даже если поля не удалялись. Например, если мы хотим инкапсулировать поля `firstName` и `surname` в новый тип `naming` для упрощения управления. Мы могли бы, конечно, передать типы словарей в качестве параметров наших вызовов, но на данном этапе теряются многие преимущества сгенерированных заглушек, потому что нам все равно придется вручную сопоставлять и извлекать нужные поля.

На практике объекты, используемые как часть двоичной сериализации при передаче данных по сети, стоит рассматривать как типы «только для расширения». Эта уязвимость приводит к тому, что типы подвергаются воздействию процесса передачи по сети и превращаются в массу полей, часть из которых больше не используются, но и не могут быть безопасно удалены.

Где использовать

Несмотря на недостатки, мне все равно очень нравится RPC, а его более современные реализации, такие как gRPC, превосходны, в то время как у аналогов имеются существенные проблемы. У Java RMI, например, есть ряд проблем, связанных с уязвимостью и ограниченным выбором технологий, а SOAP довольно тяжеловесен с точки зрения разработчика, особенно по сравнению с более современными вариантами.

Просто имейте в виду некоторые потенциальные подводные камни, связанные с RPC, если собираетесь выбрать эту модель. Не абстрагируйте свои удаленные вызовы до такой степени, чтобы сеть была полностью скрыта, и убедитесь, что сохраняется возможность развивать интерфейс сервера без необходимости настаивать на поэтапном обновлении для клиентов. Например, важно найти правильный баланс для вашего клиентского кода. Убедитесь, что клиенты не забывают о сетевом вызове. Клиентские библиотеки часто используются в контексте RPC, и при неправильном структурировании они могут вызвать проблемы. Вскоре мы поговорим о них подробнее.

Если бы я рассматривал варианты из этой области, gRPC был бы в топе моего списка. Он позволяет использовать преимущества HTTP/2, обладает некоторыми впечатляющими характеристиками производительности и в целом прост в работе. Я также ценю gRPC за выстроенную экосистему, включая такие инструменты, как Protolock (https://protolock.dev). Их мы обсудим позже в этой главе, когда будем говорить о схемах.

Платформа gRPC хорошо подходит для синхронной модели «запрос — ответ», но также в состоянии работать с реактивными расширениями. Она занимает первое место в моем списке во всех случаях, когда у меня выстроен грамотный контроль как над клиентскими, так и над серверными конечными точками. Если появляется необходимость поддерживать широкий спектр других приложений, которым может потребоваться взаимодействие с вашими микросервисами, компиляция клиентского кода в соответствии со схемой на стороне сервера может стать проблемой. В этом случае, скорее всего, лучше подойдет какая-либо форма REST API через HTTP.

REST

Передача репрезентативного состояния (representational state transfer, REST) — это архитектурный стиль, вдохновленный Интернетом. В основе REST лежит множество принципов и ограничений, но мы сосредоточимся на том, что действительно помогает в решении проблем интеграции микросервисов и при поиске альтернативы RPC для интерфейсов сервисов.

Самое важное в REST — это концепция ресурсов. Представьте себе ресурсы как что-то, что знает сам сервис, например объект `Customer`. Сервер создает различные представления `Customer` по запросу. Внешнее отображение ресурса полностью отделено от способа его хранения внутри системы. Например, клиент может запросить JSON-представление объекта `Customer`, даже если оно хранится в совершенно другом формате. Как только клиент получает представление `Customer`, он может отправлять запросы на его изменение, а сервер может выполнять их или нет.

Существует много различных стилей REST, и я лишь вкратце коснусь их здесь. Настоятельно рекомендую вам взглянуть на модель зрелости Ричардсона (https://oreil.ly/AlDzu), где сравниваются различные стили REST.

REST чаще всего он работает через HTTP. Некоторые функции, предоставляемые протоколом HTTP как часть спецификации, упрощают реализацию REST, в то время как с другими протоколами вам придется обрабатывать эти функции самостоятельно.

REST и HTTP

Сам HTTP определяет ряд полезных возможностей, которые очень хорошо сочетаются с REST. Например, методы HTTP-запросов (такие как GET, POST и PUT) уже содержат хорошо понятные значения в спецификации HTTP относительно того, как они должны работать с ресурсами. Архитектурный стиль REST на самом деле позволяет этим методам вести себя одинаково на всех ресурсах, а спецификация HTTP определяет набор

доступных для использования команд. Например, GET извлекает ресурс идемпотентным способом, а POST создает новый ресурс. Это означает, что нам удастся избежать множества различных методов `createCustomer` или `editCustomer`. Теперь мы можем с помощью метода POST просто отправить представление клиента, чтобы запросить у сервера создание нового ресурса, а затем инициировать запрос GET для получения представления ресурса. Концептуально в этих случаях существует одна *конечная точка* в виде ресурса `Customer`, и операции, которые мы можем выполнять с ним, записываются в протокол HTTP.

HTTP также предоставляет обширную экосистему вспомогательных инструментов и технологий. Имеется возможность применять прокси-серверы кэширования HTTP, например Varnish, балансировщики нагрузки `mod_proxy` и многие инструменты мониторинга, уже с поддержкой HTTP. Эти инструменты позволяют обрабатывать большие объемы HTTP-трафика и маршрутизировать их разумно и довольно прозрачно. Допускается также использовать все доступные средства контроля безопасности с помощью HTTP для защиты способов коммуникации. Начиная с базовой аутентификации и заканчивая сертификатами клиентов, экосистема HTTP предоставляет множество инструментов для упрощения процесса обеспечения безопасности (подробнее рассмотрим эту тему в главе 11). Тем не менее, чтобы получить эти преимущества, необходимо эффективно использовать протокол HTTP. В противном случае он может быть таким же небезопасным и трудным для масштабирования, как и любая другая существующая технология.

Обратите внимание, что HTTP может применяться и для реализации RPC. Протокол SOAP, например, маршрутизируется по HTTP, но, к сожалению, он использует очень мало спецификаций. Команды игнорируются, как и коды ошибок HTTP. С другой стороны, gRPC был разработан специально для раскрытия потенциала HTTP/2, например для отправки нескольких потоков «запрос — ответ» по одному соединению. Однако при использовании gRPC вам недоступен REST из-за применения HTTP!

Гипермедиа как движок состояния приложения

Еще один принцип REST, способный помочь избежать связанности между клиентом и сервером, — это концепция *гипермедиа как двигатель состояния приложения* (часто сокращается как HATEOAS — от hypermedia as the engine of application state). Это довольно сложная формулировка, но сама концепция интересная, поэтому давайте немного разберем ее.

Гипермедиа — это расширение так называемого гипертекста или возможность открывать новые веб-страницы через ссылки на другие данные в различных форматах (например, текст, изображения, звуки). Идея, лежащая в основе HATEOAS, заключается во взаимодействии клиентов с сервером (потенциально приводящим к переходам состояний) через ссылки на другие ресурсы. Клиенту

не нужно знать, где именно на сервере содержится необходимая информация. Вместо этого он использует ссылки и перемещается по ним, чтобы найти то, что ему нужно.

Это не совсем обычная концепция, поэтому сделаем шаг назад и рассмотрим, как люди взаимодействуют с веб-страницей с элементами управления гипермедиа.

Вспомните сайт Amazon.com. Расположение корзины покупок менялось с течением времени. Изменилась графика, ссылка. Тем не менее мы понимаем, что такое корзина, и по-прежнему можем взаимодействовать с ней, даже если точная форма и основной элемент управления, используемый для ее представления, изменились. Вот как веб-страницы могут постепенно меняться с течением времени. До тех пор пока этот негласный договор между клиентом и веб-сайтом по-прежнему соблюдается, изменения не обязательно должны быть критическими.

С помощью средств управления гипермедиа мы пытаемся достичь такого же уровня понятности для наших электронных потребителей. Рассмотрим элемент управления гипермедиа, который мы могли бы использовать для MusicCorp. В нашем распоряжении имеется доступ к ресурсу, представляющему запись каталога для заданного альбома в примере 5.2. Наряду с информацией об альбоме мы видим ряд элементов управления гипермедиа.

Пример 5.2. Элементы управления гипермедиа, используемые в списке альбомов

```
<album>
  <name>Give Blood</name>
  <link rel="/artist" href="/artist/theBrakes" /> ❶
  <description>
    Awesome, short, brutish, funny and loud. Must buy!
  </description>
  <link rel="/instantpurchase" href="/instantPurchase/1234" /> ❷
</album>
```

❶ Данный элемент управления гипермедиа показывает, где находится информация об исполнителе.

❷ Если мы хотим купить альбом, теперь известно, где искать.

В этом примере есть два элемента управления гипермедиа. Клиент должен знать, что для получения информации об исполнителе ему необходимо перейти к элементу управления, связанному с `artist`, и что `instantpurchase` (мгновенная покупка) представляет собой часть протокола, используемого для покупки альбома. В данном случае семантика API во многом похожа на корзину с сайта покупок — это место хранения приобретаемых товаров.

Как клиенту, мне не обязательно знать, к какой схеме URI получить доступ, чтобы *купить* альбом. Мне всего лишь нужно зайти на сайт, найти элемент управления покупкой и перейти к нему. Местоположение элемента управления покупкой может измениться, как и сам URI, а сайт вообще может направить меня

на другой сервис, но мне как покупателю должно быть все равно. Это дает нам огромное снижение связанности между клиентом и сервером.

Здесь мы сильно абстрагируемся от базовых деталей. Допустимо было бы полностью изменить представление элемента управления, пока он все еще соответствует пониманию клиента, точно так же, как элемент управления корзиной покупок может превратиться из простой ссылки в более сложный элемент. Можно также свободно добавлять дополнительные элементы управления, возможно представляющие новые переходы состояний, которые мы способны выполнять на рассматриваемом ресурсе. Если бы мы коренным образом изменили семантику одного из элементов управления, чтобы он вел себя совсем по-другому, или если бы вообще его удалили, то в итоге нарушили бы работу наших потребителей.

Теоретически, используя эти элементы управления для разделения клиента и сервера, мы получаем значительные преимущества, компенсирующие увеличение времени, которое необходимо для запуска и начала эксплуатации этих протоколов. В теории все эти идеи кажутся разумными, однако на практике оказывается, что такая форма REST редко используется. Поэтому концепцию HATEOAS гораздо сложнее популяризировать среди тех, кто уже привержен использованию REST. По сути, многие идеи в REST основаны на создании распределенных гиперпространственных систем, а это не то, чем занимается большинство людей.

Проблемы

С точки зрения простоты использования исторически сложилось так, что нельзя генерировать код на стороне клиента для вашего REST через HTTP протокола приложения, в отличие от RPC. Это часто приводило к тому, что люди создавали REST API, предоставляющие пользователям клиентские библиотеки для использования. Эти клиентские библиотеки осуществляют привязку к API, чтобы упростить интеграцию с клиентом. Проблема заключается в том, что клиентские библиотеки могут вызывать определенные трудности во взаимодействии между клиентом и сервером. Мы обсудим это в разделе «DRY и опасности повторного использования кода в мире микросервисов» этой главы.

В последние годы эта проблема была отчасти решена. Спецификация OpenAPI (https://oreil.ly/Idr1p), выросшая из проекта Swagger, теперь предоставляет возможность определять достаточно информации о конечной точке REST, чтобы обеспечить генерацию клиентского кода на различных языках. Я лично не встречал команды, которые действительно бы использовали эту функциональность, даже если они уже применяли Swagger для документации. Мне кажется, что это связано с трудностями его адаптации к текущим API. У меня также есть опасения по поводу спецификации, которая ранее использовалась только для документации, а теперь применяется для определения более явного контракта.

Это может привести к гораздо более сложной спецификации: например, сравнение схемы OpenAPI со схемой Protocol buffers представляет собой довольно резкий контраст. Однако, несмотря на мои сомнения, хорошо, что этот вариант в принципе существует.

Производительность также может оказаться слабым звеном. Полезные нагрузки REST через HTTP на самом деле могут быть более компактными, чем SOAP, так как REST поддерживает альтернативные форматы, такие как JSON или даже бинарный, но он все равно далеко не такой экономичный, каким мог бы быть Thrift. Накладные расходы HTTP для каждого запроса также могут быть проблемой из-за требований к низкой задержке. Все основные протоколы HTTP, используемые в настоящее время, требуют применения протокола передачи данных (Transmission Control Protocol, TCP), который неэффективен по сравнению с другими сетевыми протоколами, а некоторые реализации RPC позволяют применять альтернативные TCP сетевые протоколы, такие как протокол пользовательских датаграмм (User Datagram Protocol, UDP).

Ограничения, накладываемые на HTTP из-за требования использовать TCP, устраняются. Протокол HTTP/3, который в настоящее время находится в процессе доработки, стремится перейти на использование более нового протокола QUIC. QUIC предоставляет те же возможности, что и TCP (например, улучшенные гарантии по сравнению с UDP), но с некоторыми существенными доработками, обеспечивающими снижение задержки и сокращение пропускной способности. Вероятно, пройдет несколько лет, прежде чем HTTP/3 окажет обширное влияние на Всемирную сеть, но, скорее всего, организации извлекут из этого выгоду раньше, чем HTTP/3 появится в их собственных сетях.

Что касается конкретно HATEOAS, вы гарантированно столкнетесь с дополнительными проблемами производительности. Поскольку клиентам необходимо переходить от одного элемента управления к другому, чтобы найти правильные конечные точки для совершаемой операции, это может привести к появлению очень сложных протоколов — для каждой операции потребуется несколько проходов по всем элементам управления. В конечном счете это компромисс. Если вы решите использовать REST в стиле HATEOAS, я бы посоветовал начать с того, чтобы ваши клиенты в первую очередь переходили по этим элементам управления, а при необходимости позже их можно оптимизировать. Помните, что HTTP предоставляет нам большой инструментарий «из коробки», о чем мы говорили выше. Недостатки преждевременной оптимизации были хорошо задокументированы ранее, поэтому не стоит подробно останавливаться на них здесь. Обратите также внимание, что не все из этих методов подойдут, так как большинство из них было разработано для создания распределенных гипертекстовых систем. Иногда вы можете поймать себя на мысли, что вам просто нужен старый добрый RPC.

Несмотря на эти минусы, REST через HTTP представляется разумным стандартным выбором для взаимодействия между сервисами. Если вы хотите узнать больше, я рекомендую книгу *REST in Practice: Hypermedia and Systems Architecture* (O'Reilly) от Джима Веббера, Саваса Парастатидиса и Иэна Робинсона, в которой подробно рассматривается тема REST по протоколу HTTP.

Где использовать

API на основе REST через HTTP представляется очевидным выбором для синхронного интерфейса по модели «запрос — ответ», если вы хотите разрешить доступ как можно большему количеству клиентов. Было бы ошибкой представлять себе REST API просто как выбор, «достаточно хороший для большинства ситуаций», однако в этом что-то есть. Это понятный стиль интерфейса, с которым многие знакомы, и он гарантирует совместимость с огромным разнообразием технологий.

Благодаря возможностям HTTP и степени, в которой REST использует эти возможности (а не скрывает их), API на основе REST превосходны в ситуациях, требующих крупномасштабного и эффективного кэширования запросов. Именно по этой причине они стали очевидным выбором для предоставления API внешним потребителям или клиентским интерфейсам. Однако они могут проигрывать по сравнению с более эффективными протоколами связи, и, хотя допустимо создавать протоколы асинхронного взаимодействия поверх API на основе REST, это не совсем подходящий вариант по сравнению с альтернативами для общего взаимодействия между микросервисами.

Несмотря на ясность целей HATEOAS, я не видел достаточно доказательств того, что дополнительная работа по внедрению этого стиля REST приносит ощутимые выгоды в долгосрочной перспективе. И не могу вспомнить, чтобы за последние несколько лет я разговаривал с какими-либо специалистами, внедряющими микросервисную архитектуру, которые сообщили бы о целесообразности использования HATEOAS. Очевидно, что мой собственный опыт — это нерепрезентативная выборка, и я не сомневаюсь, что для некоторых людей HATEOAS, возможно, стал очень подходящим вариантом. Но эта концепция не особо прижилась. Возможно, идеи, лежащие в основе HATEOAS, слишком чужды для нашего понимания, а может быть, роль сыграло отсутствие инструментов или стандартов в этой области, или, вероятно, модель просто не работает для созданных нами в итоге систем. Конечно, также возможно, что концепции, лежащие в основе HATEOAS, на самом деле плохо сочетаются с методами создания микросервисов.

Так что в узкоспециализированном использовании эта концепция работает фантастически хорошо, и для синхронной связи по модели «запрос — ответ» между микросервисами она великолепна.

GraphQL

В последние годы язык GraphQL (https://graphql.org) приобрел большую популярность во многом благодаря тому, что он позволяет устройству на стороне клиента определять вызовы, способные избежать выполнения нескольких запросов для получения одной и той же информации. Это может значительно улучшить производительность ограниченных по производительности клиентских устройств, а также поможет избежать необходимости внедрять индивидуальную агрегацию на стороне сервера.

В качестве простого примера представьте мобильное устройство, которому нужно отобразить страницу с обзором последних заказов клиента. Страница должна содержать некоторую информацию о клиенте, а также информацию о его пяти последних заказах. На экране требуется показать всего несколько полей из записи клиента: дату, стоимость и статус доставки каждого заказа. Мобильное устройство может отправлять вызовы двум нижестоящим микросервисам для получения необходимой информации, но для этого понадобится выполнить несколько вызовов, включая извлечение информации, которая на самом деле необязательна. Это может быть расточительно: на такие операции расходуется больше средств тарифного плана мобильного устройства, чем необходимо, и может занять больше времени.

GraphQL позволяет выдавать один запрос, способный извлекать всю необходимую информацию. Чтобы это сработало, нужен микросервис, предоставляющий конечную точку GraphQL клиентскому устройству. Эта точка GraphQL служит входом для всех клиентских запросов и предлагает схему для использования клиентскими устройствами. Такая схема предоставляет типы, доступные клиенту, а также удобный графический конструктор запросов, упрощающий их создание. Уменьшая количество вызовов и объем данных, извлекаемых клиентским устройством, вы можете аккуратно решать некоторые проблемы, возникающие при создании пользовательских интерфейсов с микросервисными архитектурами.

Проблемы

На раннем этапе одной из проблем было отсутствие языковой поддержки спецификации GraphQL, и единственным выбором изначально был JavaScript. В этом плане произошел серьезный рывок, поскольку все основные технологии теперь поддерживают эту спецификацию. На самом деле произошли значительные улучшения в GraphQL и различных реализациях по всем направлениям, что сделало применение GraphQL гораздо менее рискованной перспективой, чем это было несколько лет назад. Тем не менее вам стоит знать об оставшихся проблемах.

Для начала клиентское устройство может выдавать динамически изменяющиеся запросы, и бывают команды, у которых возникают проблемы с запро-

сами GraphQL, вызывающими значительную нагрузку на серверную часть. При сравнении GraphQL с чем-то вроде SQL мы видим аналогичную проблему. Ресурсоемкий SQL-запрос может вызвать значительные проблемы для базы данных и потенциально оказать серьезное влияние на систему в целом. Та же проблема возникает и с GraphQL. Разница в том, что при использовании SQL применяются такие инструменты, как планировщики запросов для БД, которые помогают диагностировать проблемные запросы, в то время как аналогичный недостаток с GraphQL сложнее отследить. Регулирование запросов на стороне сервера — одно из потенциальных, но непростых решений, поскольку выполнение вызова может быть распределено между несколькими микросервисами.

Кэширование оказывается более сложным по сравнению с обычными HTTP-API на основе REST. При использовании API на основе REST можно задать один из множества ответов заголовков, чтобы помочь клиентским устройствам или промежуточным кэшам, таким как сети доставки контента (content delivery networks, CDNs), кэшировать ответы, чтобы их не нужно было запрашивать снова. При использовании GraphQL организовать процесс таким же образом невозможно. Советы по этому вопросу сводятся к тому, чтобы просто связать идентификатор с каждым возвращаемым ресурсом (и помните: запрос GraphQL может содержать несколько ресурсов), а затем заставить клиентское устройство кэшировать запрос по этому идентификатору. Насколько я могу судить, без лишней работы это делает использование CDN или кэширование обратных прокси невероятно сложным.

Хотя я видел некоторые решения данной проблемы, зависящие от конкретной реализации (например, обнаруженные в JavaScript Apollo), кэширование, похоже, было сознательно или бессознательно проигнорировано на этапе первоначальной разработки GraphQL. Если отправляемые вами запросы очень специфичны по своей природе для конкретного пользователя, то отсутствие кэширования на уровне запроса не может стать препятствием для контракта, поскольку коэффициент попадания в кэш, скорее всего, будет низким. Однако мне интересно, означает ли это ограничение, что вы все равно получите гибридное решение для клиентских устройств, где одни (более общие) запросы будут выполняться через обычные HTTP-API на основе REST, а другие — через GraphQL.

Еще одна проблема заключается в том, что, хотя GraphQL теоретически способен обрабатывать процесс записи, он подходит для этого не так хорошо, как для чтения. Это приводит к ситуациям, в которых команды используют GraphQL для чтения, а REST — для записи.

Последний вопрос может показаться совершенно субъективным, но я все же думаю, что его стоит поднять. GraphQL дарит вам ощущение, что вы просто работаете с данными. Это может укрепить мнение о том, что микросервисы, с которыми вы взаимодействуете, представляют собой всего лишь оболочки баз данных. Я видел, как несколько человек сравнивали GraphQL с Odata —

технологией, разработанной как универсальный API для доступа к данным из БД. Идея рассматривать микросервисы просто как оболочку баз данных может иметь негативные последствия. Микросервисы предоставляют функциональные возможности через сетевые интерфейсы. Некоторые из этих функций могут привести к раскрытию данных или потребовать сделать это, но они все равно должны содержать свою собственную внутреннюю логику и поведение. Не стоит думать, что ваши микросервисы — это не более чем API для базы данных, только потому, что вы используете GraphQL. Очень важно, чтобы GraphQL API не был связан с базовыми хранилищами данных ваших микросервисов.

Где использовать

Лучшее место для использования GraphQL — по периметру системы, где предоставляется функциональность внешним клиентам. Эти клиенты, как правило, представляют собой графические интерфейсы, и такое решение подходит для мобильных устройств, учитывая их ограниченные возможности по передаче данных конечному пользователю и характер мобильных сетей. Но GraphQL также используется для внешних API, и сервис GitHub стал одним из первых последователей GraphQL. Если у вас есть внешний API, который часто требует от внешних клиентов совершать множество вызовов для получения необходимой им информации, то GraphQL поможет сделать API намного более эффективным и удобным.

По сути, GraphQL — это механизм агрегации и фильтрации вызовов, поэтому в контексте микросервисной архитектуры он будет использоваться для агрегирования вызовов нескольких нижестоящих микросервисов. Сам по себе он не может служить заменителем общей коммуникации между микросервисами.

Альтернативой использованию GraphQL может стать шаблон «Бэкенд для фронтенда» (Backend for Frontend, BFF), который мы рассмотрим, а также сравним с GraphQL и другими методами агрегации в главе 14.

Брокеры сообщений

Брокеры сообщений — это посредники, часто называемые промежуточным ПО, находящиеся между процессами для управления связью между ними. Они стали популярным вариантом выбора для реализации асинхронной связи, поскольку предлагают множество мощных возможностей.

Как обсуждалось ранее, сообщение — это общее понятие, определяющее то, что отправляет брокер. Сообщение может содержать запрос, ответ или событие. Вместо того чтобы напрямую связываться с другим микросервисом сервис передает брокеру сообщение с информацией о том, как оно должно быть отправлено.

Топики и очереди

Брокеры, как правило, предоставляют либо очереди, либо топики, либо и то и другое. Очереди обычно бывают точечными. Отправитель помещает сообщение в очередь, а потребитель считывает его из этой очереди. В топик-ориентированной системе несколько пользователей могут подписаться на топик, чтобы получить копию этого сообщения.

Потребитель может представлять собой один или несколько микросервисов, обычно моделируемых как группа потребителей. Это может быть полезно, если у вас имеется несколько экземпляров микросервиса и вы хотите, чтобы любой из них мог получить сообщение. На рис. 5.1 показан пример, в котором у сервиса `Обработчик заказов` есть три развернутых экземпляра, входящих в одну группу потребителей. Когда сообщение помещается в очередь, только один член группы получит его. Это означает, что очередь работает как механизм распределения нагрузки. Это пример модели конкурирующих потребителей, мы кратко затронули ее в главе 4.

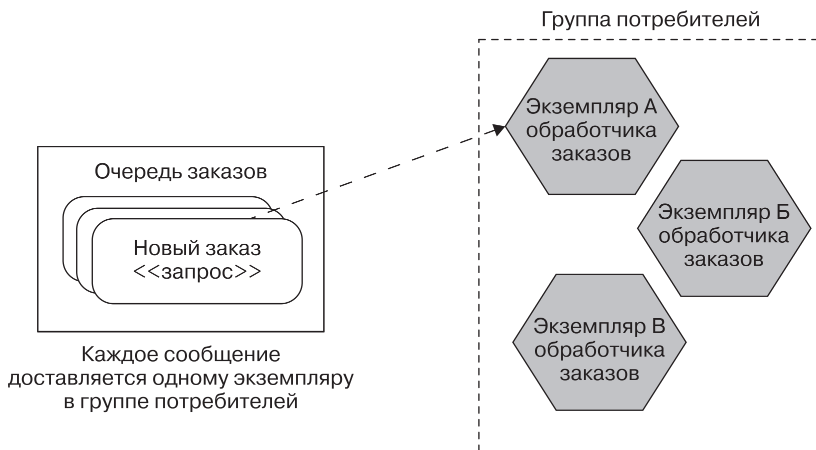


Рис. 5.1. Очередь допускает одну группу потребителей

С помощью топиков можно создать несколько групп потребителей. На рис. 5.2 событие, представляющее оплачиваемый заказ, помещается в топик `Статус заказа`. Копия этого события принимается как микросервисом `Склад`, так и сервисом `Уведомления`, которые находятся в отдельных группах потребителей. Только один экземпляр каждой группы потребителей увидит это событие.

На первый взгляд очередь выглядит просто как топик с одной группой потребителей. Основное различие между ними заключается в том, что, когда сообщение отправляется через очередь, сохраняется информация, кому оно предназначено. В случае с топиком эта информация скрыта от отправителя — он не знает, кто (если таковой блок вообще будет) в конечном счете получит сообщение.

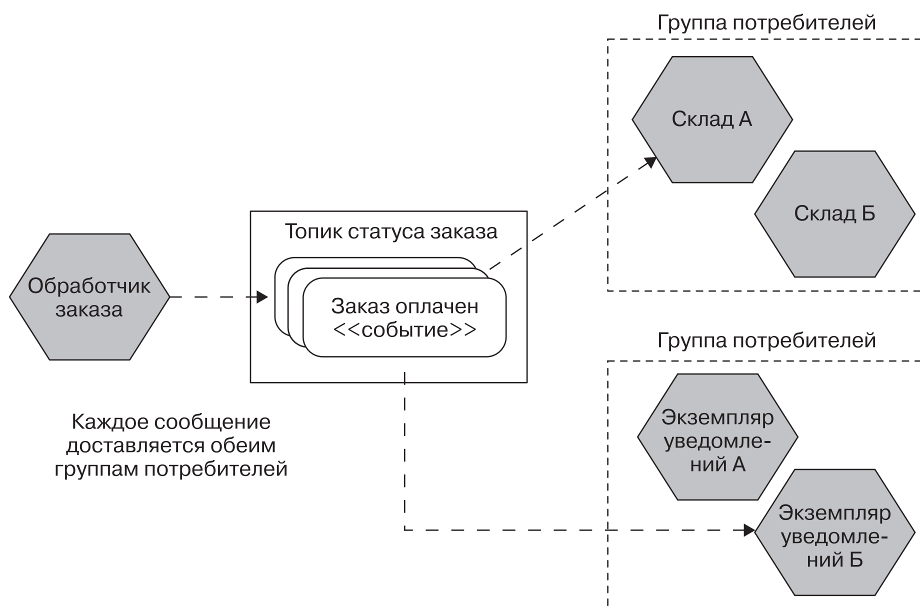


Рис. 5.2. Топики позволяют нескольким подписчикам получать одни и те же сообщения, что полезно для трансляции событий

Топики хорошо подходят для событийного сотрудничества, в то время как очереди больше для коммуникации в стиле «запрос — ответ». Однако это следует рассматривать как общее руководство, а не как строгое правило.

Гарантированная доставка

Так зачем же использовать брокеры? По сути, они предоставляют некоторые возможности, которые могут быть очень полезны для асинхронной связи. Предоставляемые ими свойства различаются, но наиболее интересной особенностью для нас будет гарантированная доставка, которую так или иначе поддерживают все широко используемые брокеры. Гарантированная доставка описывает обязательство брокера обеспечить доставку сообщения.

С точки зрения микросервиса, отправляющего сообщение, это может быть очень полезно. Если нижестоящий адресат недоступен, брокер будет удерживать сообщение до тех пор, пока доставка не осуществится, что уменьшит количество проблем для вышестоящего микросервиса. Сравните этот процесс с синхронным прямым вызовом. Например, с HTTP-запросом: если нижестоящий адресат недоступен, вышестоящему микросервису необходимо решить, что делать с запросом: повторить вызов или отказаться?

Чтобы гарантированная доставка работала, брокер должен обеспечить хранение всех еще не доставленных сообщений в надежном месте до тех пор,

пока они не будут доставлены. Чтобы выполнить это обязательство, брокер обычно работает как своего рода кластерная система, гарантируя, что потеря одного компьютера не приведет к потере сообщения. Как правило, для стабильной работы брокера требуется выполнить большой объем работ, отчасти из-за проблем с управлением программным обеспечением на основе кластеров. Часто обязательство гарантированной доставки может быть нарушено, если брокер настроен неправильно. Например, для RabbitMQ необходимо, чтобы экземпляры в кластере обменивались данными по сетям с относительно низкой задержкой. В противном случае экземпляры могут начать путаться в текущем состоянии обрабатываемых сообщений, что приведет к потере данных. Я не говорю, что RabbitMQ из-за этого плох, у всех брокеров есть ограничения относительно того, как их нужно запускать, чтобы обеспечить гарантированную доставку. Если вы планируете запустить собственный брокер, обязательно изучите документацию.

Стоит также отметить, что понятие гарантированной доставки у каждого конкретного брокера может варьироваться. Опять же, чтение документации — отличная отправная точка в изучении этой особенности.

Доверие

Гарантированная доставка — одно из главных преимуществ брокера. Но для того, чтобы это свойство работало, вам нужно доверять не только людям, которые создали брокер, но и тому, как этот брокер работает. Если вы разработали систему, основанную на предположении, что доставка гарантирована, а это оказывается не так из-за проблемы с базовым брокером, то могут возникнуть серьезные проблемы. Остается надеяться, что вы перекладываете эту работу на ПО, созданное людьми, которые могут решать эту задачу лучше вас. В конечном счете вы должны решить, насколько доверять используемому брокеру.

Другие характеристики

Брокеры полезны не только благодаря гарантированной доставке.

Большинство из них могут гарантировать порядок доставки сообщений, но даже в этом случае объем подобной гарантии ограничен. Например, в Kafka порядок гарантируется только в пределах одного раздела. Если нет уверенности, что сообщения будут получены по порядку, вашему потребителю может потребоваться компенсация, возможно, путем отсрочки обработки сообщений, полученных не по порядку, до тех пор, пока не поступят все пропущенные сообщения.

Некоторые брокеры предоставляют транзакции при записи. Например, Kafka позволяет записывать сообщения в несколько топиков за одну транзакцию. Многие брокеры также могут обеспечивать транзакцию чтения, чем я воспользовался при применении ряда брокеров через API Java Message Service (JMS).

Это полезно, если вы хотите убедиться, что сообщение может быть обработано потребителем, прежде чем удалять его у брокера.

Еще одна, несколько спорная, функция, обещанная некоторыми брокерами, — однократная доставка. Один из самых простых способов обеспечить гарантированную доставку — разрешить повторную отправку сообщения. Это может привести к тому, что потребитель увидит одно и то же сообщение несколько раз (даже если это редкая ситуация). Большинство брокеров сделают все возможное, чтобы уменьшить такую вероятность или скрыть этот факт от потребителя, но некоторые брокеры продвинуты настолько, что гарантируют доставку сообщения только один раз. Это сложная тема. Я общался с некоторыми экспертами, которые утверждают, что гарантировать подобную функцию во всех случаях невозможно. В то же время другие эксперты утверждают, что в принципе это можно реализовать с помощью нескольких простых обходных путей. В любом случае, если выбранный вами брокер утверждает, что способен на однократную доставку, обратите *пристальное* внимание на то, как это реализовано. А еще лучше построить свои потребители таким образом, чтобы они были готовы к получению сообщения более одного раза и могли справиться с этой ситуацией. Очень простой пример: каждому сообщению присваивается идентификатор, который потребитель проверяет при каждом получении сообщения. Если сообщение с таким идентификатором уже было обработано, то новое может быть проигнорировано.

Варианты выбора

Существует множество брокеров сообщений. К популярным примерам относятся RabbitMQ, ActiveMQ и Kafka (который мы вскоре рассмотрим подробнее). Основные поставщики публичных облачных сервисов также предоставляют множество продуктов, выполняющих функции брокера: от управляемых версий этих брокеров, которые вы можете установить в своей собственной инфраструктуре, до индивидуальных реализаций, специфичных для конкретной платформы. Например, в AWS есть Simple Queue Service (SQS), Simple Notification Service (SNS) и Kinesis, предоставляющие различные варианты полностью управляемых брокеров. Фактически SQS был вторым в истории продуктом, выпущенным AWS, который был запущен еще в 2006 году.

Kafka

Kafka стоит выделить в качестве особого брокера, учитывая его популярность в последнее время. Отчасти она объясняется возможностью Kafka перемещать большие объемы данных в рамках реализации конвейеров потоковой обработки. Это может помочь перейти от пакетной обработки к обработке в режиме реального времени.

Отдельного внимания заслуживают некоторые особенности Kafka. Во-первых, этот инструмент рассчитан на очень большие масштабы — он был разработан

в LinkedIn, чтобы заменить несколько существующих кластеров сообщений единой платформой. Kafka позволяет работать с множеством потребителей и производителей. Я общался с одним экспертом из крупной технологической компании, у которого более 50 000 производителей и потребителей работали в одном кластере. Честно говоря, очень немногие организации сталкиваются с проблемами при таком масштабе, но для некоторых компаний возможность легко масштабировать Kafka (условно говоря) может быть очень полезной.

Во-вторых, Kafka — это постоянство сообщений. Обычный брокер сообщений не хранит сообщение, как только последний потребитель получил его. С Kafka сообщения могут храниться в течение настраиваемого периода или вообще вечно. Такой подход дает возможность потребителям повторно отправлять уже обработанные ими сообщения или разрешает вновь развернутым потребителям обрабатывать отправленные ранее сообщения.

Наконец, Kafka внедряет встроенную поддержку потоковой обработки. Вместо того чтобы использовать Kafka для отправки сообщений в специальный инструмент обработки потоков, такой как Apache Flink, теперь появилась возможность некоторые задачи выполнять внутри Kafka. Используя потоки KSQL, можно определить SQL-подобные инструкции, которые обрабатывают один или несколько топиков на лету. Это дает что-то похожее на динамически обновляемое материализованное представление базы данных, при этом источником данных будут топики Kafka, а не БД. Такой потенциал открывает некоторые очень интересные возможности для управления данными в распределенных системах. Если вы хотите изучить эти идеи более подробно, рекомендую книгу «Проектирование событийно-ориентированных систем» от Бена Стопфорда (я должен рекомендовать книгу Бена, поскольку я написал к ней предисловие!). Для более глубокого погружения в Kafka в целом я бы предложил книгу «Apache Kafka. Потоковая обработка и анализ данных»[1].

Форматы сериализации

Некоторые из рассмотренных нами вариантов технологий, в частности отдельные реализации RPC, позволяют выбирать, как данные сериализуются и десериализуются. Например, при использовании gRPC любые отправленные данные будут преобразованы в формат Protocol buffers. Однако многие технологические решения дают нам большую свободу выбора способа скрытия данных для сетевых вызовов. Выберите Kafka в качестве своего варианта брокера — и сможете отправлять сообщения в различных форматах. Итак, какой формат следует выбрать?

[1] *Шапира Г., Палино Т., Сиварам Р., Петти К.* Apache Kafka. Потоковая обработка и анализ данных. 2-е изд. — Питер, 2023.

Текстовые форматы

Применение стандартных текстовых форматов дает клиентам большую гибкость в том, как они используют ресурсы. REST API чаще всего используют текстовый формат для тела запроса и ответа, даже если теоретически вы способны отправлять двоичные данные по протоколу HTTP. На самом деле именно так работает gRPC — использует HTTP-сервер, но отправляет данные через двоичный Protocol buffers.

JSON занял место XML в качестве предпочтительного формата сериализации текста. Можно указать целый ряд факторов, по которым это произошло, но главная причина заключается в том, что одним из основных потребителей API часто оказывается браузер, где JSON отлично подходит. В качестве еще одного довода в пользу JSON его сторонники ссылаются на относительную компактность и простоту по сравнению с XML. Реальность такова, что между размером полезной нагрузки JSON и XML редко бывает существенная разница, особенно учитывая, что эти полезные нагрузки обычно сжимаются. Также стоит отметить, что простота JSON имеет свою цену — схемы ушли в прошлое (подробнее об этом позже).

Avro — еще один интересный формат сериализации. Он принимает JSON в качестве базовой структуры и использует его для определения формата на основе схемы. Avro приобрел большую популярность в качестве формата для полезной нагрузки сообщений отчасти благодаря своей способности отправлять схему как часть полезной нагрузки, что значительно упрощает поддержку нескольких различных форматов обмена сообщениями.

Однако лично я по-прежнему остаюсь поклонником XML из-за лучших инструментов поддержки. Например, если требуется извлечь только определенные части полезной нагрузки (этот метод мы рассмотрим подробнее в разделе «Обработка изменений между микросервисами» далее), можно использовать XPATH, представляющий собой хорошо понятный стандарт с большим количеством инструментов поддержки, или даже CSS селекторы, которые многим кажутся еще проще. JSON же предоставляет способ адресации JSONPath, который не так широко поддерживается. Мне кажется странным, что люди выбирают JSON, потому что он приятный и легкий, а потом пытаются внедрить в него элементы управления гипермедиа, которые уже существуют в XML. Однако я признаю, что JSON — это выбор большинства.

Двоичные форматы

В то время как у текстовых форматов есть такие преимущества, как простота их чтения людьми и обеспечение высокой совместимости с различными инструментами и технологиями, мир протоколов двоичной сериализации — это место, где хочется оказаться, если вы начинаете задумываться о размере полезной нагрузки или об эффективности записи и чтения полезных нагрузок. Существующие

буферы протокола часто используются вне рамок gRPC — они, вероятно, представляют собой самый популярный формат двоичной сериализации для связи микросервисов.

Однако это большая область, и с учетом различных требований был разработан ряд других форматов. На ум приходят Simple Binary Encoding (https://oreil.ly/p8UbH), Cap'n Proto (https://capnproto.org) и FlatBuffers (https://oreil.ly/VdqVB). Для каждого из этих форматов существует множество целевых показателей, подчеркивающих их преимущества по сравнению с Protocol buffers, JSON или другими форматами. Проблема этих показателей заключается в том, что они не обязательно учитывают реальный режим использования. Если вы хотите выжать последние несколько байтов из вашего формата сериализации или сократить время, затрачиваемое на чтение или запись этих полезных нагрузок, на микросекунды, я настоятельно рекомендую вам провести собственное сравнение этих различных форматов. По моему опыту, подавляющему большинству систем редко приходится заботиться о такой оптимизации, поскольку они часто могут добиться желаемых улучшений, отправляя меньше данных или вообще не совершая вызовов. Однако при создании распределенной системы со сверхнизким откликом убедитесь, что вы готовы с головой окунуться в мир двоичных форматов сериализации.

Схемы

Снова и снова возникает дискуссия о том, следует ли нам использовать схемы для определения того, что предоставляют конечные точки и что они принимают. Схемы могут быть самых разных типов, и выбор формата сериализации обычно определяет, к какой технологии схемы прибегнуть. При работе с необработанным XML необходимо применять определение схемы XML (XSD); при работе с необработанным JSON — схему JSON. Некоторые из вариантов затронутых нами технологий (в частности, значительное подмножество версий RPC) требуют использования явных схем, поэтому, если вы выбрали такие технологии, вам придется применять схемы. SOAP работает с WSDL, в то время как gRPC требует спецификации Protocol buffers. Для других рассмотренных вариантов технологий использование схем необязательно, и здесь все становится еще интереснее.

Я выступаю за наличие явных схем для конечных точек микросервиса по двум ключевым причинам. Первая — это явное представление того, что предоставляет и принимает конечная точка микросервиса. Это облегчает жизнь как разработчикам микросервиса, так и его пользователям. Схемы не заменят хорошую документацию, но они определенно помогут сократить ее объем.

Вторая — это как явные схемы помогают обнаружить случайные поломки конечных точек микросервиса. Вскоре мы рассмотрим, как обрабатывать изменения между микросервисами, но сначала стоит изучить различные типы сбоев и роль, которую могут играть схемы.

Структурные и семантические разрывы контрактов

Грубо говоря, мы можем разделить нарушения контракта на две категории: *структурные* и *семантические* разрывы. Структурный разрыв — это ситуация, в которой структура конечной точки перестраивается таким образом, что потребитель становится недоступен. Это говорит об удалении полей или методов или о добавлении новых обязательных полей. При семантическом разрыве структура конечной точки микросервиса остается неизменной, но поведение изменяется, нарушая ожидания потребителей.

Возьмем простой пример. У нас есть микросервис `Сложные вычисления`, предоставляющий метод `calculate` (вычисления) в своей конечной точке. Этот метод принимает два целых числа, и оба они являются обязательными полями. Если изменить сервис `Сложные вычисления` так, что метод `calculate` станет принимать только одно целое число, тогда контракты с потребителями будут разорваны — они отправят запрос на два целых числа, которые микросервис `Сложные вычисления` отклонит. Это пример структурных изменений, и в целом их легче заметить.

А семантические изменения более проблематичны. В этом случае меняется поведение конечной точки, а не ее структура. Возвращаясь к нашему методу `calculate`, представьте, что в первой версии два предоставленных целых числа складываются, результаты возвращаются. Все идет нормально. Теперь внесем в сервис `Сложные вычисления` такие изменения, чтобы метод `calculate` перемножал целые числа и возвращал результат. Семантика метода `calculate` изменилась, что может нарушить ожидания потребителей.

Стоит ли использовать схемы

Используя схемы и сравнивая различные их версии, мы можем выявить структурные нарушения. Чтобы обнаружить семантические разрывы, необходимо прибегнуть к тестированию. Если у вас нет схем или они есть, но вы решили не сравнивать их изменения на предмет совместимости, то бремя выявления структурных разрывов до выпуска продукта также ложится на тестирование. Возможно, ситуация в чем-то похожа на статическую и динамическую типизацию в языках программирования. При использовании языка со статической типизацией типы фиксируются во время компиляции: если ваш код делает что-то с экземпляром недопустимого типа (например, вызывает несуществующий метод), компилятор способен обнаружить эту ошибку. Это позволит сосредоточить усилия по тестированию на других проблемах. Однако при использовании динамически типизированного языка некоторые из тестов должны будут выявлять ошибки, которые компилятор обнаруживает для статически типизированных языков.

Теперь я довольно спокойно отношусь к статическим и динамически типизированным языкам, и обнаружил, что работаю очень продуктивно (условно

говоря) и с теми, и с другими. Конечно, динамически типизированные языки имеют некоторые существенные преимущества, которые для многих людей оправдывают отказ от безопасности во время компиляции. Однако, с моей точки зрения, если мы вернемся к обсуждению взаимодействия микросервисов, я не заметил, что существует такой же сбалансированный компромисс, когда сравниваются схемная и неструктурированная коммуникации. Проще говоря, я думаю, что наличие явной схемы более чем компенсирует любую предполагаемую выгоду от взаимодействия без схем.

Вопрос на самом деле не в том, есть ли у вас схема или нет, а в том, *явная* ли она. Если данные потребляются из API без схемы, все еще есть ожидания относительно того, какие данные должны быть там и как они структурированы. Ваш обрабатывающий данные код будет написан с учетом ряда предположений относительно того, как эти данные структурированы. В таком случае я бы сказал, что у вас на самом деле есть схема, но она скорее неявная, чем явная[1]. Во многом мое стремление к наличию явной схемы обусловлено тем, что я считаю важным как можно более явно отображать, что микросервис предоставляет (или не предоставляет).

Основным аргументом в пользу конечных точек без схем, по-видимому, будет то, что схемы требуют больше трудозатрат и не дают достаточной выгоды. Это, по моему скромному мнению, недостаток воображения и хорошего инструментария, помогающего схемам представлять большую выгоду, когда дело доходит до их использования для обнаружения структурных разрывов.

В конечном счете многое из того, что предоставляют схемы, будет явным представлением части структурного контракта между клиентом и сервером. Они помогают сделать данные явными и могут значительно облегчить общение между командами, а также работать в качестве подстраховки. В ситуациях снижения стоимости изменений, например, когда и клиент, и сервер принадлежат одной команде, — я более спокойно отношусь к отсутствию схем.

Обработка изменений между микросервисами

Вероятно, самый распространенный вопрос, который мне задают о микросервисах, после «Насколько большими они должны быть?» — это «Как вы справляетесь с управлением версиями?». На самом деле это вопрос больше о том, как обрабатывать изменения в контрактах между микросервисами, чем о том, какую схему нумерации использовать.

Тему обработки изменений можно разделить на две подтемы. Вскоре мы рассмотрим, что произойдет, если вам потребуется внести критические изменения. Но в первую очередь посмотрим, что можно сделать, чтобы их избежать.

[1] Мартин Фаулер исследует это более подробно в контексте хранения данных без схемы (https://oreil.ly/Ew8Jq).

Избегание критических изменений

Если вы хотите избежать внесения критических изменений, стоит изучить несколько ключевых идей, многие из которых мы уже затрагивали в начале главы. Если вы сможете реализовать их, в будущем вам станет намного проще перестраивать микросервисы независимо друг от друга.

Наращивание изменений

Добавляйте новые элементы в интерфейс микросервиса; не удаляйте старые элементы.

Устойчивое считывание

При использовании интерфейса микросервиса будьте гибки в ожиданиях.

Правильные технологии

Выберите технологию, упрощающую внесение обратно совместимых изменений в интерфейс.

Явный интерфейс

Четко укажите, что предоставляет микросервис. Это упрощает задачу клиенту и помогает обслуживающим микросервис людям понять, что можно свободно изменить.

Своевременно выявляйте случайные критические изменения

Подключите механизмы для отслеживания модификаций интерфейса, нарушающих работу потребителей, в эксплуатации до того, как эти изменения будут развернуты.

Эти идеи усиливают друг друга, и многие из них основаны на той ключевой концепции скрытия информации, которую мы не раз обсуждали. Давайте рассмотрим каждую идею по очереди.

Наращивание изменений

Вероятно, проще всего начать с добавления только новых элементов в контракт микросервиса и ничего не удалять. Рассмотрим пример добавления нового поля в полезную нагрузку. Если клиент в какой-то мере терпим к таким изменениям, это не должно оказывать существенного влияния. Например, добавление нового поля `dateOfBirth` (дата рождения) в карточку клиента должно быть в порядке вещей.

Устойчивое считывание

То, как реализован потребитель микросервиса, может многое дать для облегчения внесения обратно совместимых изменений. В частности, желательно избежать слишком жесткой привязки клиентского кода к интерфейсу микро-

сервиса. Давайте рассмотрим микросервис `Электронная почта`, задача которого — время от времени отправлять электронные письма нашим клиентам. Ему необходимо отправить электронное письмо с сообщением «Заказ отправлен» покупателю с идентификатором 1234. Микросервис находит клиента с этим идентификатором и возвращает что-то вроде ответа, показанного в примере 5.3.

Пример 5.3. Пример ответа от сервиса Customer

```
<customer>
  <firstname>Sam</firstname>
  <lastname>Newman</lastname>
  <email>sam@magpiebrain.com</email>
  <telephoneNumber>555-1234-5678</telephoneNumber>
</customer>
```

Теперь, чтобы отправить электронное письмо, микросервису `Электронная почта` требуются только поля `firstname`, `lastname` и `email`. Нам не нужно знать значение поля `telephoneNumber`. Достаточно взять нужные поля и проигнорировать остальные. Некоторые технологии привязки, особенно те, которые используются в языках со строгой типизацией, могут пытаться привязать *все* поля независимо от того, хочет этого потребитель или нет. Что произойдет, если мы поймем, что никто не использует поле `telephoneNumber`, и удалим его? Потребители будут понапрасну сбоить.

А что, если бы мы захотели реструктурировать объект `Customer` для поддержки более подробной информации, возможно добавив некоторую дополнительную структуру, как в примере 5.4? Требующиеся сервису `Электронная почта` данные все еще там, с теми же именами, но если наш код делает очень явные предположения относительно того, где будут храниться поля `firstname` и `lastname`, он может снова сломаться. В этом случае можно использовать язык XPath для извлечения интересующих нас полей, что позволит нам не задумываться об их местонахождении. Этот шаблон — реализация считывателя, способного игнорировать не касающиеся нас изменения. Мартин Фаулер называет его tolerant reader (https://oreil.ly/G65yf).

Пример 5.4. Реструктурированный ресурс Customer: все данные по-прежнему в нем, но могут ли потребители найти их?

```
<customer>
  <naming>
    <firstname>Sam</firstname>
    <lastname>Newman</lastname>
    <nickname>Magpiebrain</nickname>
    <fullname>Sam "Magpiebrain" Newman</fullname>
  </naming>
  <email>sam@magpiebrain.com</email>
</customer>
```

Пример клиента, пытающегося быть максимально гибким в использовании сервиса, демонстрирует закон Постела (https://oreil.ly/GVqeI), иначе известный как *принцип надежности*, в котором говорится: «Относись консервативно к тому, что отсылаешь, и либерально к тому, что принимаешь». Эта мудрость родилась в условиях взаимодействия устройств по сетям, от которых можно было ожидать чего угодно. В контексте взаимодействий на основе микросервисов это заставляет нас пытаться структурировать клиентский код так, чтобы он был терпим к изменениям полезных нагрузок.

Правильные технологии

Как мы уже выяснили, отдельные технологии могут быть более уязвимыми, когда речь заходит об изменении нами интерфейсов (я уже упоминал о своих личных разочарованиях в Java RMI). С другой стороны, некоторые реализации интеграции стараются максимально упростить внесение изменений, не нарушая работу клиентов. Одна из простых технологий — Protocol buffers, формат сериализации, используемый как часть gRPC, поддерживает понятие номера поля. Каждая запись в формате Protocol buffers должна определять номер поля, который ищет клиентский код. Если добавляются новые поля, клиенту все равно. Avro позволяет отправлять схему вместе с полезной нагрузкой, что дает возможность клиентам потенциально интерпретировать полезную нагрузку так же, как при использовании динамических типов.

Одна из самых сложных технологий — REST HATEOAS, в значительной степени сводится к тому, чтобы позволить клиентам использовать конечные точки REST, даже когда эти точки изменяются, применяя ранее рассмотренные гипермедиассылки. Конечно, это требует от вас полного понимания HATEOAS.

Явный интерфейс

Я *большой* поклонник микросервисов, раскрывающих явную схему, которая указывает, что делают его конечные точки. Наличие явной схемы дает понимание потребителям, чего они могут ожидать, но также позволяет разработчику, работающему над микросервисом, определить, какие вещи должны оставаться нетронутыми, чтобы вы гарантированно не нарушили деятельность потребителей. Другими словами, явная схема помогает сделать границы скрытия информации более явными — то, что открыто в схеме, по определению не скрыто.

Наличие явной схемы для RPC давно установлено и фактически является требованием для многих реализаций RPC. С другой стороны, при работе со стилем REST концепция схемы рассматривается как необязательная до такой степени, что явные схемы для конечных точек REST исчезающе редки. Ситуация меняется, поскольку такие вещи, как вышеупомянутая спецификация OpenAPI, набирают обороты, как и спецификация JSON Schema.

Асинхронные протоколы обмена сообщениями испытывают больше трудностей в данной области. У вас может быть достаточно простая схема для полезной нагрузки сообщения, и на самом деле это та область, в которой часто используется Avro. Однако наличие явного интерфейса требует большего. Если мы рассмотрим запускающий события микросервис, какие события он раскрывает? В настоящее время предпринимается несколько попыток создания явных событийных схем для конечных точек. Одна из них — AsyncAPI (https://www.asyncapi.com), собравшая ряд известных пользователей, но набирающей наибольшую популярность будет CloudEvents (https://cloudevents.io) — спецификация, которая поддерживается фондом Cloud Native Computing Foundation (CNCF). Продукт сетки событий Azure поддерживает CloudEvents, что говорит о распространенности этого формата у различных поставщиков, а также должно способствовать функциональной совместимости. Будет любопытно понаблюдать, как все изменится в ближайшие несколько лет и к чему приведет.

СЕМАНТИЧЕСКОЕ УПРАВЛЕНИЕ ВЕРСИЯМИ

Разве не было бы здорово, если бы клиент мог просто посмотреть на номер версии сервиса и узнать, получится ли к нему подключиться? Семантическое управление версиями (http://semver.org) — это спецификация, позволяющая реализовать именно такую модель взаимодействия. При семантическом управлении версиями каждый номер версии записывается в виде MAJOR.MINOR.PATCH, где MAJOR означает, что были внесены изменения, несовместимые с обратной связью, MINOR — была добавлена новая функциональность, которая должна быть обратно совместима, а PATCH — исправлены ошибки.

Чтобы увидеть, насколько полезным может быть семантическое управление версиями, давайте рассмотрим простой пример. Наше приложение службы поддержки создано для работы с версией 1.2.0 сервиса `Покупатель`. При добавлении новой функции версия сервиса `Покупатель` изменится на 1.3.0, а наше приложение не заметит никаких изменений в поведении и не должно ожидать внесения каких-либо модификаций. Однако нельзя гарантировать, что сохранится возможность работать с версией 1.1.0 клиентского сервиса, поскольку мы уже будем полагаться на функциональность, добавленную в версии 1.2.0. Также следует ожидать, что придется обновить приложение, если выйдет новая версия 2.0.0 сервиса `Покупатель`.

Вы можете создать семантическую версию для сервиса или даже для отдельной конечной точки в сервисе при совместном использовании, как описано в следующем разделе.

Эта схема управления версиями позволяет упаковать много информации и ожиданий всего в три поля. Полная спецификация очень простыми словами описывает ожидания клиентов от изменений этих цифр и может упростить процесс информирования о том, должны ли изменения повлиять на потребителей. К сожалению, настоящий подход недостаточно часто использовался в распределенных системах, чтобы оценить его эффективность в данном контексте, — это не меняется со времен выхода первого издания книги.

Своевременно выявляйте случайные критические изменения

Крайне важно как можно скорее обнаружить модификации, которые приводят к сбоям у потребителей, потому что, даже если мы выберем наилучшую из возможных технологий, невинное преобразование микросервиса может привести к печальным последствиям. Как уже упоминалось, использование схем поможет выявить структурные изменения при условии, что мы используем какой-либо инструмент для сравнения версий схем. Существует широкий спектр инструментов, позволяющих сделать это для различных типов схем. Для Protocol buffers есть Protolock (https://oreil.ly/wwxBx), для JSON Schema — json-schema-diff-validator (https://oreil.ly/COSIr) и openapi-diff для спецификаций OpenAPI[1]. И похоже, что в этой области постоянно появляются новые инструменты. Однако цель вашего поиска — это нечто, что не просто сообщит о различиях между двумя схемами, но и обеспечит успешное выполнение или неудачное завершение работы в зависимости от совместимости. Это позволит вам отменить сборку CI, если будут обнаружены несовместимые схемы, гарантируя, что ваш микросервис не будет развернут.

Реестр Confluent Schema Registry (https://oreil.ly/qcggd) с открытым исходным кодом поддерживает JSON Schema, Avro и Protocol buffers, а также способен сравнивать недавно загруженные версии для обеспечения обратной совместимости. Несмотря на то что он был создан как часть экосистемы, в которой брокер Kafka используется и необходим для работы, ничто не мешает вам применять его для хранения и проверки схем, используемых для связи, не основанной на Kafka.

Инструменты сравнения схем помогут выявить структурные разрывы, но как насчет семантических? А если вы вообще не используете схемы? Тогда обратимся к тестированию. Эту тему мы рассмотрим более подробно в подразделе «Контрактное тестирование» в главе 9. Но я хотел бы остановиться на контрактном тестировании, которое явно помогает в этой области. Pact является отличным примером инструмента, специально предназначенного для решения этой проблемы. Просто помните, что, если у вас нет схем, вашим тестировщикам придется проделать больше работы, чтобы уловить критические изменения.

Если вы поддерживаете несколько разных клиентских библиотек, запуск тестов с использованием каждой такой библиотеки — это еще один подходящий способ. Как только вы поймете, что вот-вот нарушите работу потребителя, у вас

---

[1] Обратите внимание, что на самом деле в этой области есть три разных инструмента с одинаковыми именами! Инструмент openapi-diff по адресу https://github.com/Azure/openapi-diff, кажется, ближе всего подходит к определению инструмента, который на самом деле пропускает или не пропускает совместимость.

есть выбор: либо попытаться избежать поломки вообще, либо принять ее и начать консультировать людей, отвечающих за сервисы-потребители.

Управление критическими изменениями

Итак, вы сделали все возможное, чтобы модификации, вносимые вами в интерфейс микросервиса, были обратно совместимы, однако теперь вам нужно внести критическое изменение. Что можно сделать в такой ситуации? У вас есть три основных варианта.

Поэтапное развертывание

Требуется, чтобы микросервис, предоставляющий доступ к интерфейсу, и все потребители этого интерфейса изменялись одновременно.

Сосуществование несовместимых версий микросервиса

Запускайте старую и новую версии микросервиса бок о бок.

Эмуляция старого интерфейса

Пусть ваш микросервис предоставляет новый интерфейс, а также поддерживает эмуляцию старого.

Поэтапное развертывание

Конечно, поэтапное развертывание идет вразрез с возможностью независимого развертывания. Если мы хотим получить возможность независимо развернуть новую версию нашего микросервиса с критическим изменением его интерфейса, нам нужно дать пользователям время для перехода на новый интерфейс. Это подводит нас к следующим двум вариантам, которые стоит рассмотреть.

Сосуществование несовместимых версий микросервиса

Одним из популярных решений для управления версиями является одновременное использование разных версий сервиса, при этом старые потребители направляют свой трафик на предшествующую версию, а новые видят актуальную, как показано на рис. 5.3. Данный подход иногда используется Netflix в ситуациях, когда стоимость модификации старых потребителей слишком высока, особенно в редких случаях, когда устаревшие устройства все еще привязаны к более старым версиям API. Я не являюсь поклонником этой идеи и понимаю, почему Netflix редко ее использует. Во-первых, если мне нужно исправить внутреннюю ошибку в моем сервисе, теперь я должен исправить и развернуть два разных набора сервисов. Это, вероятно, означало бы, что мне придется разветвлять кодовую базу сервиса, а это всегда влечет за

собой проблемы. Во-вторых, получается, что в системе необходимы интеллектуальные блоки, чтобы направлять потребителей к нужному микросервису. Такое поведение неизбежно оказывается где-то в промежуточном ПО или в куче скриптов `nginx`, что затрудняет анализ поведения системы. Наконец, рассмотрим любое постоянное состояние, которым может управлять наш сервис. Потребители, созданные любой версией сервиса, должны храниться и отображаться для всех сервисов, независимо от того, какая версия использовалась для создания данных изначально. Это может стать дополнительным источником сложности.

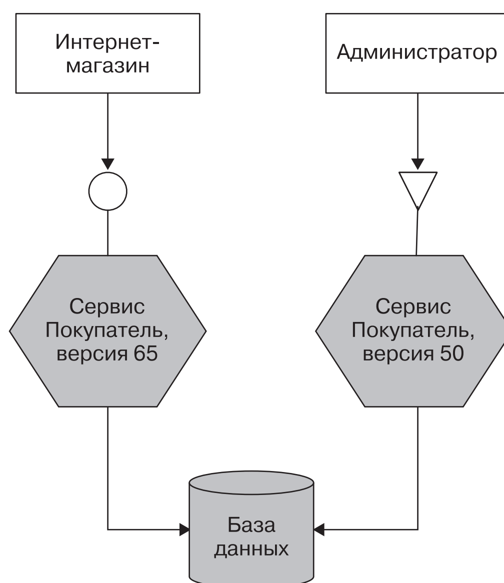


Рис. 5.3. Запуск нескольких версий одного и того же сервиса для поддержки старых конечных точек

Сосуществование параллельных версий сервиса в течение короткого периода времени может иметь смысл, особенно когда вы делаете что-то вроде канареечного релиза (мы подробнее обсудим этот шаблон в подразделе «Переходим к поэтапной доставке» в главе 8). В таких ситуациях можно организовать сосуществование различных версий на протяжении всего нескольких минут или, возможно, часов, и, как правило, у нас одновременно будут присутствовать только две версии сервиса. Чем больше времени вам потребуется, чтобы пользователи обновились до актуальной версии, тем больше вы должны стремиться к сосуществованию разных конечных точек в одном и том же микросервисе, а не к сосуществованию совершенно разных версий. Я не уверен, что эта работа имеет смысл для среднего проекта.

Эмулируйте старый интерфейс

Если мы сделали все возможное, чтобы избежать внесения критических изменений в интерфейс, наша следующая задача — ограничить влияние этих преобразований. Чего мы хотим избежать, так это принуждения потребителей к обновлению одновременно с нами, поскольку всегда необходимо сохранять возможность выпуска микросервисов независимо друг от друга. Один из подходов, который я успешно использовал для решения данной проблемы, заключается в совместном использовании как старого, так и нового интерфейсов в одном и том же запущенном сервисе. Поэтому, если требуется выпустить критическое изменение, мы развертываем новую версию сервиса, которая предоставляет как старую, так и новую версии конечной точки.

Это позволяет максимально быстро запустить новый микросервис вместе с новым интерфейсом, давая при этом потребителям время на переход. Как только все потребители перестают использовать старую конечную точку, можно удалить ее вместе с любым связанным кодом, как показано на рис. 5.4.

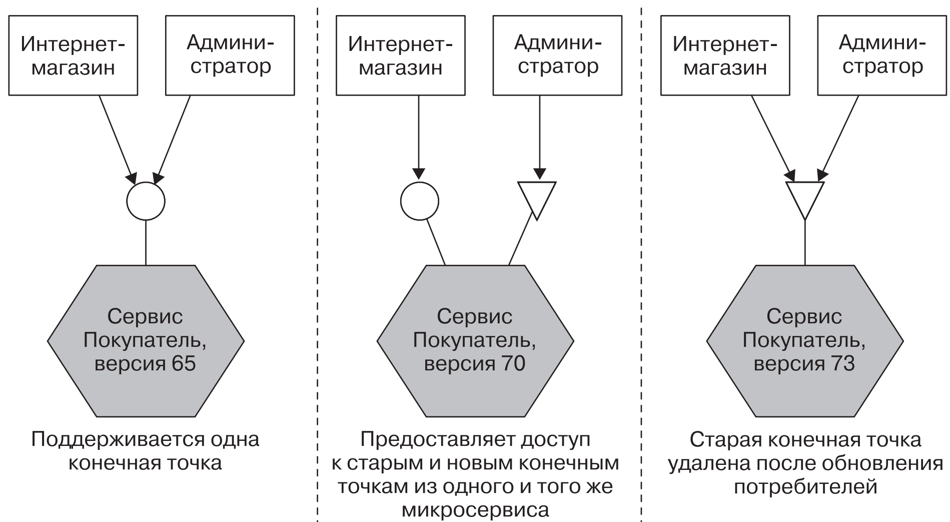


Рис. 5.4. Один микросервис, эмулирующий старую конечную точку и предоставляющий доступ к новой обратно несовместимой конечной точке

Когда я в последний раз использовал данный подход, мы немного запутались из-за количества потребителей и внесенных критических изменений. Это означало, что у нас фактически сосуществовало три разные версии конечной точки. Хотя я бы не рекомендовал так делать! Хранение всего кода и связанного с ним тестирования, необходимого для обеспечения корректной работы, было слишком обременительным. Чтобы сделать эту конструкцию более управляемой,

мы преобразовали все запросы к конечной точке версии 1 в запрос версии 2, а затем запросы версии 2 в запросы к конечной точке версии 3. Это означало, что мы могли четко определить, какой код будет удален, когда старые конечные точки исчезнут.

По сути, это пример паттерна расширения и сжатия, позволяющего поэтапно вносить критические изменения. Мы *расширяем* предлагаемые возможности, поддерживая как старые, так и новые способы выполнения чего-либо. Как только старые потребители начинают работать с новой версией, мы *сжимаем* наш API, удаляя неактуальную функциональность.

Если вы собираетесь реализовать сосуществование конечных точек, вам потребуется способ маршрутизировать запросы вызывающих абонентов. Я видел, что для систем, использующих HTTP, это делается как с номерами версий в заголовках запросов, так и в самом URI — например, */v1/customer/* или */v2/customer/*. Я теряюсь в догадках, какой подход имеет наибольший смысл. С одной стороны, мне нравится, что URI непрозрачны, чтобы отбить у клиентов охоту жестко кодировать шаблоны URI, но, с другой стороны, такой способ делает многие вещи очень очевидными и может упростить маршрутизацию запросов.

Для RPC все немного сложнее. Я справился с этой задачей при помощи Protocol buffers, поместив свои методы в разные пространства имен, например `v1.createCustomer` и `v2.createCustomer`. Однако при попытке поддерживать разные версии одних и тех же типов, отправляемых по сети, данный подход может стать очень неприятным.

Какой подход предпочтителен

В ситуациях, когда одна и та же команда управляет как микросервисом, так и всеми потребителями, я достаточно спокойно отношусь к поэтапному релизу в ограниченных условиях. Если предположить, что это разовый случай, то применение такого подхода, когда воздействие ограничено одной командой, может быть оправданно. Однако существует опасность того, что однократное действие станет обычным делом и исчезнет возможность независимого развертывания. Слишком частое использование поэтапного развертывания приведет к тому, что в скором времени вы получите распределенный монолит.

Сосуществование разных версий одного и того же микросервиса может быть проблематичным. Я бы рассматривал такой вариант только в ситуациях, когда планируется запускать версии микросервиса бок о бок в течение короткого периода времени. Когда вам нужно дать потребителям время на обновление, речь может идти о нескольких неделях. В других ситуациях, когда возможно сосуществование версий микросервисов, в рамках сине-зеленого развертывания или канареечного релиза, продолжительность одновременной работы будет гораздо меньше, что компенсирует недостатки этого подхода.

В целом я предпочитаю использовать эмуляцию старых конечных точек везде, где это возможно. На мой взгляд, с проблемами реализации эмуляции справиться гораздо проще, чем с проблемами сосуществования версий микросервисов.

Общественный договор

Выбор подхода во многом зависит от ожиданий потребителей относительно того, как будут внесены изменения. Сохранение старого интерфейса может обойтись недешево, и в идеале стоило бы отключить его и удалить связанный с ним код и инфраструктуру как можно скорее. С другой стороны, хотелось бы дать потребителям как можно больше времени для внесения изменений. И помните, что во многих случаях вносимые вами обратно несовместимые изменения часто являются тем, о чем просили потребители и/или что в конечном счете принесет им пользу. Конечно, существует баланс между потребностями разработчиков микросервисов и потребителей, и это необходимо учитывать.

Я заметил, что во многих ситуациях вопрос о том, как будут осуществляться эти преобразования, никогда не обсуждался, что приводило к разного рода проблемам. Как и в случае со схемами, наличие некоторой степени ясности того, как будут вноситься обратно несовместимые изменения, может значительно упростить ситуацию.

Вам не обязательно нужны кипы бумаг и масштабные совещания, чтобы достичь соглашения о том, как будут обрабатываться изменения. Но если вы не собираетесь идти по пути поэтапных релизов, я бы предположил, что и владелец, и потребитель микросервиса должны четко понимать несколько вещей.

- Как вы поднимете вопрос о необходимости изменения интерфейса?
- Как потребители и разработчики микросервисов будут взаимодействовать, чтобы договориться о том, как будут выглядеть изменения?
- Кто должен выполнять работу по обновлению потребителей?
- Когда изменение будет согласовано, сколько времени понадобится потребителям для перехода на новый интерфейс, прежде чем его удалять?

Помните, что одним из секретов эффективной микросервисной архитектуры является подход, ориентированный на потребителя. Ваши микросервисы существуют для того, чтобы их вызывали потребители, нужды которых имеют первостепенное значение, и, если ваш метод модификации микросервиса может привести к проблемам, это необходимо учитывать.

В некоторых ситуациях, конечно, может оказаться невозможным изменить потребителей. Я слышал, что у компании Netflix были трудности (по крайней мере, раньше) со старыми телевизионными приставками, использующими неактуальные версии API Netflix. Такие ТВ-приставки нельзя обновить, поэтому предыдущие конечные точки должны оставаться доступными до тех пор, пока количество подобных приставок не упадет до уровня, при котором поддержку

можно будет отключить. Решения о том, чтобы лишить старых потребителей доступа к конечным точкам, иногда могут иметь под собой финансовое обоснование — сколько денег вам стоит поддержка старого интерфейса в сравнении с тем, сколько денег вы зарабатываете на этих потребителях.

Отслеживание использования

Даже если вы объявите о дате прекращения поддержки старого интерфейса, можете ли вы быть уверены, что по истечении этого срока потребители перестанут им пользоваться? Убедитесь, что вход в систему доступен для каждой конечной точки, которую предоставляет ваш микросервис, и что у вас есть какой-либо идентификатор клиента, чтобы вы могли связаться с соответствующей командой, если необходимо убедить их отказаться от старого интерфейса. Это может быть что-то вроде просьбы к потребителям указывать свой идентификатор в заголовке `user-agent` при выполнении HTTP-запросов или требования, чтобы все вызовы проходили через какой-то API-шлюз, где клиентам нужны ключи для идентификации.

Крайние меры

Итак, предположим, вы знаете, что клиент все еще использует старый интерфейс, который вы хотите удалить, и медлит с переходом на новую версию; что вы можете с этим поделать? Что ж, первое — это поговорить с ним. Вдруг вы сможете помочь ему осуществить переход. Если все остальное не приносит результата и потребитель по-прежнему не обновляется даже после согласия на это, есть несколько крайних методов, свидетелем которых я был.

В одной крупной технологической компании рассказали, как справились с этой проблемой. Там был установлен очень щедрый период в один год до того, как старые интерфейсы будут выведены из эксплуатации. Я спросил, как они узнают, что потребители все еще используют старые интерфейсы, и в компании ответили, что на самом деле не утруждали себя отслеживанием этой информации. Через год они просто отключили старый интерфейс. В компании посчитали, что если это привело к поломке потребителя, то это была вина команды микросервиса-потребителя — у них был год, чтобы внести изменения, и они этого не сделали. Конечно, такой радикальный подход не сработает во многих случаях. Кроме того, он понижает эффективность. Не зная, использовался ли старый интерфейс, компания лишила себя возможности удалить его до истечения года. Если бы я предложил просто отключить конечную точку через определенный промежуток времени, я бы все равно пытался отследить, на кого это повлияет.

Другая крайняя мера, с которой я столкнулся, была связана с устареванием библиотек, которые теоретически можно было бы использовать для конечных точек микросервиса. В приведенном примере речь шла о старой библиотеке,

которую люди пытались вывести из эксплуатации внутри организации в пользу более новой, лучшей версии. Несмотря на объемную работу по переносу кода для использования свежей библиотеки, некоторые команды все еще тянули время. Решение заключалось в том, чтобы вставить режим ожидания в старую библиотеку. Из-за этого она стала медленнее реагировать на вызовы (с протоколированием, чтобы показать, что происходит). Со временем команда, ответственная за устаревшие блоки системы, просто продолжала увеличивать время отклика, пока в конце концов другие команды не сообразили, что к чему. Очевидно, что необходимо быть абсолютно уверенными, что иные разумные усилия заставить потребителей обновиться исчерпали себя, прежде чем рассматривать что-то подобное!

DRY и опасности повторного использования кода в мире микросервисов

Одна из аббревиатур, которую мы, разработчики, часто слышим, звучит как DRY (Don't repeat yourself — «Не повторяйтесь»). Хотя определение иногда упрощается до попытки избежать дублирования кода, DRY более точно означает, что желательно избежать дублирования системного *поведения и знаний*. В целом это очень разумный совет. Наличие большого количества строк кода, выполняющих одно и то же, делает ваш код громоздким, и, следовательно, его сложнее анализировать. Когда вы хотите изменить поведение, а оно дублируется во многих частях вашей системы, легко забыть внести изменения во всех этих частях, что может привести к ошибкам. Так что использование DRY в качестве мантры в целом имеет смысл.

Идея DRY приводит нас к созданию кода, который можно использовать повторно. Мы превращаем дублированный код в абстракции, а затем вызываем их из нескольких мест. Возможно, мы дойдем до того, что создадим общую библиотеку, которую сможем использовать везде! Однако оказывается, что совместное использование кода в микросервисной среде немного сложнее. Как всегда, стоит рассмотреть несколько вариантов.

Совместное использование кода через библиотеки

Одна вещь, которую необходимо избежать любой ценой, — чрезмерная связанность микросервиса с потребителями, при которой любое небольшое преобразование в самом микросервисе может вызвать ненужные изменения для потребителя. Иногда, однако, использование общего кода может создать эту самую связанность. Например, у одного нашего клиента была библиотека общих доменных объектов, которые представляли основные сущности, используемые в системе. Этой библиотекой пользовались все имеющиеся сервисы. Но, когда

в один из них вносились изменения, всем сервисам необходимо было обновляться. Наша система общалась через очереди сообщений, которые также должны были быть очищены от их теперь *недействительного* содержимого, и ждите проблем, если вы забыли это сделать.

Выход общего кода за границы микросервиса может стать формой связанности. Использование общего кода, например библиотек регистрации, вполне допустимо, поскольку они относятся к внутренним концепциям, невидимым для внешнего мира. На сайте realestate.com.au для создания новых сервисов используется специально разработанный шаблон сервиса. Вместо того чтобы делать этот код общедоступным, компания копирует его в каждый новый сервис, чтобы исключить утечку кода за границы.

Действительно важным моментом в совместном использовании кода через библиотеки является то, что вы не можете обновлять все варианты использования библиотеки одновременно. Хотя несколько микросервисов могут применять одну и ту же библиотеку, обычно они делают это путем упаковки такой библиотеки в развертывание микросервиса. Поэтому, чтобы обновить версию используемой библиотеки, вам потребуется изменить микросервис. Если вы хотите обновлять одну и ту же библиотеку везде в одно и то же время, это может привести к повсеместному развертыванию нескольких различных микросервисов одновременно со всеми вытекающими последствиями.

Поэтому, если вы применяете библиотеки для повторного использования кода через границы микросервиса, вам придется смириться с тем, что одновременно может существовать несколько различных версий одной и той же библиотеки. Конечно, вы можете со временем обновить их до последней версии, но, пока вы принимаете описанную выше проблему, конечно, используйте код повторно с помощью библиотек. Если же вам действительно необходимо обновить код для всех пользователей одновременно, то вам стоит рассмотреть возможность повторного использования кода через специальный микросервис.

Однако существует один конкретный вариант использования, связанный с повторным применением через библиотеки, который стоит изучить подробнее.

Клиентские библиотеки

Я общался с несколькими командами, которые настаивали на том, что формирование клиентских библиотек для сервисов является неотъемлемой частью создания сервисов. По их мнению, это упрощает использование сервиса и позволяет избежать дублирования кода, необходимого для применения самого сервиса.

Проблема, конечно, заключается в том, что, если одни и те же люди создают как серверный, так и клиентский API, существует опасность, что логика, которая должна существовать на сервере, начнет просачиваться в клиентскую часть.

Я сам так делал. Чем больше логики проникает в клиентскую библиотеку, тем больше нарушается связность, и приходится менять несколько клиентов, чтобы внедрить исправления на свой сервер. Также ограничивается выбор технологий, особенно если вы предписываете использовать клиентскую библиотеку.

Для клиентских библиотек мой выбор — это модель Amazon Web Services (AWS). Базовые вызовы веб-сервисов SOAP или REST могут выполняться напрямую, но в конечном счете все используют только один из многих наборов инструментов для разработки программного обеспечения (software development kit, SDK), предоставляющих абстракции поверх базового API. Однако эти SDK написаны более обширным сообществом или людьми внутри AWS, а не теми, кто работает непосредственно над API. Такая степень разделения, по-видимому, работает и позволяет избежать некоторых подводных камней клиентских библиотек. Одна из причин, по которой это работает так хорошо, заключается в том, что клиент отвечает за время обновления.

Netflix, в частности, уделяет особое внимание клиентской библиотеке, но меня беспокоит, что люди рассматривают это исключительно через призму предотвращения дублирования кода. На самом деле клиентские библиотеки, используемые Netflix, в такой же, если не в большей, степени обеспечивают надежность и масштабируемость своих систем. Клиентские библиотеки Netflix обрабатывают обнаружение сервисов, режимы сбоев, ведение логов и другие аспекты, которые на самом деле не относятся к природе самого сервиса. Без этих общих библиотек было бы трудно гарантировать, что каждая часть клиент-серверных коммуникаций работает должным образом в масштабе, подобном масштабу Netflix. Их использование в Netflix, безусловно, упростило запуск и работу и повысило производительность, а также обеспечило хорошее поведение системы. Однако, по словам одного из сотрудников компании, со временем это привело к тому, что связь между клиентом и сервером стала проблематичной.

Если вы рассматриваете подход с использованием клиентских библиотек, важно отделить клиентский код для обработки базового транспортного протокола от кода, связанного с сервисом назначения. Определитесь, будете ли вы настаивать на клиентской библиотеке или разрешите людям, использующим разные стеки технологий, совершать вызовы базового API. И наконец, убедитесь, что клиенты сами решают, когда обновлять свои клиентские библиотеки: нам необходимо обеспечить возможность выпускать сервисы независимо друг от друга!

Обнаружение сервиса

Как только у вас появляется несколько микросервисов, ваше внимание неизбежно переключается на изучение внутреннего устройства. Возможно, вы хотите знать, что запущено в конкретной среде, чтобы понимать, что вам стоит отслеживать. Может быть, это так же просто, как хранить информацию о местонахождении микросервиса `Учетные записи`, чтобы его потребители знали, где его

найти. А может быть, вы просто хотите облегчить разработчикам в вашей организации использование доступных API, чтобы они не изобретали велосипед заново. В целом все эти варианты использования подпадают под понятие *обнаружения сервисов*. И как всегда в случае с микросервисами, в нашем распоряжении есть довольно много различных вариантов решения этой проблемы.

Все рассматриваемые решения состоят из двух частей. Во-первых, они предоставляют некоторый механизм для того, чтобы экземпляр зарегистрировался и сказал: «Я здесь!» Во-вторых, они дают способ найти сервис после его регистрации. Однако обнаружение сервисов становится более сложным, когда мы рассматриваем среду, в которой постоянно уничтожаются и развертываются новые экземпляры сервисов. В идеале хотелось бы, чтобы любой выбранный способ справился с этой задачей.

Рассмотрим некоторые из наиболее распространенных решений для предоставления сервисов и обсудим наши варианты.

Система доменных имен (DNS)

Начнем, как принято, с простого. DNS позволяет связать имя с IP-адресом одной или нескольких машин. Например, мы могли бы решить, что микросервис Учетные записи всегда находится по адресу accounts.musiccorp.net. Тогда мы можем указать на IP-адрес узла, на котором работает этот микросервис, или, возможно, указать на балансировщик нагрузки, который распределяет нагрузку между несколькими экземплярами. Это означает, что нам придется обрабатывать обновление этих записей в рамках развертывания сервиса.

При работе с экземплярами сервиса в разных средах я видел, как хорошо работает шаблон домена на основе соглашений. Например, у нас может быть типовая форма, определенная как <servicename>-<environment>.musiccorp.net, предоставляя нам такие записи, как accounts-uat.musiccorp.net или accounts-dev.musiccorp.net.

Более продвинутый способ работы с различными средами — использовать разные серверы доменных имен для этих сред. Таким образом, я мог бы предположить, что accounts.musiccorp.net — это место, где всегда можно найти микросервис Учетные записи, но он может регулироваться разными хостами в зависимости от того, где выполняется поиск. Если у вас уже есть среды, расположенные в разных сегментах сети, и вам удобно управлять собственными DNS-серверами и записями, это может быть довольно изящным решением. Но такой подход требует слишком большого количества работы, что может оказаться несопоставимо с получаемыми преимуществами.

DNS обладает множеством преимуществ, главным из которых является понятный и широко используемый стандарт, поддерживаемый практически любым технологическим стеком. К сожалению, хотя существует ряд сервисов для управления DNS внутри организации, не многие из них предназначены для среды, в которой мы имеем дело с одноразовыми хостами, что делает обновление записей DNS несколько болезненным. Сервис Route 53 от Amazon довольно

хорошо справляется с этой задачей, но я еще не видел варианта с самостоятельным размещением, который был бы так же хорош, хотя (как мы вскоре обсудим) некоторые специализированные инструменты обнаружения сервисов, такие как Consul, могут нам здесь помочь. Помимо проблем с обновлением записей DNS, сама спецификация DNS может вызвать некоторые проблемы.

У записей DNS для доменных имен есть *время жизни* (time to live, TTL). Это время, в течение которого клиент может считать запись свежей. Когда мы хотим изменить хост, к которому относится доменное имя, мы обновляем эту запись, но необходимо учесть, что клиенты будут придерживаться старого IP-адреса *по крайней мере* на указанный в TTL период. Записи DNS могут кэшироваться в нескольких местах (даже JVM будет кэшировать записи DNS, если ей не запретить), и чем в большем количестве мест они кэшируются, тем более устаревшей может быть запись.

Один из способов обойти эту проблему — сделать так, чтобы запись доменного имени для сервиса указывала на балансировщик нагрузки, который, в свою очередь, указывает на экземпляры сервиса, как показано на рис. 5.5. При развертывании нового экземпляра можно удалить старый из записи балансировщика нагрузки и добавить новый. Некоторые люди используют циклическую обработку DNS, когда сами записи DNS относятся к группе компьютеров. Этот метод довольно сомнителен, поскольку клиент скрыт от базового хоста и не может легко остановить маршрутизацию трафика на один из хостов, если на нем возникнут проблемы.

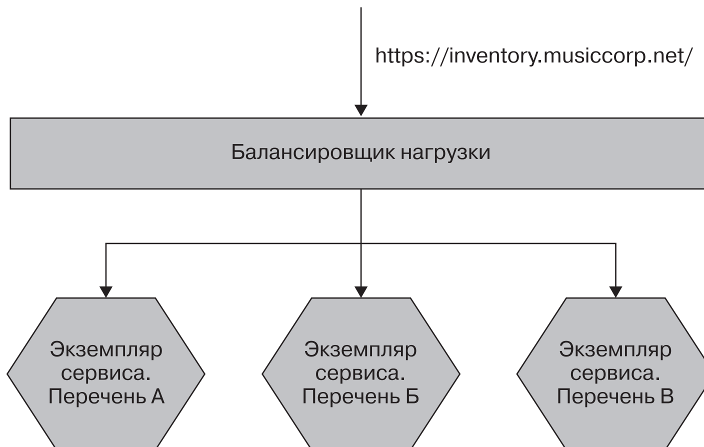


Рис. 5.5. Использование DNS для разрешения балансировщика нагрузки, чтобы избежать устаревших записей DNS

Как упоминалось ранее, система DNS хорошо изучена и широко поддерживается. Но у нее есть несколько недостатков. Я бы посоветовал вам выяснить,

подходит ли вам именно эта система, прежде чем выбирать что-то более сложное. В ситуации, когда у вас есть только одиночные узлы, использование DNS для прямой ссылки на хосты уместно. Но для тех ситуаций, в которых требуется более одного экземпляра хоста, укажите записи DNS для балансировщиков нагрузки, которые могут обрабатывать включение и выключение отдельных хостов по мере необходимости.

Динамические реестры сервисов

Недостатки DNS как способа поиска узлов в высокодинамичной среде привели к появлению ряда альтернативных систем. Большинство из них подразумевают регистрацию сервиса в каком-либо центральном реестре, который, в свою очередь, предоставляет возможность последующего поиска этих сервисов. Часто такие системы делают больше, чем просто обеспечивают регистрацию и обнаружение сервисов. Это обстоятельство может как помочь, так и навредить. Рассмотрим несколько вариантов, чтобы иметь представление о том, что доступно.

ZooKeeper

Сервис ZooKeeper (http://zookeeper.apache.org) первоначально был разработан в рамках проекта Hadoop. У него ошеломляюще много вариантов использования, включая управление конфигурацией, синхронизацию данных между сервисами, выбор лидера, очереди сообщений, и (что полезно для нас) его можно использовать в качестве сервиса именования.

Как и многие подобные типы систем, ZooKeeper полагается на запуск нескольких узлов в кластере для обеспечения различных гарантий. Это означает, что стоит рассчитывать на работу как минимум трех узлов ZooKeeper. Большинство функций ZooKeeper связаны с обеспечением безопасной репликации данных между этими узлами и сохранением целостности данных при сбое узлов.

По сути, ZooKeeper предоставляет иерархическое пространство имен для хранения информации. Клиенты могут вставлять новые узлы в эту иерархию, изменять или запрашивать их. Более того, они способны добавлять наблюдение к узлам, чтобы получать сообщения при их изменении. Это означает, что можно хранить информацию о том, где расположены сервисы в этой структуре и как клиент будет получать информацию при их изменении. ZooKeeper часто используется как общее хранилище конфигурации, поэтому в нем также хранят специфичную для сервиса конфигурацию, что позволяет выполнять такие задачи, как динамическое изменение уровней лога или отключение функций работающей системы.

На самом деле существуют лучшие решения для динамической регистрации сервисов. Они настолько эффективнее, что в настоящее время я бы активно избегал использования ZooKeeper.

Consul

Как и ZooKeeper, Consul (http://www.consul.io) поддерживает управление конфигурацией и обнаружение сервисов. Но он идет дальше, чем ZooKeeper, предоставляя больше поддержки для ключевых вариантов использования. Например, он предоставляет HTTP-интерфейс для обнаружения сервисов. А одна из главных функциональных возможностей Consul заключается в фактическом предоставлении DNS-сервера из коробки. В частности, он может обслуживать SRV-записи, которые дают вам как IP-адрес, так и порт для данного имени. Это означает, что, если часть вашей системы уже использует DNS и способна поддерживать SRV-записи, вы можете просто зайти в Consul и начать использовать его без каких-либо изменений в вашей существующей системе.

В Consul также встроены другие полезные функции, например возможность выполнять проверки работоспособности узлов. Таким образом, Consul вполне способен заменить другие специализированные инструменты мониторинга, но его обычно применяют в качестве источника информации для более комплексной системы мониторинга.

Consul использует HTTP-интерфейс RESTful для всего, от регистрации сервиса до запроса хранилища ключей/значений или вставки проверок работоспособности. Это делает интеграцию с различными технологическими стеками очень простой. У Consul также есть набор хорошо работающих с ним инструментов, еще больше повышающих его полезность. Один из примеров — consul-template (https://oreil.ly/llwVQ), предоставляющий способ обновления текстовых файлов на основе записей в Consul. На первый взгляд это не так интересно, но с помощью consul-template можно изменить значение в Consul — возможно, местоположение микросервиса или значение конфигурации — и получить динамическое обновление конфигурационных файлов по всей системе. Внезапно любая программа, считывающая свою конфигурацию из текстового файла, может динамически обновлять свои текстовые файлы без необходимости что-либо знать о самом Consul. Отличным вариантом использования для этого было бы динамическое добавление или удаление узлов в пул балансировщика нагрузки с помощью программного балансировщика нагрузки, такого как HAProxy.

Еще одним инструментом, который хорошо интегрируется с Consul, является Vault — инструмент управления секретами, к которому мы вернемся в пункте «Секреты» в главе 11. Управление секретами может быть достаточно трудным, но объединение Consul и Vault, безусловно, упростит эту работу.

etcd и Kubernetes

Если вы работаете на платформе, управляющей рабочими нагрузками контейнеров, скорее всего, у вас уже есть механизм обнаружения сервисов. Kubernetes ничем не отличается от аналогов, и частично это связано с etcd (https://etcd.io), хранилищем управления конфигурацией, поставляемым в комплекте с Kubernetes.

Хранилище etcd обладает возможностями, аналогичными Consul, и Kubernetes использует его для управления широким спектром конфигурационной информации.

Мы рассмотрим Kubernetes более подробно в разделе «Kubernetes и оркестрация контейнеров» в главе 8, но в двух словах способ обнаружения сервисов в Kubernetes заключается в том, что вы развертываете контейнер в поде, а затем сервис динамически определяет, какие поды должны быть частью сервиса, путем сопоставления с образцом метаданных, связанных с подом. Это довольно элегантный механизм, который может быть очень мощным. Запросы к сервису затем будут перенаправляться в один из подов, составляющих этот сервис.

Возможности, которые дает Kubernetes из коробки, вполне могут привести к тому, что вы захотите обойтись тем, что поставляется с базовой платформой, отказавшись от использования специализированных инструментов, таких как Consul. Для многих это имеет большой смысл, особенно если более широкая экосистема инструментов вокруг Consul не представляет интереса. Однако если вы работаете в смешанной среде, где рабочие нагрузки выполняются в Kubernetes и на других платформах, то наличие специального инструмента обнаружения сервисов, который можно использовать на обеих платформах, может оказаться правильным решением.

Разверните собственную систему

Один из подходов, который я использовал сам и видел, как его применяют другие, — развертывание собственной системы. В одном проекте мы активно использовали AWS, предлагающий возможность добавлять теги к экземплярам. При запуске экземпляров сервисов я применял теги, чтобы помочь определить, что это был за экземпляр и для чего он использовался. Это позволило связать некоторые метаданные с конкретным хостом, например:

- сервис = учетные записи;
- среда = эксплуатация;
- версия = 154.

Затем я использовал API AWS для запроса всех экземпляров, связанных с данной учетной записью AWS, чтобы найти интересующие меня машины. Здесь AWS сама хранит метаданные, связанные с каждым экземпляром, и предоставляет возможность запросить их. Затем я создал инструменты командной строки для взаимодействия с этими экземплярами и предоставил графические интерфейсы для быстрого просмотра состояния экземпляра. Все это становится довольно простым делом, если вы можете программно собирать информацию об интерфейсах сервисов.

В прошлый раз, когда я делал подобное, мы не заходили так далеко, чтобы сервисы использовали API AWS для поиска своих зависимостей от других сер-

висов, но ничто не мешает вам сделать это. Если вы хотите, чтобы вышестоящие сервисы получали предупреждение об изменении местоположения нижестоящего сервиса, — сделайте это сами.

Сегодня я бы не пошел по этому пути. Инструментарий в этой области уже достаточно развит, и в данном случае это не просто изобретение колеса заново, но и воссоздание гораздо худшего его проявления.

Не забывайте о людях!

Рассмотренные до этого момента системы позволяют экземпляру сервиса легко регистрировать себя и искать другие сервисы для взаимодействия. Но людям тоже иногда нужна эта информация. И чтобы они могли ее потреблять, например, с помощью API для получения подробной информации в гуманных реестрах (эту тему мы рассмотрим в ближайшее время), обеспечить доступность таких сведений жизненно важно.

Сервисные сети и API-шлюзы

Не многие области технологий, связанные с микросервисами, привлекли к себе столько внимания и шумихи и внесли столько путаницы, как сервисные сети и API-шлюзы. У обоих есть свое место, но, как ни странно, их обязанности могут дублироваться. API-шлюз, в частности, подвержен неправильному использованию (и мисселингу, от англ. misselling — «неправильная продажа»), поэтому важно понимать, как эти типы технологий могут вписаться в микросервисную архитектуру. Вместо того чтобы пытаться дать подробное описание того, что можно сделать с помощью этих продуктов, я хочу рассказать, где они применяются, чем они могут быть полезны, а также о некоторых подводных камнях.

Говоря языком центра обработки данных, под трафиком «восток — запад» мы подразумеваем трафик внутри этого центра, а под трафиком «север — юг» — взаимодействия, которые входят в центр обработки данных или выходят из него, сообщаясь с внешним миром. С точки зрения сетевого взаимодействия понятие центра обработки данных стало несколько размытым, поэтому для наших целей будем говорить о сетевом периметре в более широком смысле. Это может относиться ко всему центру обработки данных, кластеру Kubernetes или, возможно, просто к концепции виртуальной сети, такой как группа машин, работающих в одной виртуальной локальной сети.

Говоря в общем, API-шлюз расположен по периметру вашей системы и имеет дело с трафиком «север — юг». Его основная задача — управление доступом из внешнего мира к вашим внутренним микросервисам. С другой стороны, сервисная сеть очень тесно связана с коммуникацией между микросервисами внутри периметра — с трафиком «восток — запад», как показано на рис. 5.6.

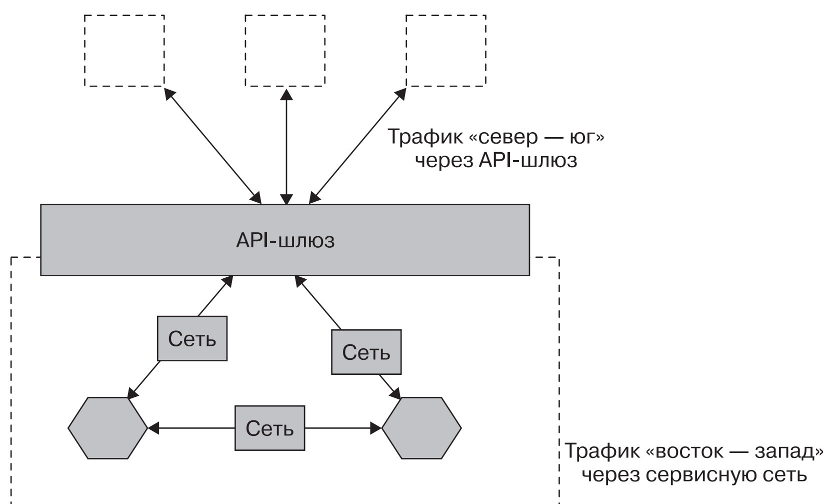


Рис. 5.6. Обзор мест использования API-шлюзов и сервисных сетей

Сервисные сети и API-шлюзы потенциально позволяют микросервисам совместно использовать код, не требуя создания новых клиентских библиотек или новых микросервисов. Проще говоря, они способны работать как прокси-серверы между микросервисами. Это означает, что они могут использоваться для реализации некоторого поведения, не зависящего от микросервиса. В противном случае такое поведение могло бы выполняться в коде, например обнаружение сервисов или сохранение логов.

При использовании API-шлюза или сервисной сети для реализации общего поведения микросервисов важно, чтобы оно было полностью универсальным — другими словами, чтобы поведение прокси-сервера не имело никакого отношения к какому-либо конкретному поведению отдельного микросервиса.

Теперь, объяснив это, я должен сказать, что мир не всегда так однозначен. Ряд API-шлюзов также пытается предоставить возможности для трафика «восток — запад». Но об этом мы поговорим немного позже. Сначала рассмотрим API-шлюзы и на что они способны.

API-шлюзы

Поскольку API-шлюз больше ориентирован на трафик «север — юг», основной его задачей в среде микросервисов является сопоставление запросов от внешних сторон к внутренним микросервисам. Эта задача сродни той, которую вы могли бы решить с помощью простого HTTP-прокси, и фактически API-шлюзы, как правило, наращивают больше функций поверх существующих HTTP-прокси и в основном функционируют как обратные прокси. Кроме того, API-шлюзы

могут использоваться для реализации таких механизмов, как API-ключи для внешних участников, ведение логов, ограничение скорости и т. п. Некоторые продукты API-шлюзов также предоставляют порталы для разработчиков, часто ориентированные на внешних потребителей.

Часть путаницы вокруг этих шлюзов связана с историей. Некоторое время назад был огромный интерес к так называемой API-экономике. В ИТ-отрасли начали понимать силу API для управляемых решений, от продуктов SaaS, таких как Salesforce, до платформ, подобных AWS, поскольку было ясно, что API дает клиентам гораздо больше гибкости в использовании их программного обеспечения. Это побудило многих людей обратить внимание на имеющееся у них ПО и рассмотреть преимущества предоставления этой функциональности своим клиентам не только через графический интерфейс, но и через API. Была надежда, что это откроет более широкие рыночные возможности и в общем принесет больше денег. На фоне этого интереса появился ряд продуктов API-шлюзов, призванных помочь в достижении этих целей. Их набор функций в значительной степени опирался на устаревшие API-ключи для третьих сторон, соблюдение ограничений скорости и отслеживание использования в целях возврата средств. Реальность такова, что, хотя API показали себя отличным способом доставки сервисов некоторым клиентам, экономическая отдача API оказалась не такой большой, как некоторые надеялись, и многие компании обнаружили, что они приобрели продукты API-шлюзов, нагруженные функциями, которые им на самом деле никогда не были нужны.

В большинстве случаев API-шлюз используется в основном для управления доступом к микросервисам организации из ее собственных клиентов с графическим интерфейсом (веб-страниц, собственных мобильных приложений) через Интернет. Здесь нет никакой «третьей стороны». Необходимость API-шлюза в какой-либо форме для Kubernetes имеет важное значение, поскольку Kubernetes изначально обрабатывает сеть только внутри кластера и ничего не делает для обработки связи с самим кластером. Но в таком случае API-шлюз, предназначенный для внешнего стороннего доступа, будет излишеством.

Поэтому, если вам нужен API-шлюз, определитесь, чего вы от него ожидаете. На самом деле я бы пошел немного дальше и сказал, что вам, вероятно, следует избегать использования API-шлюза, выполняющего слишком много задач. Но мы обсудим это позже.

Где использовать

Как только вы поймете, какие варианты применения у вас есть, вам станет немного легче определиться с типом шлюза. Если речь идет только о предоставлении доступа к микросервисам, работающим в Kubernetes, можно запустить свои собственные обратные прокси-серверы. Или, что еще лучше, выбрать специализированный продукт, такой как Ambassador, который был создан с нуля именно для этого. Если вам действительно нужно управлять большим

количеством сторонних пользователей, получающих доступ к вашему API, то стоит обратить внимание на другие продукты. Скорее всего, у вас будет несколько шлюзов, чтобы лучше справляться с разделением задач, и это разумное решение во многих ситуациях, хотя обычные предостережения, связанные с увеличением общей сложности системы и увеличением сетевых переходов, все еще актуальны.

Время от времени мне приходилось работать напрямую с поставщиками, чтобы помочь с выбором инструмента. Я могу уверенно сказать, что сталкивался с бо́льшим количеством мисселинга и плохого поведения в области API-шлюзов, чем в любой другой сфере, и поэтому в данной главе вы не найдете ссылок на продукты некоторых поставщиков. Я считаю, что во многом это связано с тем, что компании, поддерживаемые венчурным капиталом, создавали продукт в эпоху бума API-экономики, но обнаружили, что этого рынка не существует, и поэтому они сражаются на два фронта: борются за небольшое количество пользователей, которым действительно нужно то, что предлагают более сложные шлюзы, и одновременно теряют позиции в бизнесе из-за более правильно ориентированных продуктов API-шлюзов, созданных для подавляющего большинства более простых потребностей.

Чего следует избегать

Отчасти из-за очевидного отчаяния некоторых поставщиков API-шлюзов были выдвинуты различные заявления о возможностях этих продуктов. Это привело к большому количеству случаев их неправильного использования и к досадному недоверию к тому, что в основе своей представляется довольно простой концепцией. Два ключевых примера неправильного использования API-шлюзов, с которыми сталкивался я, — это агрегирование вызовов и переписывание протоколов. Но я также наблюдаю стремление к более широкому использованию API-шлюзов для вызовов внутри периметра («восток — запад»).

В этой главе мы уже кратко рассмотрели полезность такого протокола, как GraphQL. Он поможет в ситуации, когда требуется выполнить ряд вызовов, а затем объединить и отфильтровать результаты. Однако у людей часто возникает соблазн решить эту проблему на уровне API-шлюзов. Все начинается достаточно невинно: вы объединяете пару вызовов и возвращаете одну полезную нагрузку. Затем начинаете выполнять другой нисходящий вызов как часть того же агрегированного потока. После происходит попытка добавить условную логику, и вскоре вы понимаете, что превратили основные бизнес-процессы в сторонний инструмент, плохо подходящий для этой задачи.

Если вам понадобится выполнить агрегацию и фильтрацию вызовов, тогда обратите внимание на потенциал GraphQL или шаблон BFF, о котором мы поговорим в главе 14. Если выполняемое вами агрегирование вызовов, в сущности, представляет собой бизнес-процесс, то это лучше сделать с помощью явно смоделированной саги, о которой я расскажу в главе 6.

Кроме агрегации, в качестве преимущества использования API-шлюзов называется перезапись протокола. Я помню, как один неназванный поставщик очень агрессивно продвигал идею, что его продукт может «превратить любой SOAP API в REST API». Во-первых, REST — это целостное архитектурное мышление, которое не получится просто реализовать на прокси-уровне. Во-вторых, переписывание протоколов, что, по сути, и пытается сделать этот продукт, не должно осуществляться на промежуточных уровнях, поскольку это приводит к тому, что слишком большая часть поведения системы оказывается в неправильном месте.

Основная проблема как с возможностью перезаписи протокола, так и с реализацией агрегации вызовов внутри API-шлюзов заключается в том, что мы нарушаем правило сохранения каналов «глупыми», а конечных точек — «умными». «Умники» в нашей системе хотят жить в нашем коде, где у нас над ними полный контроль. API-шлюз в данном примере представляет собой канал — и хотелось бы, чтобы он был как можно более простым. С помощью микросервисов мы продвигаем модель, в которой изменения могут быть внесены и легко выпущены благодаря возможности независимого развертывания. Сохранение «умников» в наших микросервисах способствует этому. Если теперь нам также придется вносить изменения в промежуточные слои, ситуация усложнится. Учитывая важность API-шлюзов, изменения в них часто жестко контролируются. Маловероятно, что отдельным командам будет предоставлена свобода действий для самостоятельного изменения этих часто централизованно управляемых сервисов. Что это значит? Создание тикетов. Чтобы внедрить изменения в ваше ПО, в конечном счете команда API-шлюза сделает это за вас. Чем больше информации о поведении просачивается в API-шлюзы (или в сервисные шины предприятия), тем больше вы подвергаетесь риску передачи данных, усиления координации и замедления доставки.

Последняя проблема заключается в использовании API-шлюза в качестве посредника для всех вызовов между микросервисами. Это может быть крайне проблематично. Если мы вставляем API-шлюз или обычный сетевой прокси-сервер между двумя микросервисами, то обычно добавляем по крайней мере один сетевой переход. Вызов из микросервиса А в микросервис Б сначала отправляется от А к API-шлюзу, а затем от API-шлюза к Б. Необходимо учитывать влияние задержки дополнительного сетевого вызова и накладные расходы на все, что делает прокси-сервер. Сервисные сети, которые мы рассмотрим далее, гораздо лучше подходят для решения этой проблемы.

Сервисные сети

В сервисной сети общие функции, связанные с обменом данными между микросервисами, передаются в сеть. Это сокращает функциональность, которую микросервису необходимо реализовать внутри компании, в то же время обеспечивая согласованность в выполнении определенных действий.

Общие функции, реализуемые сервисными сетями, включают двухсторонний протокол TLS, идентификаторы корреляции, обнаружение сервисов, балансировку нагрузки и многое другое. Часто этот тип функциональности довольно универсален для разных микросервисов, поэтому в конечном счете применяется общая библиотека для ее обработки. Но тогда вам придется решать, что делать, если у разных микросервисов будут запущены разные версии библиотек или если у вас есть микросервисы, написанные в разных средах выполнения.

В Netflix в свое время требовали, чтобы все нелокальные сетевые коммуникации осуществлялись от JVM к JVM. Это было сделано для обеспечения повторного использования проверенных общих библиотек, которые представляют собой жизненно важную часть управления связью между микросервисами. Однако с применением сервисной сети появляется возможность повторного использования общих функций между микросервисами, написанными на разных языках программирования. Сервисные сети также могут быть невероятно полезны при реализации стандартного поведения в созданных разными командами микросервисах. И использование сервисной сети, особенно в Kubernetes, все чаще становится неотъемлемой частью любой платформы, которую вы можете создать для самостоятельного развертывания и управления микросервисами.

Упрощение реализации общего поведения между микросервисами относится к серьезным преимуществам сервисной сети. Если бы эта общая функциональность была реализована исключительно с помощью общих библиотек, преобразование этого поведения потребовало бы, чтобы каждый микросервис загружал новую версию указанных библиотек и был развернут до того, как это изменение вступит в силу. При применении сервисной сети вы получаете гораздо больше гибкости при внедрении изменений с точки зрения взаимодействия между микросервисами, не требуя повторных сборки и развертывания.

Как они работают

В целом мы ожидаем, что при микросервисной архитектуре трафик «север — юг» будет меньше, чем «восток — запад». Один вызов «север — юг» — например, размещение заказа — может привести к нескольким вызовам «восток — запад». Это означает, что при рассмотрении любого вида прокси-сервера для вызовов внутри периметра необходимо знать о накладных расходах, причиной которых могут стать эти дополнительные вызовы, и это является основной идеей при построении сервисных сетей.

Сервисные сети бывают разных форм и размеров, но их объединяет архитектура, основанная на попытке ограничить воздействие, причиной которому становятся вызовы к прокси-серверу и исходящие от него. Это достигается главным образом за счет распределения прокси-процессов для выполнения на тех же физических машинах, что и экземпляры микросервиса, чтобы гарантировать ограничение количества удаленных сетевых вызовов. На рис. 5.7 показан данный процесс в действии: сервис `Обработчик заказов` отправляет запрос

микросервису `Оплата`. Этот вызов сначала направляется локально к экземпляру прокси-сервера, работающему на том же компьютере, что и `Обработчик заказов`, прежде чем перейти к микросервису `Оплата` через его локальный экземпляр прокси-сервера. `Обработчик заказов` думает, что он выполняет обычный сетевой вызов, не подозревая, что вызов маршрутизируется локально на компьютере, а это значительно быстрее (а также менее подвержено разделению).

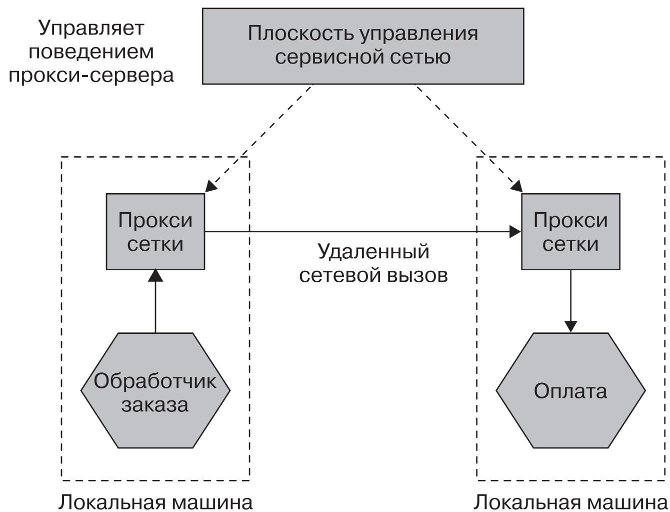


Рис. 5.7. Для обработки всей прямой связи между микросервисами развертывается сервисная сеть

Плоскость управления будет располагаться поверх локальных прокси сеток, действуя одновременно как место, в котором изменяется поведение этих прокси, и как место, в котором собирается информация о том, что делают прокси-серверы.

В Kubernetes необходимо развертывать каждый экземпляр микросервиса в поде с его собственным локальным прокси-сервером. Отдельный модуль всегда развертывается как единое целое, поэтому вы всегда знаете, что у вас есть доступный прокси-сервер. Более того, выход из строя одного прокси повлияет только на этот один под. Такая установка также позволяет вам сконфигурировать каждый прокси по-разному для различных целей. Мы рассмотрим эти концепции более подробно в разделе «Kubernetes и оркестрация контейнеров» в главе 8.

Многие реализации сервисной сети используют прокси-сервер Envoy (https://www.envoyproxy.io) в качестве основы для локально запущенных процессов. Envoy — это облегченный прокси-сервер на C++, часто используемый в качестве строительного блока для сервисных сетей и других типов ПО на основе прокси-серверов. Например, это важный строительный блок для Istio и Ambassador.

Эти прокси, в свою очередь, контролируются плоскостью управления. Это набор программного обеспечения, помогающего видеть, что происходит, и контролировать то, что делается. Например, при использовании сервисной сети для реализации двухстороннего протокола TLS плоскость управления будет использоваться для распространения сертификатов клиента и сервера.

Разве сервисные сети не относятся к интеллектуальным каналам?

Все эти разговоры о внедрении общего поведения в сервисную сеть у некоторых из вас могут вызвать тревогу. Разве этот подход не подвержен тем же проблемам, что и сервисные шины предприятия или чрезмерно раздутые API-шлюзы? Не рискуем ли мы втиснуть слишком много «умников» в свою сервисную сеть?

Главное — помнить, что общее поведение, которое мы помещаем в сети, не становится специфичным для какого-либо одного микросервиса. Никакая бизнес-функциональность не просочилась наружу. Мы создаем конфигурации типовых ситуаций, например, процесс обработки тайм-аутов запросов. Что касается общего поведения, которое может потребовать настройки для каждого микросервиса, — это, как правило, то, что хорошо обслуживается без необходимости выполнения работы на центральной платформе. Например, с помощью Istio можно определить свои требования к тайм-аутам на основе самообслуживания, просто изменив свое определение сервиса.

Надо ли вам применять эту технологию?

После выхода первого издания книги использование сервисных сетей начало приобретать популярность, и я увидел много достоинств в этой идее, но также заметил и большой отток пользователей. Были предложены, построены, а затем отброшены различные модели развертывания, и число компаний, предлагающих решения в этой области, резко возросло. Но даже для инструментов, существовавших долгое время, наблюдался очевидный недостаток стабильности. Linkerd (https://linkerd.io), которая, пожалуй, как никто другой внесла свой вклад в развитие сервисной сети, стала пионером в этой области, полностью перестроила свой продукт с нуля при переходе от версии 1 к версии 2. Istio (https://istio.io) представляла собой сервисную сеть, одобренную Google. Этому продукту потребовались годы, чтобы добраться до первоначального релиза 1.0, и даже после этого в его архитектуре произошли значительные преобразования (по иронии судьбы, хотя это и разумно, был выполнен переход к более монолитной модели развертывания для своей плоскости управления).

На протяжении последних пяти лет, когда меня спрашивали: «Стоит ли нам применять сервисную сеть?» — мой совет звучал так: «Если вы можете позволить себе подождать полгода, прежде чем сделать выбор, — подождите». Я был увлечен этой идеей, но переживал о стабильности. Рисковать чем-то вроде сервисной сети не очень хочется, ведь эта часть системы так важна, так необходима для того, чтобы все стабильно работало. Вы ставите систему на критический

путь. По степени серьезности я бы отнесся к этому наравне с выбором брокера сообщений или облачного провайдера.

Я рад сообщить, что с тех пор эта область стала более зрелой. Отток в некоторой степени замедлился, и все еще сохраняется (здоровое) множество поставщиков. Тем не менее сервисные сети подходят не для всех. Во-первых, если вы не используете Kubernetes, ваши возможности ограничены. Во-вторых, они действительно добавляют сложности. Если у вас пять микросервисов, не уверен, что получится легко оправдать применение сервисной сети (можно поспорить о том, оправданно ли использование Kubernetes, если у вас только пять микросервисов!). Организациям, у которых больше микросервисов, особенно если они хотят, чтобы эти микросервисы были написаны на разных языках программирования, стоит обратить внимание на сервисные сети. Однако знайте: переход между сервисными сетями достаточно проблематичный!

Monzo — одна из организаций, в которой открыто заговорили о важности применения сервисной сети для запуска своей архитектуры в имеющемся масштабе. Они использовали Linkerd первой версии для управления вызовами RPC между микросервисами, и это оказалось чрезвычайно выгодно. Интересно, что когда более старая архитектура Linkerd v1 перестала удовлетворять требованиям организации, Monzo пришлось перенести (https://oreil.ly/5dLGC) все трудности от такой миграции, чтобы достичь необходимого масштаба. В конце концов предприятие эффективно перешло на внутреннюю сервисную сеть, использующую прокси-сервер Envoy.

А как насчет других протоколов?

API-шлюзы и сервисные сети в основном используются для обработки вызовов, связанных с HTTP. Таким образом, с помощью данных продуктов можно управлять REST, SOAP, gRPC и т. п. Однако все становится немного более туманным, когда вы начинаете рассматривать коммуникацию с помощью других протоколов, например используя такие брокеры сообщений, как Kafka. Обычно на этом этапе сервисную сеть обходят стороной — связь осуществляется непосредственно с самим брокером. Это означает, что нельзя считать, что сервисная сеть способна работать в качестве посредника для всех вызовов между микросервисами.

Документирование сервисов

Разбивая системы на более мелкие микросервисы, мы рассчитываем выявить множество швов в виде API, которые люди могут использовать для выполнения многих, как мы надеемся, замечательных задач. Если до этого момента мы все сделали правильно, то должны знать, где что находится. Но как мы узнаем, что эти блоки делают или как их использовать? Очевидно, что одним из вариантов

будет воспользоваться документацией по API. Да, она часто может быть устаревшей. В идеале необходимо обеспечить, чтобы документация всегда обновлялась с помощью API-микросервиса, и упростить ее просмотр, когда известно, где находится конечная точка сервиса.

Явные схемы

Наличие явных схем действительно значительно облегчает представление о любой конкретной конечной точке, но схем самих по себе часто недостаточно. Как мы уже обсуждали, схемы показывают структуру, но они недостаточно хорошо передают поведение конечной точки, поэтому все равно может потребоваться качественная документация, чтобы помочь потребителям понять, как использовать конечную точку. Конечно, стоит отметить, что, если вы не будете использовать явную схему, ваша документация в итоге *сильно* усложнится. Вам потребуется объяснить, что делает конечная точка, а также задокументировать структуру и детали интерфейса. Более того, без явной схемы сложнее определить, соответствует ли ваша документация действительности. Устаревшая документация — это постоянная проблема, но по крайней мере явная схема дает вам больше шансов на сохранение ее актуальности.

Я уже говорил об OpenAPI как о формате схемы, но он также очень эффективен при предоставлении документации. В настоящее время существует множество инструментов как с открытым исходным кодом, так и коммерческих, способных поддерживать использование дескрипторов OpenAPI для создания полезных порталов для работы с документацией. Стоит отметить, что порталы с открытым исходным кодом для просмотра OpenAPI кажутся несколько простыми — я изо всех сил пытался найти поддерживающий функциональность поиска портал, например. Для тех, кто работает в Kubernetes, портал разработчиков Ambassador (https://oreil.ly/8pg12) особенно интересен. Ambassador довольно популярен в качестве API-шлюза для Kubernetes, а его портал отвечает требованиям автоматического обнаружения доступных конечных точек OpenAPI. Идея развертывания нового микросервиса и автоматического доступа к его документации мне очень нравится.

В прошлом не хватало хорошей поддержки для документирования событийных интерфейсов. Теперь по крайней мере у нас есть варианты. Формат AsyncAPI появился как адаптация OpenAPI, и у нас также есть CloudEvents — проект CNCF. Я не использовал ни то ни другое, но CloudEvents меня привлекает. Он обладает обширной интеграцией и поддержкой, во многом благодаря своей связи с CNCF. Исторически по крайней мере CloudEvents казался более строгим с точки зрения формата событий по сравнению с AsyncAPI, при этом должным образом поддерживался только JSON, пока поддержка Protocol buffers не была недавно вновь добавлена после удаления. Так что это тоже заслуживает внимания.

Самоописывающаяся система

В ходе ранней эволюции SOA появились такие стандарты, как Universal Description, Discovery, and Integration (UDDI), которые помогли нам разобраться, какие сервисы запущены. Эти подходы были довольно объемными, что привело к появлению альтернативных методов. Мартин Фаулер обсудил концепцию гуманного реестра (https://oreil.ly/UI0YJ) — гораздо более простого подхода, при котором люди могут записывать информацию о сервисах в организации так же просто, как в вики (wiki).

Важно получить представление о нашей системе и о том, как она себя ведет, особенно когда мы работаем с масштабной системой. Мы рассмотрели ряд различных методов, помогающих получить представление непосредственно из системы. Отслеживая работоспособность нижестоящих сервисов вместе с идентификаторами корреляции, которые позволяют видеть цепочки вызовов, мы получаем реальные данные о взаимодействии сервисов. Используя системы обнаружения сервисов, например Consul, можно видеть, где работают микросервисы. Такие механизмы, как OpenAPI и CloudEvents, помогают увидеть, какие возможности размещаются на любой заданной конечной точке, в то время как листы проверки работоспособности и системы мониторинга позволяют узнать о работоспособности как всей системы, так и отдельных сервисов.

Вся эта информация доступна программно и позволяет сделать гуманный реестр более эффективным, чем простая вики-страница, которая, несомненно, устареет. Вместо этого необходимо применять подобный реестр для использования и сбора всей информации, отправляемой нашей системой. Создавая пользовательские информационные панели, можно собрать воедино огромный массив доступной информации, которая поможет разобраться в нашей экосистеме.

Обязательно начните с чего-то такого простого, как статическая веб- или вики-страница, которая будет собирать немного данных из работающей системы. Но со временем старайтесь получать все больше и больше информации. Обеспечение легкодоступности этой информации представляет собой ключевой инструмент для управления возникающими сложностями, которые появятся в результате масштабной эксплуатации таких систем.

Я имел дело с рядом компаний, столкнувшихся с подобными проблемами. Они в итоге создали простые внутренние реестры, помогающие сопоставлять метаданные по сервисам. Некоторые из этих реестров просто сканируют хранилища исходного кода в поисках файлов метаданных, чтобы создать список доступных сервисов. Эта информация может быть объединена с реальными данными, поступающими из систем обнаружения сервисов, таких как Consul или etcd, для создания более полной картины того, что запущено и с кем вы могли бы поговорить об этом.

В Financial Times разработали Biz Ops, чтобы помочь решить эту проблему. Компания располагает несколькими сотнями сервисов, спроектированных командами по всему миру. Инструмент Biz Ops (рис. 5.8) предоставляет компании единое место, где собрано много полезной информации о ее микросервисах в дополнение к информации о других сервисах ИТ-инфраструктуры, таких как сети и файловые серверы. Biz Ops, надстроенный поверх графической базы данных, обладает большой гибкостью в отношении данных, которые он собирает, и способов их моделирования.

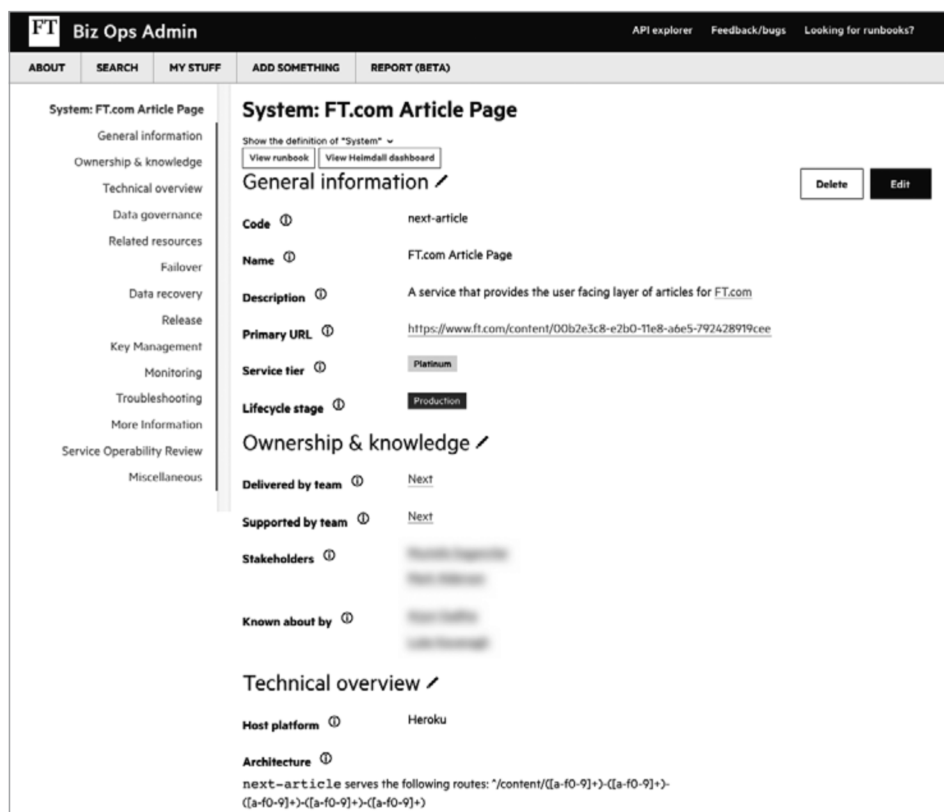


Рис. 5.8. Инструмент Biz Ops от Financial Times, который собирает информацию о микросервисах

Однако Biz Ops ушел дальше, чем большинство попадавшихся мне аналогичных инструментов. Он вычисляет показатели работоспособности системы (рис. 5.9). Идея заключается в том, что есть определенные процессы, которые сервисы и их команды должны выполнять для обеспечения простоты управления микросервисами. Такие процессы могут варьироваться от проверки правильности предоставленной командой информации в реестре до обеспечения

надлежащих проверок дееспособности сервисов. Оценка работоспособности системы после ее расчета позволяет командам сразу увидеть, есть ли какие-либо проблемы, требующие исправления.

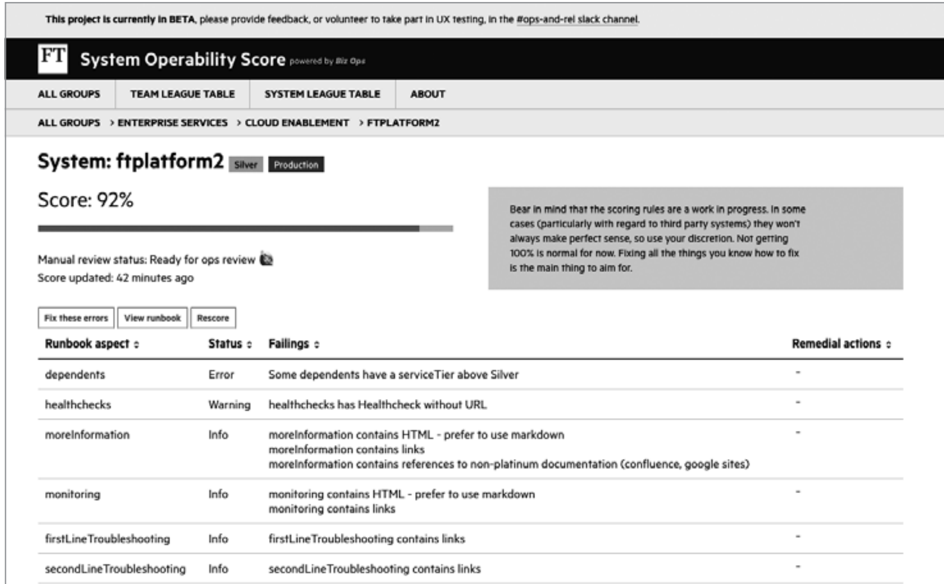


Рис. 5.9. Пример оценки работоспособности сервиса для микросервиса в Financial Times

Это развивающаяся область. В мире открытого исходного кода инструмент Backstage (https://back stage.io) от Spotify предлагает механизм для создания каталога сервисов, подобного Biz Ops, с подключаемой моделью, позволяющей разрабатывать сложные дополнения, такие как возможность инициировать создание нового микросервиса или извлекать оперативную информацию из кластера Kubernetes. Каталог Service Catalog (https://oreil.ly/7o649) от Ambassador более узко ориентирован на видимость сервисов в Kubernetes. Скорее всего, он не так привлекателен, как, например, Biz Ops от FT, но тем не менее приятно видеть некоторые новые более доступные подходы к этой идее.

Резюме

Итак, в этой главе мы рассмотрели много вопросов, давайте кратко вспомним некоторые из них.

Убедитесь, что проблема, которую вы пытаетесь решить, определяет ваш выбор технологии. Для начала определитесь с контекстом и предпочитаемым стилем коммуникации, а затем выберите наиболее подходящую технологию —

не наоборот! Краткое описание стилей взаимодействия между микросервисами, впервые представленное в главе 4 и снова показанное на рис. 5.10, может помочь вам в принятии решений, но простое следование этой модели не панацея. Сядьте и обдумайте все.

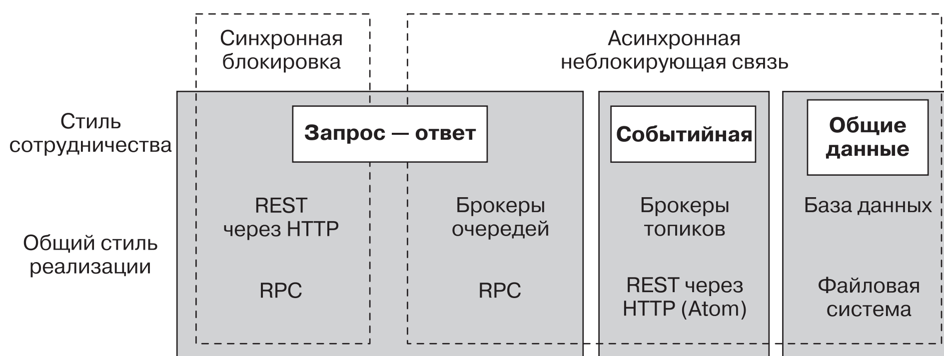


Рис. 5.10. Различные стили межмикросервисной коммуникации наряду с примерами технологий внедрения

- Какой бы выбор вы ни сделали, рассмотрите возможность использования схем, чтобы сделать ваши контракты более явными и помочь выявить случайные изменения.
- Там, где это возможно, стремитесь вносить обратно совместимые изменения, чтобы сохранить возможность независимого развертывания.
- Если вам все же придется вносить обратно несовместимые изменения, найдите способ предоставить потребителям время для обновления, чтобы избежать поэтапного развертывания.
- Подумайте, как вы можете помочь людям получить информацию о конечных точках, — рассмотрите использование гуманных реестров и т. п., чтобы разобраться в хаосе.

Мы рассмотрели возможности реализации вызова между двумя микросервисами, но что произойдет, если потребуется координировать операции между несколькими микросервисами? Этому будет посвящена наша следующая глава.

ГЛАВА 6

Рабочий поток

В двух предыдущих главах мы рассмотрели особенности взаимодействия микросервисов. Но что делать, когда требуется взаимодействие нескольких микросервисов, например, для реализации бизнес-процесса? Процесс моделирования и реализации такого рода потоков работ в распределенных системах, может оказаться сложной задачей.

В данной главе мы рассмотрим подводные камни, связанные с использованием распределенных транзакций для решения этой проблемы, а также шаблоны «Сага» — концепцию, способную помочь моделировать рабочие потоки микросервисов гораздо более приемлемым образом.

Транзакции базы данных

Размышляя о транзакции в контексте вычислений, мы представляем себе одно или несколько действий, которые должны произойти и которые мы хотим рассматривать как единое целое. При внесении нескольких изменений в рамках одной и той же общей операции необходимо подтверждение, все ли преобразования были внесены. Также нужен способ производить очистку, если во время этих изменений возникает ошибка. Как правило, это приводит к тому, что мы используем что-то вроде транзакции базы данных.

В базах данных транзакция используется, чтобы убедиться, что одна или несколько модификаций состояния были выполнены успешно, например удаление, вставка или изменение данных. В реляционной БД этот процесс может включать в себя обновление нескольких таблиц в рамках одной транзакции.

Транзакции ACID

Обычно, когда обсуждаются транзакции БД, речь идет об ACID. ACID — это аббревиатура, описывающая ключевые свойства транзакций базы данных, приводящие к созданию системы, на которую можно положиться для обеспечения

долговечности и согласованности хранилища данных. ACID означает *атомарность (atomicity)*, *согласованность (consistency)*, *изоляцию (isolation)* и *устойчивость (durability)*.

Атомарность

Гарантирует, что все операции, предпринятые в рамках транзакции, завершатся успешно или неудачей. Если какое-либо из вносимых изменений по какой-то причине завершается неудачей, то вся операция прерывается, и это выглядит так, будто никаких изменений никогда не было.

Согласованность

Когда в базу данных вносятся изменения, мы следим за тем, чтобы она оставалась в действительном и согласованном состоянии.

Изоляция

Позволяет нескольким транзакциям работать одновременно, не мешая друг другу. Это достигается за счет того, что любые промежуточные изменения состояния, внесенные во время одной транзакции, незаметны для других транзакций.

Устойчивость

Гарантирует, что после завершения транзакции данные не будут потеряны в случае какого-либо системного сбоя.

Стоит отметить, что не все базы данных предоставляют транзакции ACID. Все системы реляционных БД, которые я когда-либо использовал, работают так же, как и многие новые базы данных NoSQL, такие как Neo4j. Система MongoDB в течение многих лет поддерживала транзакции ACID только при внесении изменений в один документ, что могло вызвать проблемы, если требовалось выполнить атомарное обновление более чем одного документа[1].

Моя книга не предназначена для детального изучения этих концепций. Я, конечно, упростил некоторые из описаний ради краткости. Но тем, кто хочет подробнее изучить эту тему, я рекомендую «Высоконагруженные приложения»[2]. В дальнейшем мы будем в основном говорить об атомарности. Это не значит, что другие свойства неважны, просто проблема атомарности операций базы данных становится основной при начале разбивки функциональности на микросервисы.

---

[1] Ситуация изменилась, поскольку поддержка многодокументных транзакций ACID была выпущена как часть Mongo 4.0. Сам я не использовал эту функцию Mongo, просто знаю, что она существует!

[2] *Клеппман М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. — Питер, 2018.

Все еще ACID, но с недостаточной атомарностью?

Имейте в виду, что мы все еще можем использовать транзакции в стиле ACID при использовании микросервисов. Микросервис способен свободно использовать транзакцию ACID, например, для операций со своей собственной БД. Просто объем этих транзакций сводится к изменению состояния, происходящему локально в рамках этого единственного микросервиса. Рассмотрим рис. 6.1. Здесь мы отслеживаем процесс, связанный с привлечением нового клиента в MusicCorp. Мы подошли к концу процесса, который включает в себя изменение `Статуса` покупателя `2346` с `РАССМАТРИВАЕТСЯ` на `ПОДТВЕРЖДЕН`. Поскольку регистрация уже завершена, нам требуется удалить соответствующую строку из таблицы `PendingEnrollments`. С одной базой данных это выполняется в рамках одной транзакции базы данных ACID: либо происходят оба изменения состояния, либо ни одно из них.

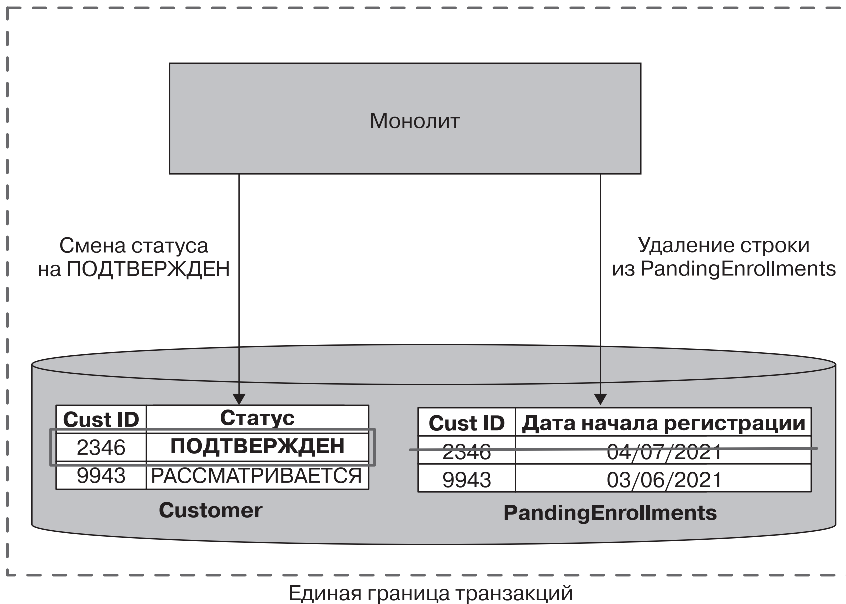


Рис. 6.1. Обновление двух таблиц в рамках одной транзакции ACID

Сравните эту картину с рис. 6.2, где вносились точно такие же изменения, только в отдельную базу данных. Такой процесс означает, что необходимо рассмотреть две транзакции, каждая из которых может сработать или завершиться неудачей независимо от другой.

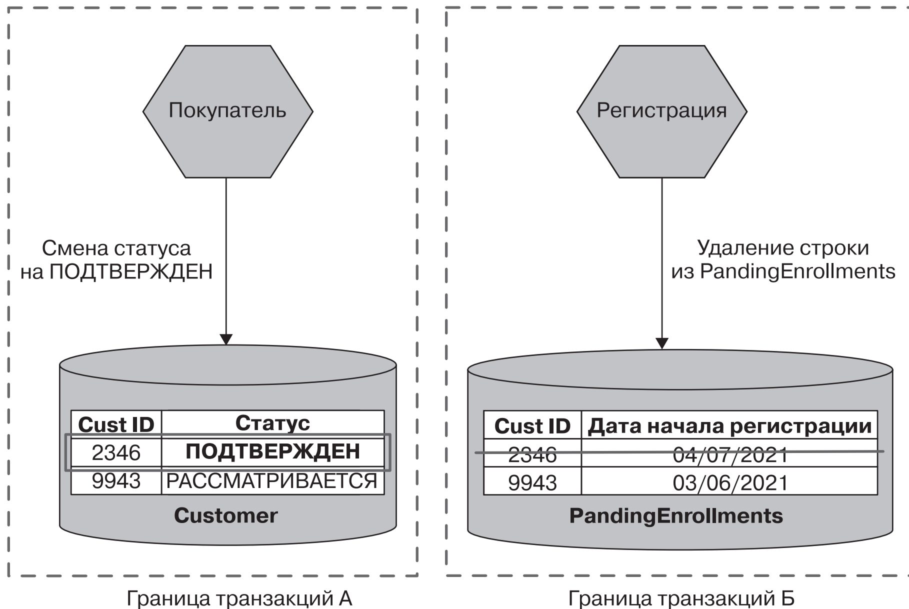


Рис. 6.2. Изменения, внесенные микросервисами Покупатель и Регистрация, теперь выполняются в рамках двух разных транзакций

Конечно, можно было бы упорядочить эти две транзакции, удалив строку из таблицы `PendingEnrollments`, только если бы мы могли изменить строку в таблице `Customer`. Но нам все равно пришлось бы задуматься о дальнейших шагах, если удаление из таблицы `PendingEnrollments` завершится неудачей, то всю логику необходимо было бы реализовать самостоятельно. Однако возможность изменять порядок шагов выполнения, чтобы лучше справляться с этими вариантами использования, может быть действительно полезной идеей (к которой мы вернемся, когда будем изучать шаблоны «Сага»). Но нужно признать, что, разложив эту операцию на две отдельные транзакции базы данных, мы утратили гарантированную атомарность операции в целом.

Отсутствие атомарности может повлечь за собой неприятные последствия, особенно при переносе систем, ранее полагавшихся на это свойство. Обычно первым вариантом, который люди начинают рассматривать, по-прежнему остается использование одной транзакции, охватывающей несколько процессов, — распределенной транзакции. Вскоре мы убедимся, что распределенные транзакции — далеко не всегда правильный путь развития. Давайте рассмотрим один из наиболее распространенных алгоритмов реализации распределенных транзакций — двухфазную фиксацию как способ изучения проблем, связанных с распределенными транзакциями в целом.

Распределенные транзакции — двухфазная фиксация

Алгоритм двухфазной фиксации (2PC, two-phase commit — «двухфазный коммит») часто используется в попытке дать возможность вносить транзакционные изменения в распределенную систему, где может потребоваться обновление нескольких отдельных процессов в рамках общей операции. Распределенные транзакции и, в частности, 2PC часто рассматриваются командами, переходящими на микросервисные архитектуры, как способ решения встречающихся проблем. Однако вскоре мы обнаружим, что они могут не только не решить ваши проблемы, но и еще больше запутать систему.

Алгоритм 2PC разбит на две фазы (отсюда и название *«двухфазная фиксация»*): фаза подготовки и фаза фиксации. На *фазе подготовки* центральный координатор связывается со всеми исполнителями, которые должны участвовать в транзакции, и запрашивает подтверждение того, можно ли внести какие-либо изменения в состояние. На рис. 6.3 показаны два запроса: один для изменения статуса покупателя на `ПОДТВЕРЖДЕН` и второй для удаления строки из таблицы `PendingEnrollments`. Если все исполнители согласны с тем, что изменение состояния, о котором их просят, допустимо, алгоритм переходит к следующему этапу. Если преобразование не может быть выполнено, например, потому что запрошенное изменение состояния нарушает какое-то локальное условие, вся операция прерывается.

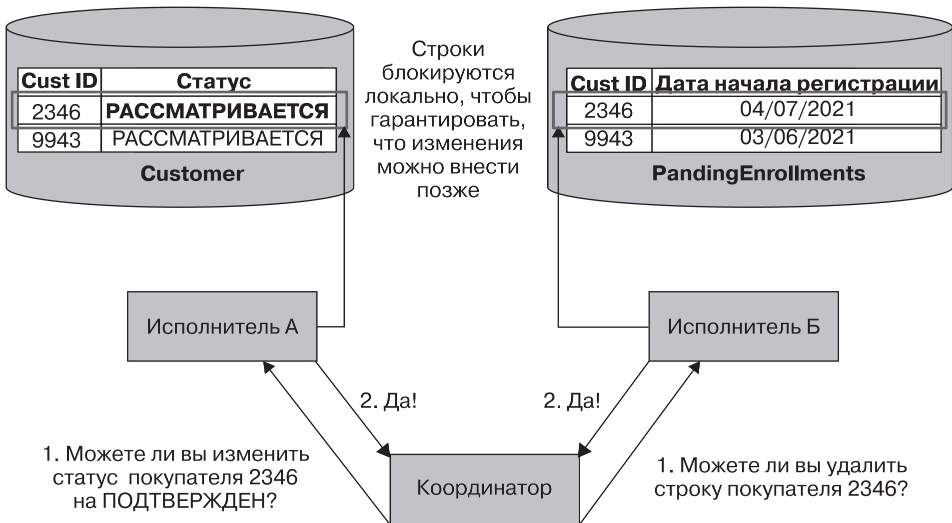


Рис. 6.3. В первой фазе двухфазной фиксации исполнители «голосуют», чтобы решить, могут ли они осуществить некоторые локальные изменения состояния

Важно подчеркнуть, что изменение не вступает в силу сразу после указания о его внесении. Вместо этого исполнитель гарантирует, что он сможет внести это изменение в какой-то момент в будущем. Как он может дать такую гарантию? На рис. 6.3, например, `Исполнитель А` сообщает, что готов изменить статус строки в таблице `Customer`, чтобы обновить статус конкретного покупателя на `ПОДТВЕРЖДЕН`. Что, если другая операция в какой-то более поздний момент удалит строку или внесет какое-то другое менее значительное преобразование, которое тем не менее означает, что изменение статуса на `ПОДТВЕРЖДЕН` позже станет недействительным? Чтобы гарантировать, что отредактированный статус будет применен позже, `Исполнитель А`, вероятно, заблокирует запись, чтобы защитить ее от других изменений.

Если какие-либо рабочие процессы не проголосовали за фиксацию, всем сторонам должно быть отправлено сообщение об откате, чтобы убедиться, что они могут выполнить локальную очистку. Это позволяет исполнителям снять любые блокировки. Если же они согласились внести изменения, мы переходим к фазе фиксации, как показано на рис. 6.4. Здесь изменения фактически вносятся, и связанные с ними блокировки снимаются.

Рис. 6.4. При использовании 2PC на фазе фиксации изменения фактически применяются

Важно отметить, что в такой системе нет никакой гарантии, что эти фиксации произойдут одновременно. `Координатору` необходимо отправить запрос на фиксацию всем участникам, и это сообщение может прийти и быть обработано в разное время. Если бы мы могли непосредственно наблюдать за состоянием любого рабочего процесса, то увидели бы, что в `Исполнитель А` внесены изменения,

а в Исполнитель Б — еще нет. Чем больше задержка между Координатором и участниками двухфазной фиксации и чем медленнее исполнители обрабатывают ответ, тем шире может быть это окно непоследовательности. Возвращаясь к нашему определению ACID, *изоляция* гарантирует, что мы не видим промежуточных состояний во время транзакции. Но с 2PC мы потеряем эту гарантию.

По своей сути двухфазная фиксация очень часто просто координирует распределенные блокировки. Исполнителям необходимо заблокировать локальные ресурсы, чтобы фиксацию можно было осуществить во время второго этапа. Управлять блокировками и избегать взаимоблокировок в однопроцессной системе совсем не весело. Теперь представьте себе проблемы координации блокировок между несколькими участниками. Это не очень приятно.

Существует множество видов сбоев, связанных с 2PC, на изучение которых у нас нет времени. Рассмотрим случай, когда исполнитель подготавливает продолжение транзакции, но затем не отвечает на запрос о фиксации. И что же делать? Некоторые из этих видов сбоев могут быть обработаны автоматически, но иные могут оставить систему в таком состоянии, что оператору придется устранять неполадки вручную.

Чем больше у вас участников и чем больше задержка в системе, тем масштабнее проблемы, возникающие при двухфазной фиксации. Алгоритм 2PC может стать простым способом ввести огромные задержки в вашу систему, особенно если область блокировки объемна или продолжительность транзакции велика. Именно по этой причине двухфазные фиксации обычно используются только для очень кратковременных операций. Чем дольше длится операция, тем дольше у вас заблокированы ресурсы!

Просто скажите «нет» распределенным транзакциям

По всем причинам, изложенным выше, я настоятельно рекомендую вам избегать использования распределенных транзакций, таких как двухфазная фиксация, для координации изменений состояния между вашими микросервисами. Так что же еще вы можете сделать?

Ну, первый вариант — просто не разделять данные на части. Если у вас есть фрагменты состояния, которыми вы хотите управлять по-настоящему атомарным и согласованным способом, и непонятно, как разумно получить эти характеристики без транзакции в стиле ACID, тогда оставьте это состояние в отдельной БД и управляющую этим состоянием функциональность в отдельном сервисе (или в своем монолите). Если вы находитесь в процессе поиска мест разделения своего монолита и выбора определения легких (или трудных) декомпозиций, то вполне можете решить, что разделение данных,

которые в настоящее время управляются в транзакции, слишком сложно для обработки на данном этапе. Поработайте над какой-нибудь другой областью системы и вернитесь к этому позже.

Но что, если вам действительно нужно разделить эти данные на части, но вы не хотите испытывать все трудности, связанные с управлением распределенными транзакциями? Как выполнять операции в нескольких сервисах, но избежать блокировки? Что, если операция займет минуты, дни или даже месяцы? В подобных случаях рассмотрите альтернативный подход: шаблоны «Сага».

РАСПРЕДЕЛЕННЫЕ ТРАНЗАКЦИИ БАЗЫ ДАННЫХ

Я выступаю против повсеместного использования распределенных транзакций для координации изменения состояния между микросервисами. В таких случаях каждый микросервис управляет своим собственным локальным устойчивым состоянием (например, в своей базе данных). Алгоритмы распределенных транзакций успешно используются для некоторых крупномасштабных БД. Одной из таких систем является Spanner от Google. В такой ситуации распределенная транзакция применяется нижестоящей базой данных прозрачно с точки зрения приложения, а другая распределенная транзакция просто используется для координации изменений состояния в рамках одной логической БД (хотя она может быть распределена по нескольким машинам и потенциально по нескольким центрам обработки данных).

То, чего Google удалось достичь с помощью Spanner, впечатляет. Но также стоит отметить, что проделанная работа не была простой. Скажем так, это связано с очень дорогими центрами обработки данных и атомными часами на базе спутников (я серьезно). Чтобы подробно ознакомиться с возможностями Spanner, рекомендую презентацию *Google Cloud Spanner: Global Consistency at Scale*[1].

Саги

В отличие от двухфазной фиксации *сага* по своей конструкции представляет собой алгоритм, способный координировать множественные изменения состояния, но позволяющий избежать необходимости блокировки ресурсов на длительные периоды времени. Данный шаблон делает это путем моделирования этапов как отдельных действий, которые могут выполняться независимо друг от друга. Применение саги предоставляет дополнительное возможности, заставляя явно моделировать свои бизнес-процессы, что дает значительные преимущества.

Основная идея, впервые изложенная в статье «Саги» Гектора Гарсиа-Молины и Кеннета Салема[2], касается того, как наилучшим образом обрабатывать

[1] *Kubis R.* Google Cloud Spanner: Global Consistency at Scale // Devoxx. 7 ноября 2017 года, видео на YouTube, 33:22. https://oreil.ly/XHvY5.

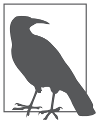
[2] *Garcia-Molina H., Salem K.* Sagas // ACM Sigmod Record 16, no. 3 (1987): 249–59.

операции, известные как *долгоживущие транзакции* (long lived transactions, LLT). Эти транзакции могут занять много времени (минуты, часы или даже дни) и потребовать внесения изменений в базу данных.

Если вы напрямую сопоставили LLT с обычной транзакцией БД, одна транзакция базы данных будет охватывать весь жизненный цикл LLT. Это может привести к длительной блокировке нескольких строк или даже целых таблиц во время выполнения LLT, что вызовет серьезные последствия, если другие процессы попытаются прочитать или изменить эти заблокированные ресурсы.

Вместо этого авторы статьи предлагают нам разбить LLT на последовательность транзакций, каждая из которых может обрабатываться независимо. Идея заключается в том, что продолжительность каждой из этих «вспомогательных» транзакций будет короче и изменит только часть данных, на которые влияет LLT целиком. В результате в базовой БД будет гораздо меньше конфликтов, поскольку объем и продолжительность блокировок значительно сократятся.

Хотя изначально саги задумывались как механизм, помогающий LLT работать с единой базой данных, они так же хорошо работают для координации изменений в нескольких сервисах. Можно разбить один бизнес-процесс на набор вызовов, которые будут совершаться к взаимодействующим сервисам, — это и есть сага.



Прежде чем двигаться дальше, вы должны понять, что в терминах ACID сага не дает нам атомарности, к которой мы привыкли при обычной транзакции БД. Поскольку мы разбиваем LLT на отдельные транзакции, у нас нет атомарности на уровне самого шаблона. Она есть для каждой отдельной транзакции внутри общей саги, поскольку каждая из них может быть связана с изменением ACID-транзакции, если это необходимо. Шаблон «Сага» дает нам достаточно информации, чтобы понять, в каком состоянии он находится. И нам решать, что из этого следует.

Давайте взглянем на простой процесс выполнения заказов для MusicCorp, описанный на рис. 6.5. Его можно использовать для дальнейшего изучения саги в контексте микросервисной архитектуры.

Здесь процесс выполнения заказа представлен как единый шаблон «Сага», причем каждый шаг в этом потоке — операция, которую может выполнить другой сервис. Внутри каждого сервиса любое изменение состояния допускается обрабатывать в рамках локальной транзакции ACID. Например, когда мы проверяем и резервируем запасы с помощью сервиса `Склад`, внутри него может быть создана строка в локальной таблице `Reservation` с записью о резервировании товара. Эти изменения будут обрабатываться в обычной транзакции базы данных.

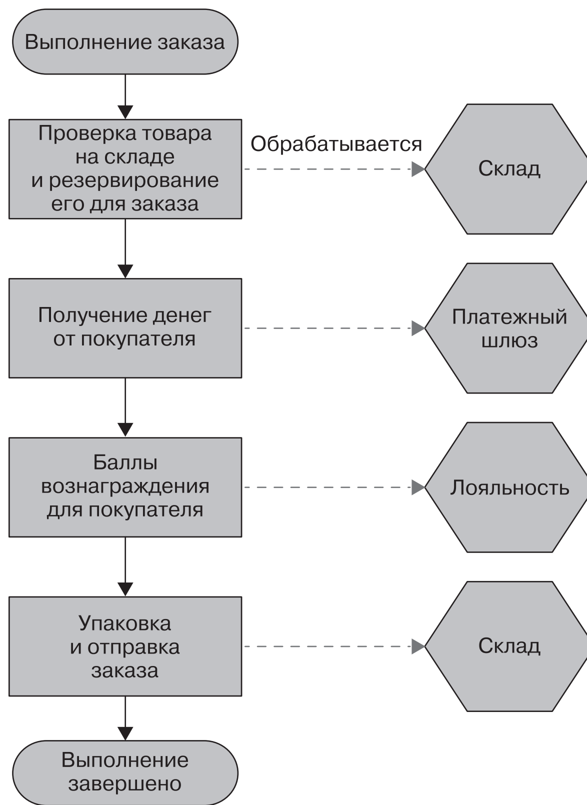


Рис. 6.5. Пример потока выполнения заказа вместе с сервисами, ответственными за проведение операции

Режимы сбоя саги

Поскольку сага разбивается на отдельные транзакции, нам необходимо придумать, как справиться с отказом, то есть восстановиться в случае сбоя. В оригинальной статье о сагах описаны два типа восстановления: обратное и прямое.

Обратное восстановление включает в себя возврат сбоя и последующую очистку — откат. Чтобы это сработало, требуется определить компенсирующие действия, которые позволят отменить ранее совершенные транзакции. *Прямое восстановление* позволяет начать с места, где произошел сбой, и продолжить обработку. Для этого у нас должна быть возможность повторять транзакции. Это, в свою очередь, подразумевает, что в системе сохраняется достаточно информации для совершения повторной попытки.

В зависимости от характера моделируемого бизнес-процесса можно ожидать, что любой режим сбоя запускает обратное восстановление, прямое или, возможно, сочетание того и другого.

Очень важно отметить, что сага позволяет восстанавливаться после *бизнес-сбоев*, а не *технических*. Например, попытка принять платеж от клиента, когда у клиента недостаточно средств, — это бизнес-сбой, с которым шаблон «Сага» должен справиться. С другой стороны, если время ожидания `Платежного шлюза` истекает или выбрасывается ошибка `500 Internal Service Error`, тогда это технический сбой, который следует обрабатывать отдельно. Сага предполагает, что базовые компоненты работают должным образом, а базовая система надежна и что затем мы координируем работу испытанных компонентов. Мы рассмотрим некоторые способы повышения надежности технических компонентов в главе 12, но для получения дополнительной информации об этом ограничении шаблонов «Сага» я рекомендую «Пределы шаблона сага» (https://oreil.ly/II0an) Уве Фридрихсена.

Откаты саги

При использовании транзакции ACID если мы сталкиваемся с проблемой, то запускаем откат до момента фиксации. После отката как будто ничего и не было: изменения, которые мы пытались внести, не применились. Однако в нашем шаблоне задействовано несколько транзакций, и некоторые из них могут быть уже зафиксированы до того, как мы решим откатить всю операцию. Итак, как нам откатить транзакции после их фиксации?

Вернемся к нашему примеру обработки заказа из рис. 6.5. Смоделируйте потенциальный режим отказа. Например, мы дошли до процесса упаковки товара, но обнаружили, что его невозможно найти на складе (рис. 6.6). Наша система считает, что товар существует, но его просто физически нет на полке!

Теперь давайте предположим, что было принято решение отменить весь заказ, вместо того чтобы предоставить клиенту возможность поместить товар в резерв. Проблема в том, что мы уже приняли оплату и начислили баллы лояльности.

Если бы все эти шаги были выполнены в рамках одной транзакции БД, простой откат очистил бы эти данные. Однако все шаги в процессе выполнения заказа обрабатывались отдельными вызовами. Каждый вызов осуществлялся в отдельной области транзакции. Простого «отката» для всей операции не существует.

Вместо этого реализовать откат можно при помощи компенсирующей транзакции. *Компенсирующая транзакция* — это операция, которая отменяет ранее зафиксированную транзакцию. Чтобы откатить процесс, необходимо запустить компенсирующую транзакцию для каждого уже зафиксированного шага в нашей саге, как показано на рис. 6.7.

Обратите внимание, что эти компенсирующие транзакции могут вести себя иначе, чем при обычном откате базы данных. Откат БД происходит до фиксации,

а после него — транзакции будто никогда и не было. В нашем примере транзакция уже *произошла*, и теперь мы создаем новую транзакцию, отменяющую изменения, внесенные исходной. Но невозможно повернуть время вспять и сделать так, будто ничего не происходило.

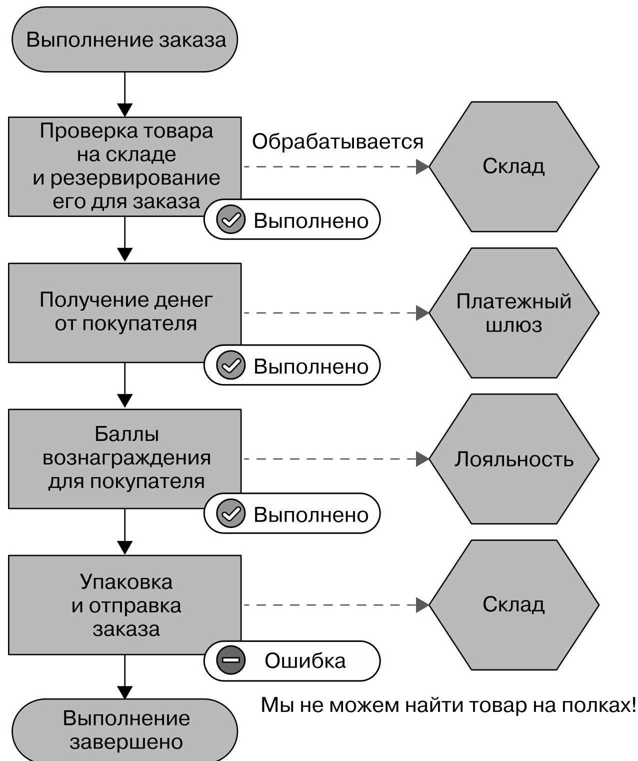


Рис. 6.6. Мы пытались упаковать наш товар, но не можем найти его на складе

Поскольку не всегда есть возможность полностью отменить транзакцию, будем называть эти компенсирующие транзакции *семантическими откатами*. Далеко не каждый раз можно убрать все данные, но мы делаем достаточно для контекста нашей саги. Например, один из шагов мог включать отправку клиенту электронного письма с сообщением о том, что его заказ уже в пути. Если потребуется отменить это действие, мы не сможем денонсировать отправку электронного письма![1] Вместо этого наша компенсирующая транзакция может привести к отправке клиенту второго электронного письма, в котором говорится, что с заказом возникла проблема и он отменен.

[1] Действительно не можем. Я пробовал!

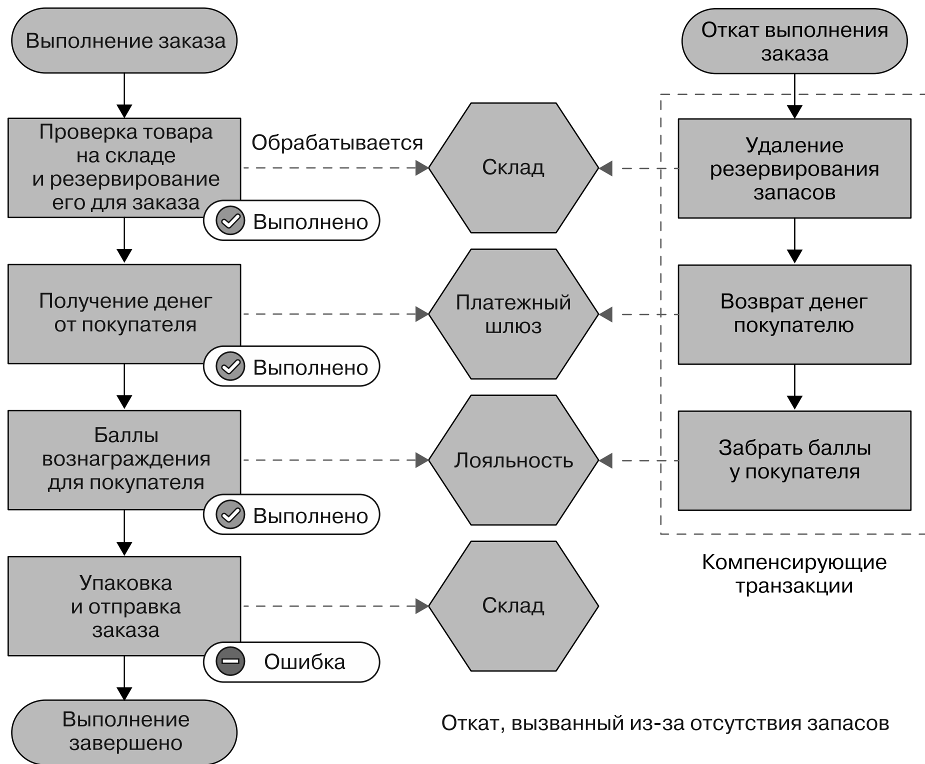


Рис. 6.7. Запуск отката всей саги

Вполне уместно, чтобы информация, относящаяся к откату, сохранялась в системе. Возможно, вы захотите сохранить запись в сервисе `Заказ` для отмененного заказа вместе с информацией о произошедшем по целому ряду причин.

Упорядочение шагов рабочего процесса для уменьшения откатов

Можно было бы несколько упростить вероятные сценарии отката на рис. 6.7, перестроив порядок шагов в исходном рабочем процессе. Простым изменением могли бы стать начисление баллов только тогда, когда заказ фактически отправлен, как показано на рис. 6.8.

Таким образом, нам не пришлось бы беспокоиться об отмене этого шага, если возникнут проблемы при попытке упаковать и отправить заказ. Иногда можно упростить операции отката, просто изменив способ выполнения рабочего потока. Перенося вперед те шаги, которые с наибольшей вероятностью приведут к сбою, и завершая процесс раньше, вы избегаете необходимости запускать более поздние компенсирующие транзакции, поскольку эти шаги даже не были запущены на начальном этапе выполнения.

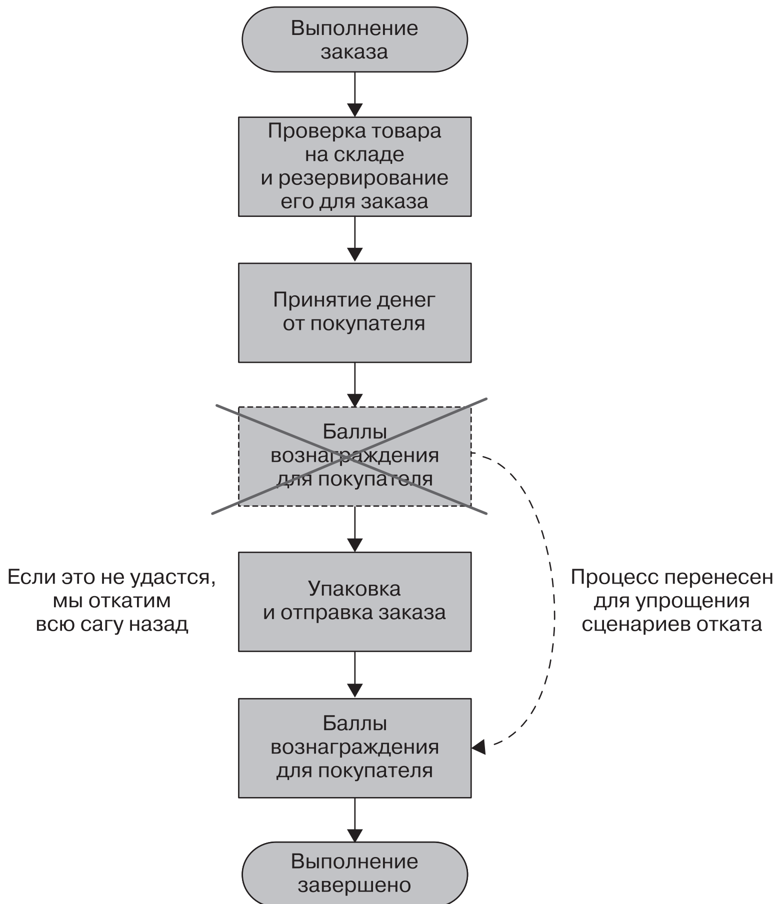


Рис. 6.8. Перемещение шагов на более поздние этапы саги может уменьшить объем отката в случае сбоя

Данные преобразования, если есть возможность их учесть, могут значительно облегчить вашу жизнь, избавив от необходимости даже создавать компенсирующие транзакции для некоторых шагов. Это особенно важно, если осуществление компенсирующей транзакции затруднено. Возможно, у вас получится переместить шаг на более поздний этап, где не может возникнуть потребности в откате.

Смешивание ситуаций с прямым и обратным отказами

Вполне уместно сочетать режимы восстановления после сбоя. Некоторые сбои могут потребовать отката (обратный сбой), другие могут представлять собой прямые сбои. Например, при обработке заказа после получения денег от

покупателя и упаковки товара остается только отправить посылку. Если по какой-либо причине сделать это не получается (возможно, у перевозчика, с которым мы работаем, нет места в фургонах, чтобы принять заказ сегодня), то будет очень странным отменить весь заказ. Достаточно просто повторить отправку (например, поставить ее в очередь на следующий день), и, если это не удастся, потребуется вмешательство человека для разрешения ситуации.

Реализация саг

До сих пор мы рассматривали логическую модель работы саги, но нам нужно пойти немного глубже, чтобы изучить способы реализации самой саги. Рассмотрим два стиля реализации. *Оркестрованные саги* более точно следуют первоначальной задумке и полагаются в первую очередь на централизованную координацию и отслеживание. Их можно сравнить с *хореографическими сагами*, избегающими необходимости централизованной координации в пользу более слабо связанной модели, однако такой подход усложняет отслеживание хода саги.

Оркестрованные саги

В оркестрованных сагах используется центральный координатор (далее *оркестратор*) для определения порядка выполнения и запуска любого требуемого компенсирующего действия. Оркестрованные саги можно рассматривать как командно-контрольный подход: оркестратор контролирует, что происходит и когда, и вместе с этим проявляется хорошая степень наглядности того, что творится с любой конкретной сагой.

Рассматривая процесс выполнения заказа на рис. 6.5, давайте узнаем, как процесс централизованной координации будет работать в виде набора сотрудничающих сервисов, как показано на рис. 6.9.

Здесь центральный `Обработчик заказов`, играющий роль оркестратора, координирует процесс выполнения заказов. Ему известно, какие сервисы необходимы для выполнения операции, и он определяет, когда совершать вызовы этих сервисов. Если вызовы завершаются неудачей, им принимается решение о дальнейших действиях. В целом в оркестрованных сагах, как правило, широко используются взаимодействия «запрос — ответ» между сервисами: `Обработчик заказов` отправляет запрос сервисам (таким как `Платежный шлюз`) и ожидает ответа, чтобы узнать, был ли запрос успешным, и предоставить результаты запроса.

Наличие явного моделирования бизнес-процесса внутри `Обработчика заказов` чрезвычайно выгодно. Это позволяет взглянуть на конкретное место в нашей системе и понять, как процесс должен работать. Это может облегчить адаптацию новых сотрудников и помочь лучше понять основные части системы.

Однако есть несколько недостатков, которые следует учитывать. Во-первых, по своей природе данный подход — с несколько повышенной связанностью.

Сервис Обработчик заказов должен иметь информацию обо всех связанных сервисах, что приводит к более высокой степени доменной связи. Хотя она по своей сути не является плохой лучше бы по возможности свести ее к минимуму. В нашем случае Обработчик заказов должен знать и контролировать так много вещей, что этой формы связанности трудно избежать.

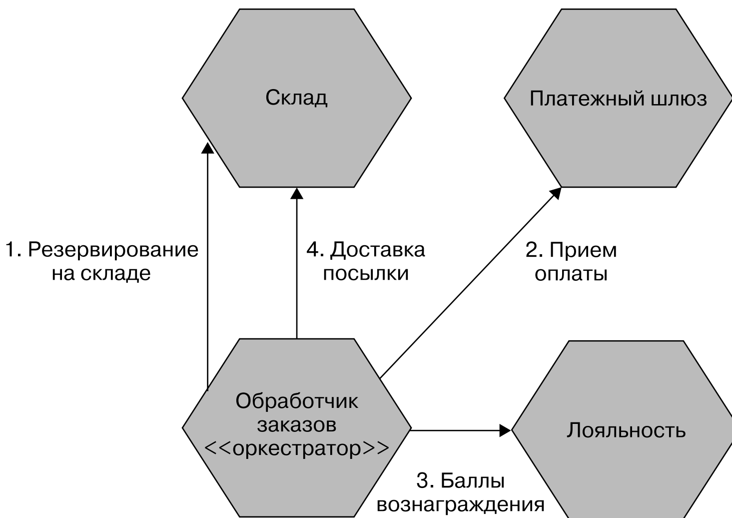
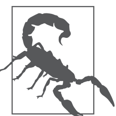


Рис. 6.9. Пример того, как оркестрованную сагу можно использовать для реализации процесса выполнения заказов

Во-вторых, логика, которую необходимо внедрить в сервисы, может начать поглощаться оркестратором. Если подобное произойдет, вы заметите, что ваши сервисы станут слабыми, практически не проявляющими собственного поведения, просто принимающими задачи от оркестраторов, таких как Обработчик заказов. Важно, чтобы вы по-прежнему рассматривали сервисы, составляющие эти организованные потоки, как объекты, имеющие свое собственное локальное состояние и поведение. Они отвечают за свои собственные локальные конечные автоматы.



Если у логики есть место возможной централизации, она станет централизованной!

Один из способов избежать чрезмерной централизации при использовании оркестрованных потоков — обзавестись разными сервисами, играющими роль оркестратора для различных потоков. У вас может быть микросервис Обработчик

`заказов`, обрабатывающий размещение заказа, микросервис `Возврат` для управления процессами возврата товаров и возмещения средств, микросервис `Получение товара`, занимающийся новыми поступлениями товаров, размещаемых на стеллажах склада, и т. д. Что-то вроде нашего микросервиса `Склад` может использоваться всеми этими оркестраторами. Такая модель упрощает сохранение функциональности в самом микросервисе `Склад`, позволяя повторно использовать функциональность во всех этих потоках.

ИНСТРУМЕНТЫ BPM

Инструменты моделирования бизнес-процессов (business process modeling, BPM) доступны уже много лет. По большому счету, они предназначены для определения разработчиками потоков бизнес-процессов, в которых часто используются визуальные инструменты drag-and-drop. Идея заключается в том, чтобы программисты создавали строительные блоки этих процессов, а затем те, кто не занимается разработкой, объединяли бы эти строительные блоки в более крупные технологические потоки. Использование таких инструментов, по-видимому, очень хорошо подходит для реализации оркестрованных саг. И действительно, оркестрация процессов — это основной вариант использования инструментов BPM (или, наоборот, использование инструментов BPM приводит к тому, что вам приходится применять оркестрацию).

На основе моего личного опыта выросла стойкая неприязнь к инструментам BPM. Основная причина заключается в том, что центральная идея — что бизнес-процесс будут определять не разработчики — по моему опыту, почти никогда не соответствовала действительности. Инструментарий, предназначенный для «неразработчиков», в конечном счете используется разработчиками, и, к сожалению, эти инструменты часто работают не так, как привыкли программисты, и нередко требуют использования графических интерфейсов для изменения потоков. Потоки, создаваемые таким образом, могут быть трудными (или невозможными) для контроля версий, разработаны без учета тестирования и многое другое.

Если ваши разработчики собираются реализовывать бизнес-процессы, позвольте им применять инструменты, которые они знают и понимают и которые им подходят. В общем, это означает просто позволить им работать! Если вам нужна наглядность того, как был реализован бизнес-процесс или как он функционирует, то гораздо проще спроецировать визуальное представление рабочего процесса из кода, чем использовать его для описания того, как должен работать ваш код.

Предпринимаются усилия по созданию более удобных для разработчиков инструментов BPM. Отзывы о них, похоже, неоднозначны, но некоторым программистам они подошли. Приятно видеть, что люди пытаются улучшить эти фреймворки. Если вы чувствуете необходимость в дальнейшем изучении этих инструментов, рассмотрите Camunda (https://camunda.com) и Zeebe (https://github.com/camunda-cloud/zeebe). Они оба представляют собой фреймворки оркестрации с открытым исходным кодом, ориентированные на разработчиков микросервисов, и они были бы в топе моего списка, если бы я решил, что инструмент BPM подходит для меня.

Хореографические саги

Хореографическая сага направлена на распределение ответственности за функционирование саги между несколькими взаимодействующими сервисами. Если оркестрация — это командно-контрольный подход, то хореографические саги представляют собой архитектуру «доверяй, но проверяй». Как показано в примере на рис. 6.10, в хореографических сагах часто широко используются события для совместной работы между сервисами.

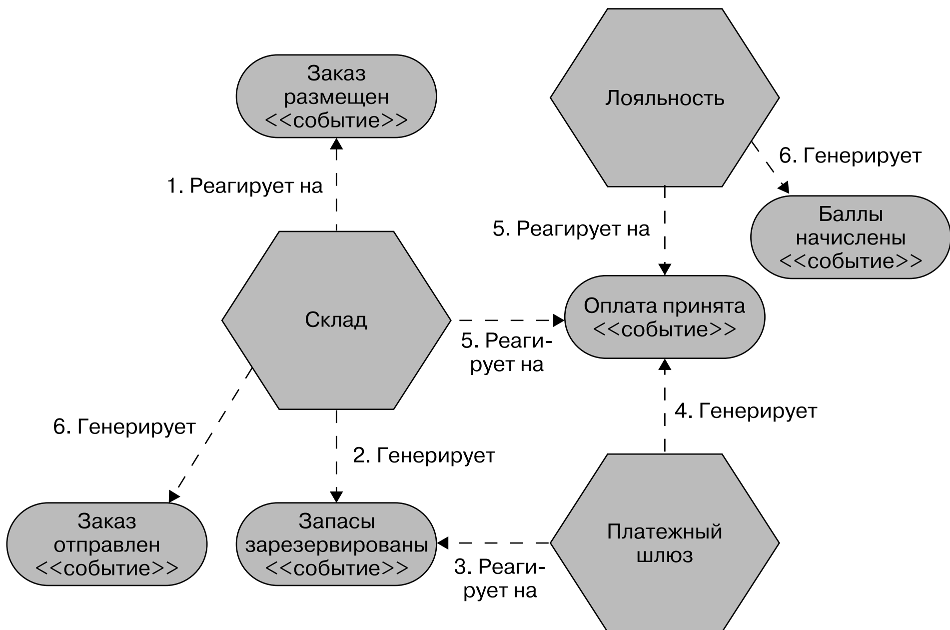


Рис. 6.10. Пример хореографической саги для реализации выполнения заказа

Здесь происходит довольно много процессов, так что давайте по порядку. Эти микросервисы реагируют на принимаемые события. Концептуально события широко представлены в системе, и заинтересованные стороны могут их получать. Вспомните, в главе 4 мы обсуждали, что вы не отправляете события *в* микросервис, а просто запускаете их, и микросервисы, которые заинтересованы в этих событиях, могут получать их и действовать соответствующим образом. В нашем примере, когда сервис `Склад` получает событие `Заказ размещен`, он знает, что его задача — зарезервировать соответствующий запас и после этого запустить события. Если товар не может быть получен, `Склад` должен будет вызвать соответствующее событие (например, событие `Недостаточно запасов`), которое приведет к отмене заказа.

В этом примере мы также видим, как события могут облегчить параллельную обработку. Запуск события `Оплата принята` сервисом `Платежный шлюз` вызывает реакцию в микросервисах `Лояльность` и `Склад`. `Склад` реагирует, отправляя посылку, в то время как микросервис `Лояльность` — начислением баллов.

Как правило, используется какой-то брокер сообщений для управления надежной трансляцией и доставкой событий. Возможно, что несколько микросервисов среагируют на одно и то же событие, и именно здесь вы могли бы использовать топик. Стороны, заинтересованные в определенном типе событий, будут подписываться на определенный топик, не беспокоясь, откуда взялись эти события, а брокер гарантирует долговечность топика и то, что события в нем успешно доставляются подписчикам. В качестве примера у нас может быть сервис `Рекомендации`, также прослушивающий события `Заказ размещен` и использующий их для создания БД музыкальных композиций, которые могут вам понравиться.

В предыдущей архитектуре ни у одного сервиса нет информации о каком-либо другом микросервисе. Им нужно знать только, что делать, когда получено определенное событие — мы резко сократили количество предметных связанностей. По сути, это делает архитектуру гораздо менее связанной. Поскольку реализация процесса здесь разложена и распределена между тремя микросервисами, мы также избегаем рисков централизации логики (если у вас нет места, где логика может быть централизована, тогда она не будет централизованной!).

Однако разобраться в происходящем может быть сложнее, чем кажется на первый взгляд. При применении оркестрации наш процесс был явно смоделирован в оркестраторе. Теперь, с такой архитектурой, какой она представлена, как бы вы построили ментальную модель процесса? Вам пришлось бы изучить поведение каждого сервиса в изоляции и воссоздать эту картину в своей голове — это далеко не просто, даже при таком незамысловатом бизнес-процессе.

Отсутствие четкого представления нашего бизнес-процесса достаточно плохо, но нам также не хватает способа узнать, в каком состоянии находится сага, а это может лишить нас возможности применить компенсирующие действия при необходимости. Можно возложить определенную ответственность за выполнение компенсационных действий на отдельные сервисы, но для некоторых видов восстановления нам необходим способ узнать, в каком состоянии находится сага. Отсутствие единого центра для обсуждения статуса саги — большая проблема, которую можно решить с помощью оркестрации. Как же нам поступить здесь?

Один из самых простых способов — спроецировать представление о состоянии саги, используя передаваемые события. Если сгенерировать уникальный идентификатор для саги, так называемый *идентификатор корреляции*, мы сможем поместить его во все события, генерируемые как часть этой саги. Когда один из сервисов реагирует на событие, идентификатор корреляции извлекается и используется для различных локальных процессов авторизации, а также передается вниз по потоку с любыми дальнейшими вызовами или событиями. Далее

можно организовать сервис, задача которого — просто очистить все эти события и отобразить информацию о том, в каком состоянии находится каждый заказ, и, возможно, программно выполнять действия по решению проблем в рамках процесса выполнения заказа, если другие сервисы не способны сделать это самостоятельно. Я считаю, что некоторая форма идентификатора корреляции необходима для подобных хореографических саг, но эти идентификаторы также имеют большую ценность в более общем плане (подробнее — в главе 10).

Смешивание стилей

Хотя может показаться, что саги с оркестрацией и хореографией — это диаметрально противоположные взгляды на реализацию саг, давайте рассмотрим возможность смешивания и сопоставления моделей. Возможно, в вашей системе есть какие-то бизнес-процессы, которые более естественно вписываются в ту или иную модель. Может существовать одна сага, в которой сочетаются разные стили. В случае выполнения заказа, например, внутри сервиса `Склад` при управлении упаковкой и отправкой заказа допускается использовать оркестрованный поток, даже если первоначальный запрос был сделан как часть более масштабной хореографической саги[1].

Если вы все-таки решите смешивать стили, важно, чтобы у вас все еще было четкое представление, в каком состоянии находится сага и какие действия уже произошли в рамках саги. Без этого понимание режимов сбоев становится сложным, а восстановление после сбоя — затруднительным.

> **ОТСЛЕЖИВАНИЕ ВЫЗОВОВ**
>
> Независимо от того, выбрали ли вы хореографию или оркестрацию, при реализации бизнес-процесса с использованием нескольких микросервисов обычно требуется иметь возможность отслеживать все вызовы, связанные с процессом. Иногда это может быть просто для понимания, правильно ли работает бизнес-процесс, или для диагностирования проблем. В главе 10 рассматриваются такие понятия, как идентификаторы корреляции и агрегирование логов, и то, как они могут помочь в этом отношении.

Стоит ли использовать хореографию или оркестрацию (или смешать их)?

Реализация хореографических саг может принести с собой незнакомые вам и вашей команде идеи. Как правило, они предполагают интенсивное использование событийного взаимодействия, что не получило широкого понимания.

---

[1] Это выходит за рамки данной книги, но Гектор Гарсия-Молина и Кеннет Салем продолжили исследование, как можно «вложить» несколько саг для реализации более сложных процессов. Подробнее об этой теме см.: *Garcia-Molina H.* Modeling Long-Running Activities as Nested Sagas // Data Engineering 14, № 1. Март 1991. 14–18.

Но, по моему опыту, дополнительную сложность, связанную с отслеживанием прогресса саги, почти всегда перевешивают преимущества, касающиеся наличия более слабо связанной архитектуры.

Однако, несмотря на свои личные пристрастия, общий совет, который я даю относительно оркестрации при сравнении ее с хореографией, заключается в том, что я очень спокойно отношусь к использованию оркестрованных саг, когда одна команда управляет реализацией всей саги. В такой ситуации более тесно связанной архитектурой гораздо проще управлять в рамках команды. Если у вас задействовано несколько команд, я предпочитаю более декомпозированную хореографическую сагу, поскольку легче распределить ответственность за реализацию саги между командами, а слабо связанная архитектура позволяет этим командам работать более изолированно.

Стоит отметить, что, как правило, люди с большей охотой тяготеют к вызовам модели «запрос — ответ» с оркестрацией, в то время как хореография интенсивнее использует события. Это не жесткое правило, а просто общее наблюдение. Моя предрасположенность к хореографии, вероятно, обусловлена страстью к моделям событийного взаимодействия: если вам трудно разобраться в применении событийного сотрудничества, хореография, скорее всего, не для вас.

Саги в сравнении с распределенными транзакциями

Распределенные транзакции сопряжены с некоторыми серьезными проблемами, и, за исключением некоторых очень специфических ситуаций, я стараюсь избегать применять их. Пэт Хелланд, пионер в области распределенных систем, так описывает фундаментальные проблемы, возникающие при реализации распределенных транзакций для разрабатываемых сегодня приложений[1].

> В большинстве систем распределенных транзакций отказ одного узла приводит к остановке фиксации транзакции. Это, в свою очередь, приводит к сбою приложения. Чем больше становятся такие системы, тем больше вероятность, что они выйдут из строя. При полете на самолете, которому для работы нужны все его двигатели, добавление двигателя снижает эксплуатационную готовность самолета.

По моему опыту, явное моделирование бизнес-процессов в виде саги позволяет избежать многих проблем, связанных с распределенными транзакциями. В то же время такой подход имеет дополнительное преимущество: то, что в противном случае могло бы быть неявно смоделированными процессами, становится гораздо более явным и очевидным для ваших разработчиков. Превращение основных бизнес-процессов вашей системы в первоклассную концепцию даст множество возможностей.

---

[1] *Helland P.* Life Beyond Distributed Transactions: An Apostate's Opinion // Acmqueue 14, № 5. 12 декабря 2016 года.

Резюме

Итак, как мы видим, путь к внедрению рабочих потоков в нашей микросервисной архитектуре сводится к явному моделированию бизнес-процесса, который мы пытаемся реализовать. Это возвращает нас к идее моделирования аспектов бизнес-области в своей микросервисной архитектуре — явное моделирование бизнес-процессов имеет смысл, если границы микросервисов также определяются в первую очередь в терминах бизнес-области.

Неважно, тяготеете ли вы к оркестрации или хореографии, надеюсь, вы четко знаете, какая модель лучше подойдет для вас.

Если вы хотите изучить эту область более подробно, то в книге «Шаблоны интеграции корпоративных приложений» Грегора Хоупа и Бобби Вулфа есть ряд шаблонов, которые могут быть невероятно полезны при реализации различных типов рабочих процессов (хотя саги *явно* не освещаются)[1]. Я также могу от всей души порекомендовать книгу «Практическая автоматизация процессов» Бернда Рюккера[2]. Книга Бернда больше сосредоточена на оркестрации саг, но это кладезь полезной информации.

Теперь у вас есть представление, как микросервисы могут взаимодействовать и координироваться друг с другом. Но как их создавать? В следующей главе мы рассмотрим, как применять управление версиями, непрерывную интеграцию и непрерывную доставку в контексте микросервисной архитектуры.

---

1 *Хоуп Г., Вулф Б.* Шаблоны интеграции корпоративных приложений. — Вильямс.

2 *Ruecker B.* Practical Process Automation. — O’Reilly, 2021.

ГЛАВА 7

Сборка

Мы потратили много времени на освещение аспектов проектирования микросервисов, теперь немного углубимся в то, как может измениться ваш процесс разработки, чтобы приспособиться к этому новому стилю архитектуры. В следующих главах мы рассмотрим, как следует развертывать и тестировать микросервисы, но перед этим нам необходимо разобраться: что происходит, когда у разработчика есть изменения, готовые к фиксации?

Мы начнем это исследование с рассмотрения некоторых основополагающих концепций — непрерывных интеграции и доставки. Это важные понятия независимо от того, какую системную архитектуру вы используете. Далее мы рассмотрим конвейеры и различные способы управления исходным кодом сервисов.

Краткое введение в непрерывную интеграцию

Непрерывная интеграция (continuous integration, CI) существует всего несколько лет. Тем не менее стоит потратить немного времени на изучение основ, поскольку есть несколько различных вариантов, обязательных к рассмотрению, особенно когда мы рассуждаем о связке между микросервисами, сборками и репозиториями контроля версий.

В CI основная цель — чтобы все синхронизировалось друг с другом, чего мы добиваемся, часто проверяя, правильно ли интегрировался недавно зафиксированный код с существующим. Для этого сервер CI обнаруживает, что код был зафиксирован, проверяет его и осуществляет некоторую верификацию, например, чтобы убедиться, что код компилируется, а тесты проходят. Как минимум ожидается, что эта интеграция будет выполняться ежедневно, хотя на практике я сталкивался с командами, в которых разработчики фактически интегрировали свои изменения несколько раз в день.

В рамках этого процесса мы часто создаем артефакты, применяемые для дальнейшей проверки, например развертывания работающего сервиса для запуска тестов на нем (подробнее о тестировании — в главе 9). В идеале хотелось бы создавать эти артефакты раз и навсегда и использовать их для всех развертываний этой версии кода. Это делается для того, чтобы избежать повторения одинаковых действий и чтобы была возможность подтвердить, что развертываемые артефакты были именно теми блоками, которые мы тестировали. Чтобы эти артефакты можно было использовать повторно, мы помещаем их в какое-либо хранилище, предоставляемое самим инструментом CI, или в отдельную систему.

В главе 9 мы более подробно рассмотрим роль артефактов, а также тестирование.

У CI есть ряд преимуществ. Мы получаем быструю обратную связь о качестве кода благодаря использованию статического анализа и тестирования. CI также позволяет нам автоматизировать создание бинарных артефактов. Весь требуемый для их создания код сам по себе контролируется по версиям, поэтому при необходимости можно воссоздать артефакт. Также можно проследить путь от развернутого артефакта до кода, и, в зависимости от возможностей самого инструмента CI, мы способны увидеть, какие тесты были выполнены для кода и артефакта. Если мы принимаем инфраструктуру как код, допустимо также управлять им и его версиями, повышая прозрачность изменений и упрощая воспроизведение сборок. Именно по этим причинам CI имеет такой успех.

Вы действительно выполняете CI?

CI — это ключевая практика, позволяющая быстро и легко вносить изменения и безболезненно осуществлять переход к микросервисам. Вероятно, вы уже используете инструмент CI в своей организации, но, скорее всего, не по прямому назначению. Я видел, как многие люди путают внедрение инструмента CI с фактическим внедрением CI. Инструмент CI при правильном использовании поможет вам выполнить непрерывную интеграцию, но применение таких инструментов, как Jenkins, CircleCI, Travis, или одного из многих других вариантов не гарантирует, что вы действительно правильно выполняете процесс CI.

Итак, как узнать, действительно ли вы практикуете CI? Мне очень нравятся три вопроса Джеза Хамбла, которые он задает людям, чтобы проверить, действительно ли они понимают, что такое CI, — возможно, было бы интересно задать себе те же вопросы.

Регистрируете ли вы код в главной ветви один раз в день?

Вам нужно убедиться, что ваш код интегрируется. Если вы не будете часто проверять свой код вместе со всеми остальными изменениями, вы в конечном

счете усложните будущую интеграцию. Даже если вы используете недолговечные ветви для управления модификациями, интегрируйте как можно чаще в одну основную ветвь — по крайней мере один раз в день.

Есть ли у вас набор тестов для проверки вносимых изменений?

Без тестов мы просто знаем, что синтаксически наша интеграция сработала, но не понимаем, нарушилось ли поведение системы. CI без проверки на ожидаемое поведение кода — не CI.

Когда сборка нарушена, является ли ее исправление приоритетной задачей для команды?

Удачная зеленая сборка означает, что изменения были надежно интегрированы. Красная сборка означает, что последние преобразования, вероятно, не были интегрированы. Вам нужно остановить все дальнейшие контрольные проверки, которые не связаны с исправлением сборки, чтобы она снова заработала. Если вы позволите накопиться большему количеству изменений, время, необходимое для исправления сборки, резко увеличится. Я работал с командами, где сборка нарушалась на несколько дней, что приводило к значительным трудозатратам для получения удачной сборки.

Модели ветвления

Лишь немногие темы, связанные со сборкой и развертыванием, по-видимому, вызывают столько споров, сколько использование ветвления исходного кода для разработки функций. Ветвление в исходном коде позволяет осуществлять разработку изолированно, не нарушая работу, выполняемую другими. На первый взгляд создание ветви исходного кода для каждой функции, над которой ведется работа, — ветвление функций — кажется полезной концепцией.

Проблема в том, что при работах над веткой функций вы не регулярно интегрируете свои изменения со всем остальным кодом. По сути, возникает *задержка* интеграции. И когда вы наконец решите интегрировать свои изменения в основную часть программы, слияние будет происходить намного сложнее.

Альтернативный подход заключается в том, чтобы все регистрировались в одной и той же магистрали исходного кода. Чтобы изменения не повлияли на других людей, для скрытия незавершенной работы используются такие методы, как флаги функций. Данный метод, при котором все работают с одним и тем же стволом, называется *магистральной разработкой* (Trunk-Based Development, TBD).

Дискуссия на эту тему имеет множество нюансов, но я считаю, что преимущества частой интеграции и ее проверка достаточно значительны, чтобы магистральная разработка была для меня предпочтительным стилем разра-

ботки. Более того, работа по внедрению флагов функций часто оказывается полезной с точки зрения поэтапной доставки. Эту концепцию мы рассмотрим в главе 8.

Будьте осторожны с ветвями

Интегрируйте рано и интегрируйте часто. Избегайте использования долгоживущих ветвей для разработки функций и вместо этого рассмотрите метод магистральной разработки. Если вам действительно нужно использовать ветви, пусть они будут короткими!

Помимо моего собственного анекдотического опыта, появляется все больше исследований, показывающих эффективность сокращения количества ветвей и внедрения магистральной разработки. В отчете компаний DORA и Puppet о состоянии DevOps за 2016 год[1] рассказывается о практике предоставления услуг организациями по всему миру и изучаются методы, которые обычно используются высокопроизводительными командами.

> Мы обнаружили, что наличие ветвей или форков с очень коротким временем жизни (менее суток) перед объединением в магистраль и менее трех активных ветвей в целом — важные аспекты непрерывной доставки, и все они способствуют повышению производительности. То же самое происходит с ежедневным объединением кода в магистраль или главную ветку.

В последующие годы продолжались более глубокое изучение этой темы и поиск доказательств эффективности этого подхода.

В разработке с открытым исходным кодом по-прежнему широко распространен метод, основанный на множественном ветвлении, часто за счет принятия модели разработки GitFlow. Стоит отметить, что разработка с открытым исходным кодом — это не то же самое, что обычная ежедневная разработка. Она характеризуется большим количеством разовых вкладов от временных «ненадежных» коммиттеров, чьи изменения требуют проверки меньшим числом «доверенных» участников. Типичная ежедневная разработка с закрытым исходным кодом обычно выполняется сплоченной командой, все члены которой получают права на фиксацию, даже если они решают принять ту или иную форму процесса проверки кода. Поэтому то, что может подойти для разработки с открытым исходным кодом, не обязательно подойдет для вашей повседневной работы. Даже в отчете за 2019 год[2], развивающем исследование этой темы, были обнаружены

---

[1] *Brown A., Forsgren N., Humble J., Kersten N., Kim G.* State of DevOps Reports. 2016. https://oreil.ly/YqEEh.

[2] *Forsgren N., Smith D., Humble J., Frazelle J.* Accelerate: State of DevOps, 2019. https://oreil.ly/mfkIJ.

некоторые интересные сведения о разработке с открытым исходным кодом и влиянии «долгоживущих» ветвей.

> Результаты наших исследований распространяются на разработку с открытым исходным кодом в некоторых областях:
>
> - чем раньше зафиксировать код, тем лучше: в проектах с открытым исходным кодом многие заметили, что более скорое слияние (merge) исправлений для предотвращения перебазирования (rebase) помогает разработчикам продвигаться быстрее;
> - лучше работать небольшими пакетами: сложнее и дольше проводить слияние крупных патч-бомб, чем мелких, легко читаемых наборов исправлений, поскольку командам обслуживания кода требуется больше времени на просмотр изменений.
>
> Независимо от того, работаете ли вы над кодовой базой с закрытым исходным кодом или над проектом с открытым исходным кодом, недолговечные ветки, небольшие читаемые исправления и автоматическое тестирование изменений делают весь процесс более продуктивным.

Конвейеры сборки и непрерывная доставка

На заре работы с CI я и мои в то время еще коллеги из Thoughtworks осознали ценность сборки в несколько этапов. Тесты — отличный тому пример. У меня может быть много быстрых тестов с нешироким охватом и небольшое количество медленных тестов с широким охватом. Если запустить их все сразу и ждать завершения медленных тестов с большим охватом, мы можем не получить быструю обратную связь, когда быстрые тесты завершатся неудачей. И если быстрые тесты потерпят неудачу, то и нет особого смысла запускать более медленные! Решение этой проблемы заключается в разбиении своей сборки на различные этапы и создании так называемого *конвейера сборки*. Таким образом, может появиться выделенный этап для всех быстрых тестов. Мы запускаем их первыми и, если все они проходят успешно, запускаем отдельный этап для более медленных тестов.

Эта концепция конвейера сборки дает хороший способ отслеживать прогресс разработки ПО по мере прохождения каждого этапа, помогая нам получить представление о качестве производимого продукта. Мы создаем развертываемый артефакт — то, что в итоге будет развернуто в эксплуатацию, и используем его на протяжении всего конвейера. В нашем контексте данный артефакт будет относиться к микросервису, который мы хотим развернуть. На рис. 7.1 показано, как это происходит: один и тот же артефакт используется на каждом этапе конвейера, что дает нам все больше уверенности, что запущенное в эксплуатацию ПО будет работать.

Непрерывная доставка (continuous delivery, CD) основывается на этой и некоторых других концепциях. Как описано в книге Джеза Хамбла и Дэвида Фарли[1], CD — это подход, при котором мы постоянно получаем обратную связь

[1] *Хамбл Дж., Фарли Д.* Непрерывное развертывание ПО: автоматизация процессов сборки, тестирования и внедрения новых версий программ.

о готовности к релизу каждой фиксации изменений, и, кроме того, каждый результат такой фиксации рассматривается как кандидат на релиз.

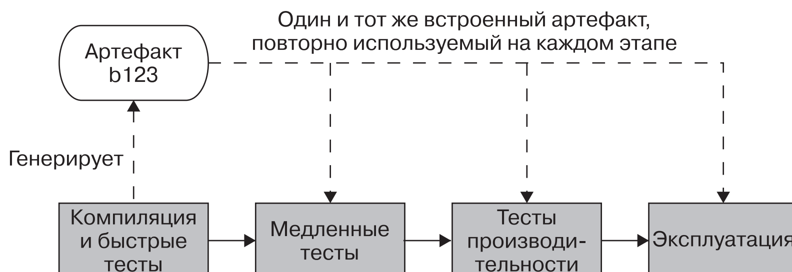


Рис. 7.1. Простой процесс релиза для сервиса Каталог, смоделированный как конвейер сборки

Чтобы полностью охватить эту концепцию, необходимо смоделировать все процессы, связанные с перемещением ПО, от фиксации до выпуска в эксплуатацию, и нужно знать, где находится любая указанная версия программного обеспечения с точки зрения получения разрешения на релиз. В CD мы делаем это, моделируя каждый этап, который наше ПО должно пройти, как вручную, так и автоматизированным образом. Пример такого моделирования показан для сервиса `Каталог` на рис. 7.1. Большинство инструментов CI в настоящее время предоставляют некоторую поддержку определения и визуализации состояния конвейеров сборки, подобных этому.

Если новый сервис `Каталог` проходит все проверки, выполняемые на этапе конвейера, он может перейти к следующему шагу. Если же он не проходит этап, инструмент CI сообщает нам, какие этапы прошла сборка и что не удалось. Если потребуется совершить какие-то действия по исправлению, допускается внести изменения и произвести проверку, позволив новой версии микросервиса попробовать пройти все этапы, прежде чем она станет доступной для развертывания. На рис. 7.2 показан пример такой модели: `сборка-120` потерпела неудачу на этапе быстрого тестирования, `сборка-121` — на тестах производительности, а вот `сборка-122` успешно прошла весь путь до выпуска в эксплуатацию.

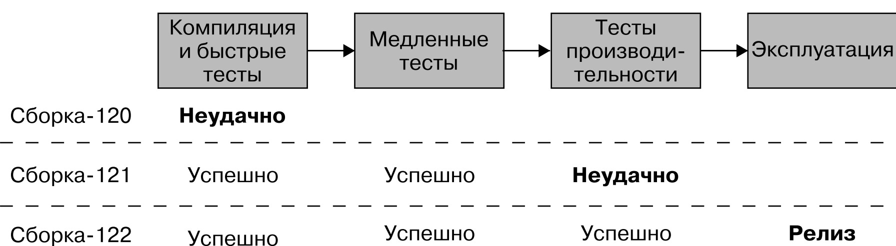


Рис. 7.2. Микросервис Каталог может быть развернут только в том случае, если он проходит каждый этап конвейера

НЕПРЕРЫВНАЯ ДОСТАВКА В СРАВНЕНИИ С НЕПРЕРЫВНЫМ РАЗВЕРТЫВАНИЕМ

Иногда я сталкивался с определенной путаницей в терминах «непрерывная доставка» (CD или CDE) и «непрерывное развертывание» (continuous deployment, CDP), иногда тоже используется обозначение CD. Как мы уже обсуждали, непрерывная доставка — это концепция, при которой каждая фиксация рассматривается как кандидат на релиз и с помощью которой можно оценить качество сборки, чтобы решить, готова ли она к развертыванию. С другой стороны, при непрерывном развертывании все фиксации должны осуществляться с помощью автоматизированных механизмов (например, тестов) и любое ПО, которое проходит эти проверки, развертывается автоматически, без вмешательства человека. Таким образом, непрерывное развертывание можно рассматривать как расширение непрерывной доставки. Без непрерывного развертывания вы *сможете* выполнить непрерывную доставку, но не наоборот!

CDP подходит не всем — многие хотят некоторого физического взаимодействия, чтобы самостоятельно решить, следует ли развертывать ПО, что полностью соответствует модели непрерывной доставки. Однако внедрение CD подразумевает постоянное внимание к оптимизации пути к выпуску в эксплуатацию, а повышенная наглядность облегчает понимание, где следует провести оптимизацию. Часто участие человека в процессе после контрольной проверки становится узким местом, например переход от ручного регрессионного тестирования к автоматизированному функциональному. В результате по мере того, как процесс сборки, развертывания и выпуска автоматизируется все больше и больше, вы обнаружите, что становитесь все ближе и ближе к непрерывному развертыванию.

Инструментарий

В идеале нужен инструмент, охватывающий непрерывную доставку как первостепенную концепцию. Я видел, как многие люди пытались взломать и расширить инструменты CI, чтобы заставить их работать с CD, что часто приводило к созданию сложных систем, которые далеко не так просты в использовании, как инструменты, изначально встроенные в CD. Инструменты, полностью поддерживающие CD, позволяют определять и визуализировать конвейеры, моделируя весь путь к выпуску вашего ПО в эксплуатацию. По мере продвижения версии нашего кода по конвейеру, если она проходит один из этих этапов автоматической проверки, то переходит к следующему этапу.

Некоторые этапы могут выполняться физически. Например, если присутствует процесс приемочного тестирования пользователя (user acceptance testing, UAT), выполняемый вручную, необходимо иметь возможность применить инструмент CD для его моделирования. Я вижу следующую доступную сборку, готовую к развертыванию в среде UAT, и затем развертываю ее, а после, если она проходит проверки вручную, отмечаю этот этап как успешный для перехода к следующему. Если последующий этап автоматизирован, он запустится автоматически.

Компромиссы и среды выполнения

Когда мы перемещаем артефакт микросервиса по конвейеру, наш сервис развертывается в разных средах, которые служат разным целям и могут иметь разные характеристики.

Структурирование конвейера и, следовательно, определение необходимых вам сред само по себе будет балансирующим действием. На ранних стадиях конвейера мы стремимся к получению быстрой обратной связи о готовности ПО к запуску в эксплуатацию. Хочется как можно скорее сообщить разработчикам о наличии проблемы: чем раньше поступит информация о возникшей неполадке, тем быстрее ее можно будет устранить. По мере приближения ПО к выпуску необходимо больше уверенности, что программное обеспечение заработает, и поэтому мы должны развертывать его в условиях, близких к эксплуатационным, — этот компромисс можно изучить на рис. 7.3.

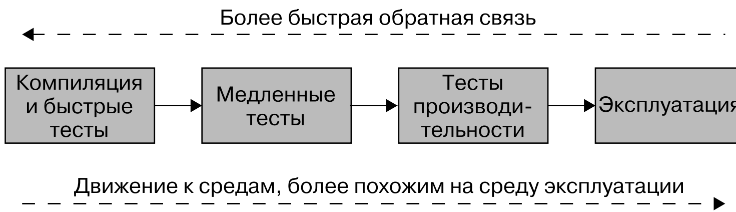


Рис. 7.3. Балансировка конвейера сборки для быстрой обратной связи и запуска в эксплуатационных средах выполнения

Вы получаете самую быструю обратную связь на своем ноутбуке для разработки, но эти результаты далеки от работы в эксплуатационной среде. Вы могли бы развернуть каждую фиксацию в среде, которая точно воспроизводит условия эксплуатации, но это, скорее всего, займет больше времени и будет стоить дороже. Поэтому ключевым моментом остается компромиссное балансирование между быстрой обратной связью и необходимостью создания эксплуатационной среды, максимально приближенной к реальной.

Проблемы, связанные с созданием подобной среды, также стали причиной того, почему все больше людей проводят различные виды тестирования в эксплуатации, включая такие методы, как дымовое и параллельное тестирование. Мы вернемся к этой теме в главе 8.

Создание артефакта

Поскольку микросервис перемещается в разные среды, нам действительно нужно что-то развертывать. Оказывается, существует несколько различных вариантов того, какой тип артефакта развертывания можно использовать. В общем, какой

артефакт вы создадите, будет во многом зависеть от технологии, выбранной для развертывания. Мы подробно рассмотрим это в следующей главе, но я хотел бы дать несколько очень важных советов о том, как создание артефактов должно вписываться в ваш процесс сборки CI/CD.

Чтобы упростить задачу, мы не будем указывать на тип создаваемого артефакта — на данный момент просто рассматривайте его как единый развертываемый двоичный большой объект. Есть два важных правила, которые мы должны учесть. Во-первых, как упоминалось ранее, необходимо создать артефакт раз и навсегда. Создание одного и того же объекта снова и снова — пустая трата времени и вообще вредно для экологии, и теоретически это может привести к проблемам, если конфигурация сборки не будет одинаковой каждый раз. На некоторых языках программирования отличный флаг сборки может заставить ПО вести себя совершенно по-другому. Во-вторых, проверяемый артефакт должен быть именно тем, который вы развертываете! Если вы создаете микросервис, тестируете его, говорите: «Да, он работает», а затем выполняете его повторную сборку для развертывания в среде эксплуатации, как вы узнаете, что проверенное вами ПО — то же самое, которое вы разворачиваете?

Объединив эти две идеи, мы получим довольно простой подход. Создайте свой развертываемый артефакт один раз и навсегда, а в идеале делайте это на ранних этапах конвейера. Обычно я практикую это по окончании компиляции кода (если требуется) и запуска быстрых тестов. После создания данный артефакт хранится в соответствующем репозитории, например Artifactory, Nexus, или, возможно, в реестре контейнеров. Ваш выбор артефакта развертывания, вероятно, продиктует характер хранилища артефактов. Этот же артефакт затем можно использовать для всех последующих этапов конвейера, вплоть до релиза. Итак, возвращаясь к предыдущему конвейеру, на рис. 7.4 видим, что мы создали артефакт для сервиса `Каталог` на первом этапе конвейера, а затем развернули тот же артефакт `сборка-123` как часть медленного тестирования, тестирования производительности и этапа эксплуатации.

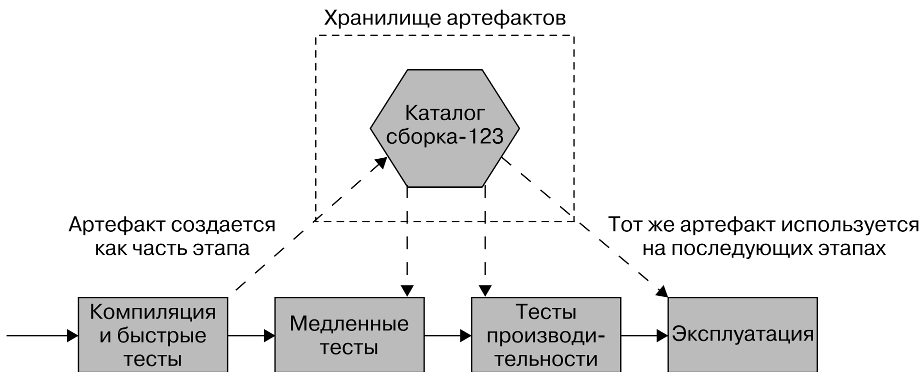


Рис. 7.4. Один и тот же артефакт развертывается в каждой среде

Если один и тот же артефакт будет использоваться в нескольких средах, любые конфигурации, которые варьируются от среды к среде, должны храниться вне самого артефакта. В качестве простого примера я мог бы настроить логи приложений так, чтобы все на уровне `DEBUG` и выше регистрировалось при запуске этапа медленных тестов, что дало бы больше информации для диагностики причины сбоя теста. Однако я мог бы принять решение изменить этот уровень на `INFO`, чтобы уменьшить объем для тестов производительности и развертывания для эксплуатации.

Советы по созданию артефактов

Создайте артефакт развертывания для вашего микросервиса один раз. Используйте его везде, где хотите развернуть эту версию своего микросервиса. Сохраняйте артефакт развертывания независимо от среды — храните конфигурацию, зависящую от среды, в другом месте.

Сопоставление исходного кода и сборок с микросервисами

Мы уже рассмотрели одну тему, которая может вызвать особо горячие споры, — ветвление функций против магистральной разработки. Еще одна тема, которая, вероятно, вызовет разделение мнений, — это организация кода для микросервисов. У меня есть свои предпочтения, но, прежде чем мы перейдем к ним, давайте рассмотрим основные варианты организации кода для микросервисов.

Один гигантский репозиторий — одна гигантская сборка

Если начать с самого простого варианта, то можно свести все воедино. У нас есть единый гигантский репозиторий, в котором хранится весь код, и есть единая сборка, как показано на рис. 7.5. Любая фиксация исходного кода в этом репозитории приведет к запуску сборки, где мы выполним все шаги проверки, связанные со всеми микросервисами, и создадим несколько артефактов, привязанных к одной и той же сборке.

По сравнению с другими подходами, на первый взгляд это кажется намного проще: меньше репозиториев, о которых нужно беспокоиться, и концептуально более простая сборка. С точки зрения разработчика все тоже довольно легко. Я просто выполняю фиксацию кода. Если приходится работать с несколькими сервисами одновременно, необходимо позаботиться только о фиксации.

Эта модель способна отлично работать, если вы согласны с идеей поэтапных релизов, когда вы не возражаете против одновременного развертывания нескольких сервисов. В целом такой модели стоит избегать, но на очень ранней

стадии проекта, особенно если работает только одна команда, эта модель может иметь смысл, но не долго.

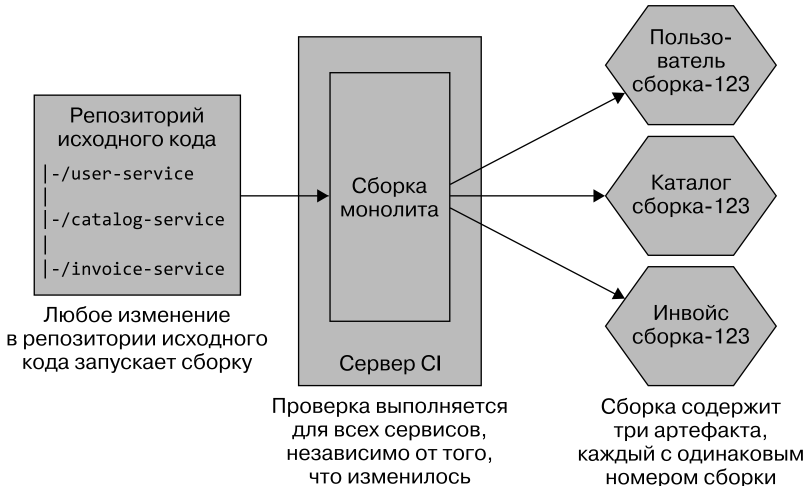


Рис. 7.5. Использование единого репозитория исходного кода и сборки CI для всех микросервисов

Теперь позвольте мне объяснить некоторые существенные недостатки этого подхода. Если я внесу однострочное изменение в один сервис, например изменив поведение в сервисе `Пользователь` на рис. 7.5, *все* остальные сервисы подвергнутся проверке и сборке. Это займет больше времени, чем нужно, а я ожидаю, что тестирование не потребуется. Это влияет на время цикла, скорость, с которой мы можем перенести одно изменение из разработки в жизнь. Однако более тревожным будет незнание, какие артефакты следует или не следует развертывать. Нужно ли мне теперь развертывать все сборки сервисов, чтобы запустить свои небольшие изменения? Трудно сказать. Определить, какие сервисы *действительно* изменились, просто прочитав сообщения о фиксации, очень сложно. Организации, использующие данный подход, часто возвращаются к простому комплексному развертыванию, чего мы на самом деле хотим избежать.

Кроме того, если мое однострочное изменение в сервисе `Пользователь` прерывает сборку, никакие другие изменения в другие сервисы не получится внести, пока это прерывание не будет устранено. И подумайте о сценарии, при котором у вас есть несколько команд, совместно работающих над этой гигантской сборкой. Кто из них главный?

Возможно, такой подход представляет собой форму монорепозитория. Однако на практике большинство реализаций подобных хранилищ сопоставляют несколько сборок с разными частями репозитория, что мы вскоре рассмотрим подробнее. Таким образом, можно рассматривать текущий шаблон

сопоставления одного репозитория с одной сборкой как *наихудшую* форму монорепозитория для тех, кто хочет создать несколько независимо развертываемых микросервисов.

Я почти никогда не видел, чтобы этот подход использовался на практике, за исключением самых ранних стадий проектов. Честно говоря, любой из двух следующих подходов значительно предпочтительнее, поэтому мы сосредоточимся на них.

Шаблон: один репозиторий на один микросервис (то есть мультирепозиторий)

При использовании шаблона одного репозитория для одного микросервиса (более известного как шаблон «Мультирепозиторий») исходный код для каждого микросервиса хранится в его собственном репозитории, как показано на рис. 7.6. Такой подход приводит к прямому сопоставлению изменений исходного кода и сборок CI.

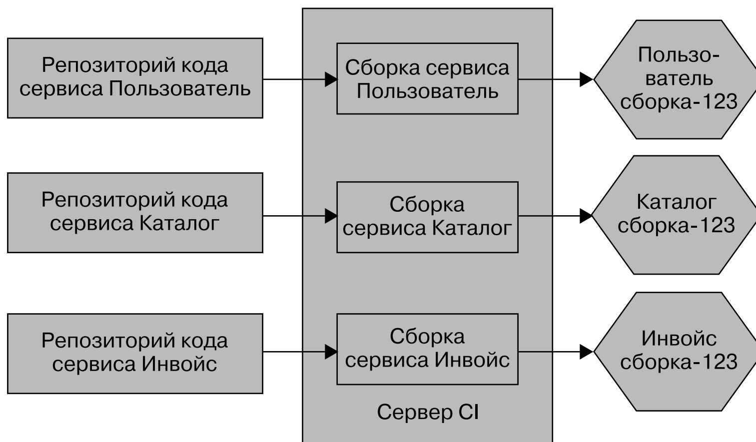


Рис. 7.6. Исходный код для каждого микросервиса хранится в отдельном репозитории

Любое изменение в репозитории исходного кода сервиса `Пользователь` запускает соответствующую сборку, и, если это пройдет, появится новая версия микросервиса `Пользователь`, доступного для развертывания. Наличие отдельного хранилища для конкретного микросервиса также позволяет менять владельца каждого хранилища, что имеет смысл, если вы хотите рассмотреть сильную модель владения для своих микросервисов (подробнее об этом — в ближайшее время).

Однако прямолинейный характер данной модели создает некоторые проблемы. В частности, разработчики могут обнаружить, что работают с несколькими

репозиториями одновременно, что особенно болезненно, если они пытаются вносить в них изменения синхронно. Кроме того, изменения нельзя внести атомарным способом в отдельных репозиториях, по крайней мере с помощью Git.

Повторное использование кода в разных репозиториях

При использовании такого шаблона ничто не мешает микросервису зависеть от другого кода, управляемого из разных репозиториев. Простой механизм реализации заключается в том, что код для повторного применения должен быть упакован в библиотеку, которая затем становится явной зависимостью нижестоящих микросервисов. Пример такого подхода показан на рис. 7.7, где сервисы `Инвойс` и `Зарплата` используют библиотеку `Соединение`.

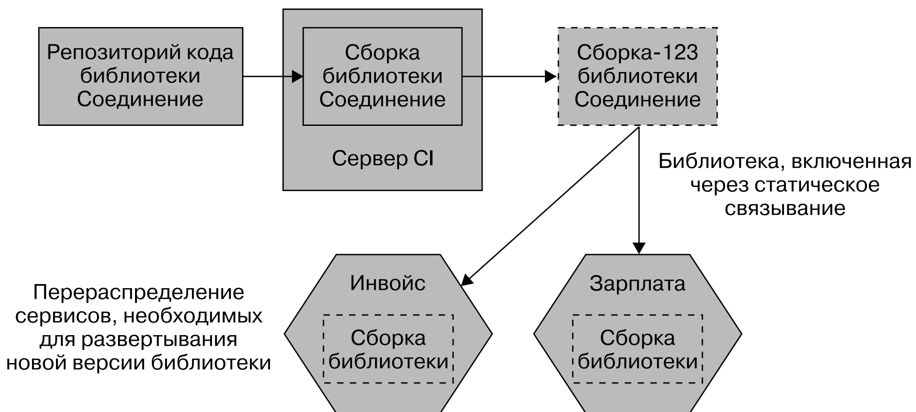


Рис. 7.7. Повторное использование кода в разных репозиториях

Если требуется преобразовать библиотеку `Соединение`, придется внести изменения в соответствующий репозиторий исходного кода и дождаться завершения его сборки, что даст новую версию артефакта. Чтобы фактически развернуть новые версии сервисов `Инвойс` и `Зарплата` с помощью новой вариации библиотеки, необходимо преобразовать используемую ими версию библиотеки `Соединение`. Это может потребовать вмешательства разработчика (если зависимость выстроена от конкретной версии) или такой настройки, чтобы изменение происходило динамически, в зависимости от характера используемого вами инструментария CI. Концепции, лежащие в основе такого подхода, более подробно изложены в книге Джеза Хамбла и Дэвида Фарли[1].

[1] См. раздел «Управление графиками зависимостей» в книге «Непрерывное развертывание ПО: автоматизация процессов сборки, тестирования и внедрения новых версий программ».

Учтите, если вам потребуется развернуть новую версию библиотеки `Соединение`, необходимо будет также развернуть новые сборки сервисов `Инвойс` и `Зарплата`. Помните, что все предостережения, рассмотренные в разделе «DRY и опасности повторного использования кода в мире микросервисов» в главе 5, по-прежнему применимы: если вы решите повторно применить код с помощью библиотек, то необходимо принять тот факт, что эти преобразования не могут быть развернуты атомарным способом, иначе будет подорвана цель независимого развертывания. Также необходимо иметь в виду, что станет сложнее узнать, используют ли некоторые микросервисы определенную версию библиотеки. Это выльется в проблему при попытке отказаться от использования старой версии библиотеки.

Работа с несколькими репозиториями

Итак, как еще можно внести изменения в нескольких репозиториях, помимо повторного использования кода через библиотеки? На рис. 7.8 продемонстрирована моя попытка изменить API, предоставляемый сервисом `Запасы`, а также обновить сервис `Доставка`, чтобы он мог использовать изменения. Если бы код для сервисов `Запасы` и `Доставка` располагался в одном репозитории, у меня была бы возможность зафиксировать код один раз. Но вместо этого приходится разбивать изменение на две процедуры фиксации — одна для сервиса `Запасы` и вторая для `Доставки`.

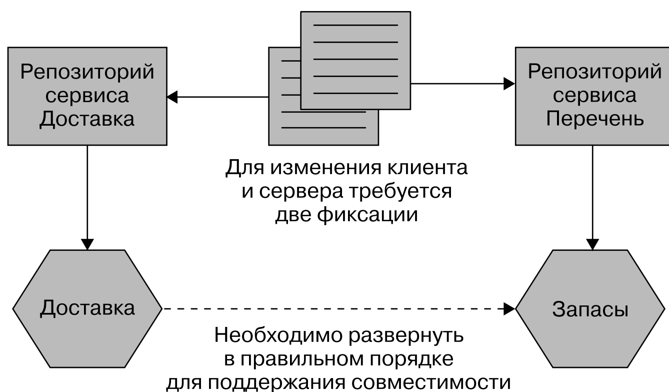


Рис. 7.8. Изменения в границах репозитория требуют нескольких фиксаций

Разделяя эти изменения, мы рискуем вызвать проблемы, если одна фиксация завершится неудачей, а другая сработает. Например, мне требуется внести две модификации, чтобы откатить изменение, и если кто-то за это время успел провести проверку, то это может оказаться непросто. Реальность такова, что

в данной конкретной ситуации хотелось бы в некоторой степени разбить фиксации на этапы. Я бы хотел убедиться, что фиксация для изменения сервиса `Запасы` сработала, прежде чем я изменю какой-либо клиентский код в `Доставке`. Если новая функциональность в API отсутствует, нет смысла вводить использующий ее клиентский код.

Многие люди считают отсутствие атомарного развертывания при этом серьезным недостатком. Я, конечно, могу оценить создаваемую этим сложность, но я думаю, что в большинстве случаев это указывает на более серьезную основную проблему. Если вы постоянно вносите изменения в несколько микросервисов, то границы сервиса могут находиться в неожиданных местах, что, скорее всего, означает слишком высокую связанность между сервисами. Мы же пытаемся оптимизировать архитектуру и границы микросервисов, чтобы преобразования с большей вероятностью применялись в пределах границ. Сквозные изменения должны быть исключением, а не нормой.

На самом деле я бы сказал, что трудности, связанные с работой с несколькими репозиториями, могут быть полезны для обеспечения соблюдения границ микросервиса, поскольку это заставляет вас тщательно подумать, где находятся эти границы, и о характере взаимодействий между ними.

Если вы постоянно вносите изменения в несколько микросервисов, скорее всего, границы микросервиса сместились. Вероятно, стоит рассмотреть возможность объединения микросервисов.

Кроме того, в рамках обычного рабочего процесса возникает необходимость извлекать и отправлять данные, взаимодействуя с несколькими репозиториями. По моему опыту, это можно упростить либо с помощью IDE, поддерживающей несколько репозиториев (с этим могут справиться все IDE, которые я использовал за последние пять лет), либо с помощью написания простых скриптов-оболочек для упрощения работы в командной строке.

Где использовать этот шаблон

Применение подхода «один репозиторий на микросервис» работает одинаково хорошо для небольших и больших команд. Но если вы заметите, что много изменений вносится за пределами микросервиса, то это, скорее всего, не лучший вариант для вас, и шаблон монорепозитория, который мы обсудим далее, станет более подходящим, хотя и повлечет много преобразований. Пересечение границ сервиса в целом рассматривается как предупреждающий знак, что что-то не так. Это также может усложнить повторное использование кода по сравнению с подходом монорепозитория, поскольку придется зависеть от кода, упакованного в артефакты версий.

Шаблон: монорепозиторий

При монорепозиторном подходе код для нескольких микросервисов (или других типов проектов) хранится в одном репозитории исходного кода. Я сталкивался с ситуациями, когда монорепозиторий использовался только одной командой для управления версиями всех своих сервисов, хотя эта концепция была популяризирована некоторыми очень крупными технологическими компаниями, где несколько команд и сотни, если не тысячи разработчиков работают над одним и тем же репозиторием исходного кода.

Храня в одном репозитории весь исходный код, вы позволяете вносить изменения в него в нескольких проектах атомарным способом и обеспечиваете более детализированное повторное применение кода из одного проекта в другой. Google, вероятно, представляет собой самый известный пример компании (но далеко не единственный), использующей подход монорепозитория. Хотя у данного подхода есть и некоторые другие преимущества, такие как улучшенная видимость чужого кода, в качестве основной причины принятия этого шаблона часто упоминается возможность повторного использования кода и внесения изменений, влияющих на множество различных проектов.

Если взять показанный выше пример, где мы пытались модифицировать сервис `Запасы`, чтобы он предоставлял новое поведение, а также обновить сервис `Доставка`, чтобы использовать эту новую раскрываемую функциональность, тогда эти изменения можно внести за одну процедуру фиксации, как показано на рис. 7.9.

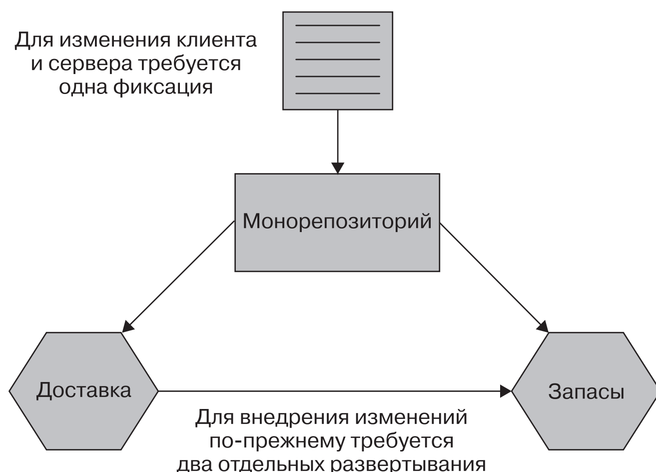
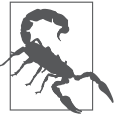


Рис. 7.9. Использование одной фиксации для внесения изменений в двух микросервисах с использованием монорепозитория

Конечно, как и в случае с шаблоном мультирепозиториев, нам все еще необходимо разобраться с развертыванием. Вероятно, придется тщательно продумать его порядок, если требуется избежать поэтапного развертывания.



Атомарные фиксации или атомарное развертывание

Возможность сделать атомарную фиксацию в нескольких сервисах не дает атомарного развертывания. Если возникнет ситуация, когда потребуется изменить код сразу в нескольких сервисах и одновременно запустить его в эксплуатацию, это нарушит основной принцип независимого развертывания. Подробнее об этом см. в разделе «DRY и опасности повторного использования кода в мире микросервисов» в главе 5.

Сопоставление со сборкой

При наличии отдельного репозитория исходного кода для каждого микросервиса сопоставление исходного кода с процессом сборки будет простым. Любое изменение в репозитории может вызвать соответствующую сборку CI. При использовании монорепозитория все становится немного сложнее.

Отправной точкой становится сопоставление папок внутри монорепозитория со сборкой, как показано на рис. 7.10. Например, внесенные в папку `user-service` изменения вызовут процесс сборки сервиса `Пользователь`. Если выполнить проверку кода в измененных файлах в папках `user-service` и `catalog-service`, тогда будут вызваны процессы сборки сервисов `Пользователь` и `Каталог`.

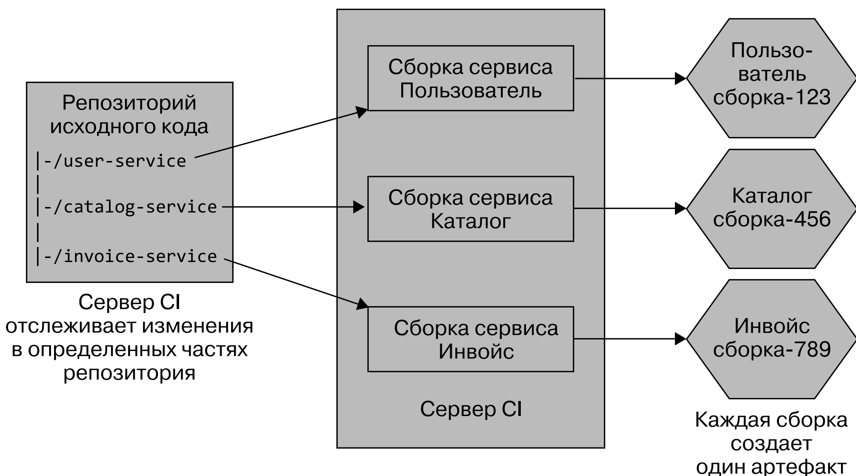


Рис. 7.10. Единый репозиторий исходного кода с подкаталогами, сопоставленными с независимыми сборками

Все становится более запутанным, поскольку структуры папок в репозитории усложнились. В проектах покрупнее можно столкнуться с тем, что несколько разных папок пытаются инициировать одну и ту же сборку, а некоторые папки запускают более одной сборки. В лучшем случае у вас может быть общая папка, используемая всеми микросервисами, а изменение в ней приводит к повторной сборке всех микросервисов. В худшем — командам придется использовать больше инструментов сборки на основе графов, например Bazel (https://bazel.build), чтобы более эффективно управлять этими зависимостями (Bazel — это версия с открытым исходным кодом собственного внутреннего инструмента сборки Google). Внедрение новой системы сборки производится нелегко, но собственный монорепозиторий от Google был бы невозможен без подобных инструментов.

Одним из преимуществ подхода с монорепозиторием является более детализированное неоднократное использование кода в разных проектах. В модели мультирепозиториев, если потребуется повторно применить чужой код, его, скорее всего, придется упаковать как артефакт с версиями, который затем можно будет включить в свою сборку (например, файл JAR, пакеты Nuget или NPM). Поскольку нашей единицей повторного использования является библиотека, мы потенциально используем больше кода, чем нам действительно нужно. Теоретически с применением монорепозитория можно было бы просто зависеть от одного исходного файла из другого проекта, хотя это, конечно, приведет к тому, что мы получим более сложное сопоставление сборки.

Определение владения

При меньших размерах команды и кодовой базы монорепозитории, вероятно, будут хорошо работать с традиционными инструментами сборки и управления исходным кодом, к которым вы привыкли. Однако по мере расширения монорепозитория вам, вероятно, придется присмотреться и к другим типам инструментов. Мы более подробно рассмотрим модели владения в главе 15, но пока — кратко.

Мартин Фаулер ранее писал (https://oreil.ly/nNNWd) о различных моделях владения, описывающих скользящую шкалу владения от *сильного владения* к *слабому* и далее к *коллективному*. С тех пор как Мартин ввел эти термины, практика разработки изменилась, поэтому, возможно, их стоит пересмотреть и переопределить.

При сильном владении некоторый код принадлежит определенной группе людей. Если кто-то из-за пределов этой группы захочет внести изменения, он должен будет попросить владельцев сделать это за него. Слабое владение тоже поддерживает концепцию владельцев, и людям, не входящим в эту группу, разрешается вносить изменения, но только по согласованию с кем-то из

владельцев. Это означает, что запрос на извлечение должен быть отправлен на рассмотрение основной группе владельцев, прежде чем будет выполнено слияние. При коллективном владении любой разработчик может преобразовать любой фрагмент кода.

При небольшом количестве разработчиков (20 или менее, как правило) можно позволить себе применять коллективное владение. Однако, когда у вас появится больше сотрудников, вы, скорее всего, захотите перейти к иной модели владения, чтобы создать более четкие границы ответственности. Это, вероятно, вызовет проблемы у команд, использующих монорепозитории, если их инструмент управления исходным кодом не поддерживает более детальный контроль владения.

Некоторые инструменты работы с исходным кодом позволяют указывать права собственности на определенные каталоги или даже конкретные пути к файлам внутри одного репозитория. Google изначально внедрила эту систему поверх системы Perforce для своего монорепозитория, прежде чем разрабатывать собственную систему управления версиями, и подобный подход поддерживает GitHub (https://oreil.ly/zxmXn) с 2016 года. С помощью GitHub вы создаете файл `CODEOWNERS`, позволяющий сопоставлять владельцев с каталогами или путями к файлам. Некоторые образцы показаны в примере 7.1. Они взяты из собственной документации GitHub и демонстрируют гибкость подобных систем.

Пример 7.1. Примеры того, как определить владение в определенных каталогах в файле CODEOWNERS GitHub

```
# В этом примере @doctocat владеет любыми файлами в каталоге build/logs
# в корне репозитория и любыми из его
# подкаталогов.
/build/logs/ @doctocat

# В этом примере @octocat владеет любым файлом в каталоге приложений
# в любом месте вашего репозитория.
apps/ @octocat

# В этом примере @doctocat владеет любым файлом в каталоге `/docs`
# в корне вашего репозитория.
/docs/ @doctocat
```

Собственная концепция владения кодом GitHub гарантирует, что владельцам кода исходных файлов приходит запрос рассмотрения, когда возникает запрос на извлечение соответствующих файлов. Это станет проблемой при больших запросах на извлечение, так как в итоге вам может потребоваться подтверждение от нескольких рецензентов, но в любом случае есть много веских причин стремиться к менее объемным запросам на извлечение.

Инструментарий

Собственный монорепозиторий от Google огромен, и для его работы в таком масштабе требуется значительный объем инженерных разработок. Рассмотрим такие вещи, как система построения на основе графов, которая пережила несколько поколений, распределенный компоновщик объектов для ускорения времени сборки, плагины для IDE и текстовые редакторы, способные динамически контролировать файлы зависимостей — это огромный объем работы. По мере роста компания Google все чаще сталкивалась с ограничениями в использовании системы Perforce и в итоге была вынуждена создать собственный запатентованный инструмент управления версиями под названием Piper. Когда я работал в Google в 2007–2008 годах, там было более сотни человек, поддерживающих различные инструменты для разработчиков, причем значительная часть их работы была направлена на решение проблем, связанных с последствиями применения подхода монорепозитория. Конечно, такой подход можно оправдать, если у вас есть десятки тысяч инженеров.

Для более подробного обзора причин использования монорепозитория компанией Google я рекомендую статью «Почему Google хранит миллиарды строк кода в одном репозитории» (https://oreil.ly/wMyH3) Рэйчел Потвин и Джоша Левенберга[1]. Мне кажется, что она обязательна к прочтению для тех, кто думает: «Мы должны использовать монорепозиторий, потому что Google так делает!» Ваша организация, вероятно, не Google и, скорее всего, не имеет проблем, ограничений или ресурсов, как у Google. Другими словами, какой бы монорепозиторий вы в итоге ни получили, он, вероятно, не будет такой, как у Google.

Корпорация Microsoft столкнулась с аналогичными проблемами при увеличении масштаба предприятия. Компания приняла Git с целью управления основным репозиторием исходного кода для Windows. Полный рабочий каталог для этой кодовой базы составляет около 270 Гб исходных файлов[2]. Загрузка всего этого заняла бы целую вечность, да и в этом нет необходимости — разработчики в конечном счете будут работать только над одной небольшой частью общей системы. Поэтому Microsoft пришлось создать специальную виртуальную файловую систему VFS для Git (ранее известную как GVFS), гарантирующую, что фактически загружаются только требующиеся разработчику исходные файлы.

VFS для Git — впечатляющее достижение, как и собственный набор инструментов Google, хотя оправдать столь масштабные инвестиции в технологии такого плана гораздо проще для подобных компаний. Также стоит отметить, что,

---

[1] *Potvin R., Levenberg J.* Why Google Stores Billions of Lines of Code in a Single Repository // Communications of the ACM 59, № 7. Июль 2016. 78–87.

[2] Git Virtual File System Design History. https://oreil.ly/SM7d4.

хотя VFS для Git — это система с открытым исходным кодом, я еще не встречал команду за пределами Microsoft, использующую ее. А подавляющая часть инструментария Google, поддерживающей монорепозиторий, имеет закрытый исходный код (Bazel представляет собой заметное исключение, но неясно, в какой степени открытый исходный код Bazel фактически отражает то, что используется внутри Google).

Статья Маркуса Оберлехнера «Монорепозиторий в дикой природе» (Monorepos in the Wild, https://oreil.ly/1SR0A) познакомила меня с Lerna (https://lerna.js.org) — инструментом, созданным командой разработчиков компилятора JavaScript Babel. Lerna предназначен для упрощения создания артефактов с несколькими версиями из одного и того же репозитория исходного кода. Не могу прямо сказать, насколько эффективен Lerna для решения этой задачи (в дополнение к ряду других заметных недостатков, я не являюсь опытным разработчиком JavaScript), но при поверхностном рассмотрении кажется, что он несколько упрощает работу.

Почему «моно» — это моно?

Google не хранит *весь* свой код в монорепозитории. Есть некоторые проекты, особенно разрабатываемые открытым способом, которые хранятся в других местах. Тем не менее на основании ранее упомянутой статьи ACM 95 % кода Google хранилось в монорепозитории по состоянию на 2016 год. В других организациях охват монорепозитория может быть ограничен только одной системой или небольшим количеством систем. Это означает, что компания может создать некоторое количество монорепозиториев для разных подразделений организации.

Я также общался с организациями, практикующими монорепозитории для каждой команды. Хотя с технической точки зрения это, вероятно, не соответствует первоначальному определению этого шаблона (в котором обычно говорится о нескольких командах, совместно использующих один и тот же репозиторий), я все же думаю, что это скорее монорепозиторий, чем что-либо еще. В этой ситуации у каждой команды есть свой собственный монорепозиторий, находящийся полностью под ее контролем. У всех микросервисов, принадлежащих конкретной команде, есть свой код, хранящийся в монорепозитории этой команды, как показано на рис. 7.11.

Для команд, практикующих коллективное владение, эта модель предоставляет много преимуществ, вероятно, перевешивая возможности подхода с монорепозиторием, обходя при этом некоторые проблемы, возникающие при увеличении масштаба системы. Этот промежуточный вариант может иметь большой смысл с точки зрения работы в рамках существующих границ владения в организации. Также он может несколько смягчить опасения по поводу использования этой модели в более крупном масштабе.

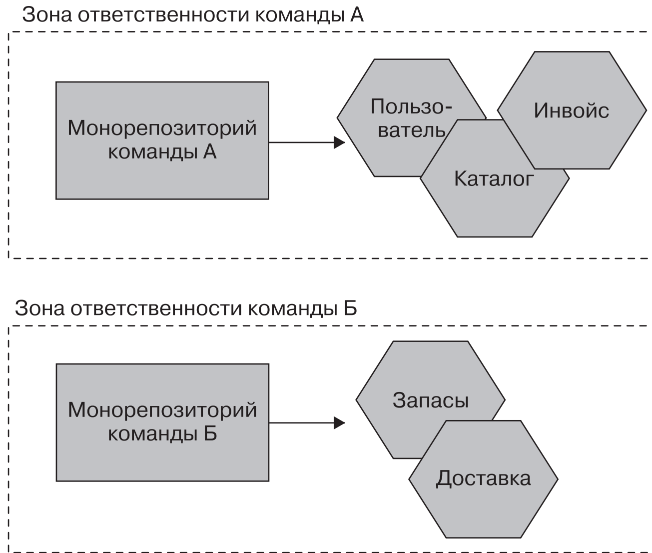


Рис. 7.11. Вариация шаблона, в котором у каждой команды есть свой собственный монорепозиторий

Где использовать этот шаблон

Некоторые организации, работающие в очень больших масштабах, обнаружили, что подход с монорепозиторием — отличный вариант для них. Я уже упоминал Google и Microsoft и могу добавить к ним Facebook, Twitter и Uber. У всех этих организаций есть одна общая черта — это крупные, ориентированные на технологии компании, способные выделять значительные ресурсы, чтобы извлечь максимум пользы из этой модели. Я вижу, что монорепозитории хорошо работают и на другом конце спектра, где число разработчиков и команд поменьше. При наличии всего 10–20 разработчиков легче управлять границами владения и поддерживать простой процесс сборки, используя монорепозиторий. Проблемы чаще возникают у организаций среднего звена, которые разрослись до тех масштабов, когда уже требуются новые инструменты или способы работы для решения проблем, но у этих компаний еще нет лишних ресурсов, чтобы инвестировать в эти идеи.

Какой подход использовал бы я?

По моему опыту, основные преимущества подхода с монорепозиторием — более детализированное повторное использование и атомарные фиксации — похоже, не перевешивают проблем, возникающих при масштабировании. Для небольших команд подойдет любой метод, но по мере масштабирования подход с одним

репозиторием на микросервис (мультирепозитории) более прост. В принципе, я обеспокоен поощрением межсервисных изменений, более запутанными линиями владения и потребностью в новых инструментах, которые могут принести монорепозитории.

Проблема, с которой я неоднократно сталкивался, заключается в том, что организации, начавшие с малого, где коллективное владение (и, следовательно, монорепозиторий) изначально работало хорошо, впоследствии с трудом переходили к другим моделям, поскольку концепция монорепозитория сильно укоренилась. По мере роста организации доставки возрастает количество проблем, связанных с монорепозиторием, а также увеличивается стоимость перехода к альтернативному подходу. Это еще более сложная задача для быстро растущих организаций, поскольку часто проблемы становятся очевидными только после подобного роста, и тогда стоимость перехода на подход с мультирепозиториями выглядит слишком высокой. Это может привести к «ошибке утопленных затрат»: до этого момента вы вкладывали столько средств, чтобы заставить монорепозиторий работать, еще немного инвестиций — и он будет работать так же хорошо, как раньше, верно? Возможно, и нет, но только храбрец может признать, что компания бросает деньги на ветер, и принять решение изменить курс.

Опасения по поводу владения и монорепозиториев можно снять с помощью тонкого контроля владения, но для этого, как правило, требуются инструменты и/или приложение бóльших усилий. Мое мнение по этому поводу может измениться по мере совершенствования инструментария для монорепозиториев, но, несмотря на большую работу, проделанную в отношении разработки инструментов сборки на основе графов с открытым исходным кодом, я по-прежнему наблюдаю очень низкий уровень использования этих цепочек инструментов. Так что мой выбор — мультирепозитории.

Резюме

В этой главе мы рассмотрели несколько важных идей, которые должны помочь вам независимо от того, будете ли вы в итоге использовать микросервисы или нет. Существует множество других аспектов, которые необходимо изучить, от непрерывной доставки до магистральной разработки, от монорепозитория до мультирепозитория. Я предоставил вам множество ресурсов и дополнительную информацию, но нам пора перейти к другой важной теме — развертыванию.

ГЛАВА 8

Развертывание

Развертывание монолитного приложения с одним процессом — довольно простой процесс. Микросервисы с их взаимозависимостью и богатством технологических возможностей — совсем другая история. Когда я писал первое издание, в этой же главе уже было много сказано об огромном разнообразии доступных вариантов развертывания. С тех пор Kubernetes вышел на первый план, а платформы типа «функция как услуга» (function as a service, FaaS) дали нам еще больше поводов подумать о том, как на самом деле стоит поставлять ПО.

Хотя технология, возможно, изменилась за последнее десятилетие, я думаю, что многие из основных принципов, связанных с созданием программного обеспечения, не поменялись. Мне кажется, что тем более важно досконально изучить эти основополагающие идеи, поскольку они могут помочь сориентироваться в мире новых технологий. В данной главе будут освещены некоторые основные принципы, связанные с развертыванием, которые важно понимать, а также мы рассмотрим, как различные доступные вам инструменты помогут (или помешают) в применении этих принципов на практике.

Однако для начала давайте заглянем за занавес и посмотрим, что происходит при переходе от логического представления архитектуры наших систем к реальной топологии физического развертывания.

От логического к физическому

До сих пор при обсуждении микросервисов мы говорили о них в логическом смысле, а не в физическом. Мы могли бы обсудить взаимодействие микросервиса `Инвойс` с микросервисом `Заказ`, как показано на рис. 8.1, без фактического рассмотрения физической топологии развертывания этих сервисов. Логический взгляд на архитектуру обычно абстрагирует от основных проблем физического развертывания — это представление необходимо изменить в рамках данной главы.

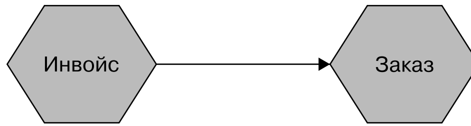


Рис. 8.1. Простое логичное представление двух микросервисов

Подобный логический взгляд на микросервисы может скрывать множество сложностей, когда дело доходит до их фактического запуска в реальной инфраструктуре. Давайте посмотрим, какие детали могут быть скрыты такой диаграммой.

Несколько экземпляров

Топология развертывания двух микросервисов (рис. 8.2) не так проста, как взаимодействие одного элемента с другим. Скорее всего, у нас будет более одного экземпляра каждого сервиса. Наличие нескольких экземпляров позволяет обрабатывать бо́льшую нагрузку, а также теоретически повышает надежность системы, поскольку отказ одного экземпляра переносится легче. Таким образом, у нас потенциально есть один или несколько экземпляров сервиса `Инвойс`, взаимодействующих с одним или несколькими экземплярами сервиса `Заказ`. То, как именно обрабатывается связь между этими экземплярами, будет зависеть от характера механизма связи. Но если предположить, что в этой ситуации используется некоторая форма API на основе HTTP, балансировщика нагрузки будет достаточно для обработки маршрутизации запросов к разным экземплярам. Такой вариант показан на рис. 8.2.

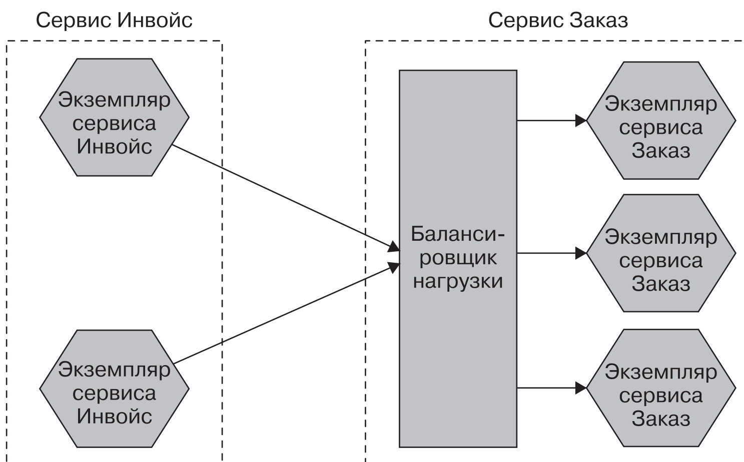


Рис. 8.2. Использование балансировщика нагрузки для сопоставления запросов с конкретными экземплярами сервиса Заказ

Количество необходимых экземпляров будет зависеть от характера приложения — вам нужно будет оценить требуемую избыточность, ожидаемые уровни нагрузки и т. п., чтобы получить приемлемое количество экземпляров. Возможно, вам также потребуется принять во внимание, где будут выполняться эти экземпляры. Если у вас по соображениям надежности есть несколько экземпляров сервиса, вы, вероятно, захотите убедиться, что все они не находятся на одном и том же аппаратном обеспечении. В дальнейшем может потребоваться, чтобы имеющиеся разные экземпляры были распределены не только по нескольким машинам, но и по разным центрам обработки данных (data center, DC), что позволит обеспечить защиту от блокировки всего дата-центра. Это может привести к созданию топологии развертывания, подобной той, что показана на рис. 8.3.

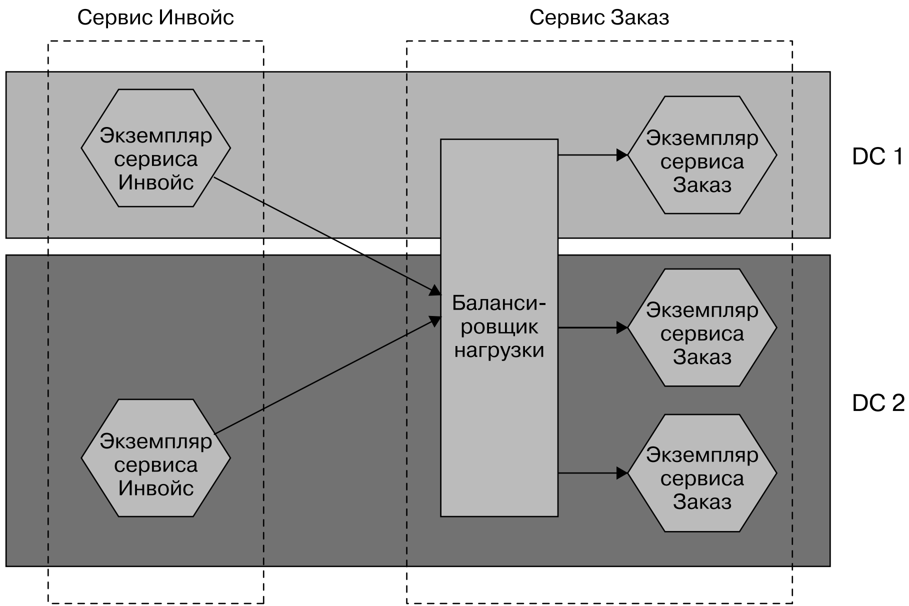


Рис. 8.3. Распределение экземпляров по нескольким различным центрам обработки данных

Каковы шансы, что весь центр обработки данных окажется недоступным? Что ж, для каждой ситуации по-разному, но, по крайней мере когда вы работаете с основными поставщиками облачных услуг, абсолютно точно это стоит учитывать. Когда дело доходит до чего-то вроде управляемой виртуальной машины, ни AWS, ни Azure, ни Google не предоставят вам SLA всего для одной машины или для одной зоны доступности (что является ближайшим эквивалентом центра обработки данных для этих поставщиков). На практике это означает, что любое развертываемое вами решение должно быть распределено по нескольким зонам доступности.

База данных

Если идти дальше, то есть еще один важный компонент, который мы до сих пор игнорировали, — база данных. Как уже обсуждалось, нам требуется, чтобы микросервис скрывал свое внутреннее управление состоянием, поэтому любая используемая в данном ключе БД считается скрытой внутри микросервиса. Это приводит к часто повторяемой мантре «не используйте базы данных совместно», аргументы в пользу которой, я надеюсь, к настоящему времени уже достаточно обоснованы.

Но как это работает, если учесть, что у нас несколько экземпляров микросервиса? У каждого экземпляра должна быть своя отдельная база данных? Одним словом — нет. В большинстве случаев при обращении к любому экземпляру сервиса `Заказ` требуется получать информацию о конкретном заказе. Поэтому нам нужна некоторая степень общего состояния между различными экземплярами одного и того же логического сервиса. Этот принцип показан на рис. 8.4.

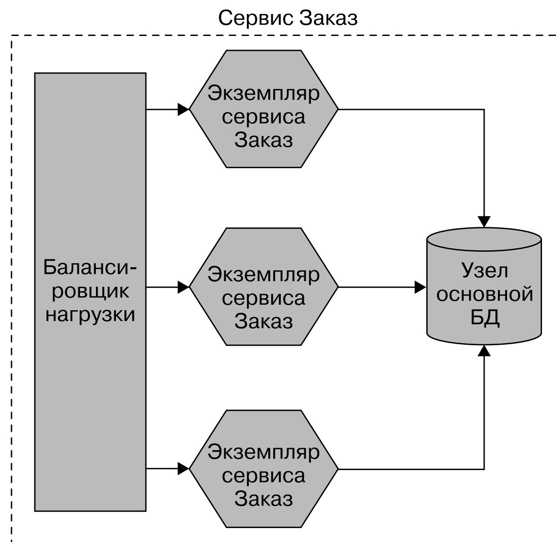


Рис. 8.4. Несколько экземпляров одного и того же микросервиса могут совместно использовать базу данных

Но разве это не нарушает правило «не использовать базы данных совместно»? Не совсем. Одна из основных проблем заключается в том, что при совместном использовании БД различными микросервисами логика, связанная с доступом к этому состоянию и управлением им, теперь распределяется по нескольким микросервисам. Но здесь данные совместно используются разными экземплярами *одного и того же* микросервиса. Логика доступа к состоянию и управлению им по-прежнему хранится в рамках одного логического микросервиса.

Развертывание и масштабирование базы данных

Как и в случае с микросервисами, о базах данных мы до сих пор говорили в основном в логическом контексте. На рис. 8.3 мы проигнорировали любые опасения по поводу избыточности или необходимости масштабирования исходной БД.

В общем и целом развертывание физической базы данных может быть размещено на нескольких машинах по целому ряду причин. Типичным примером может служить разделение нагрузки для чтения и записи между основным узлом и одним или несколькими узлами, предназначенными только для чтения (эти узлы обычно называются репликами чтения). Если бы предстояло реализовать эту идею для сервиса `Заказ`, мы могли бы столкнуться с ситуацией, показанной на рис. 8.5.

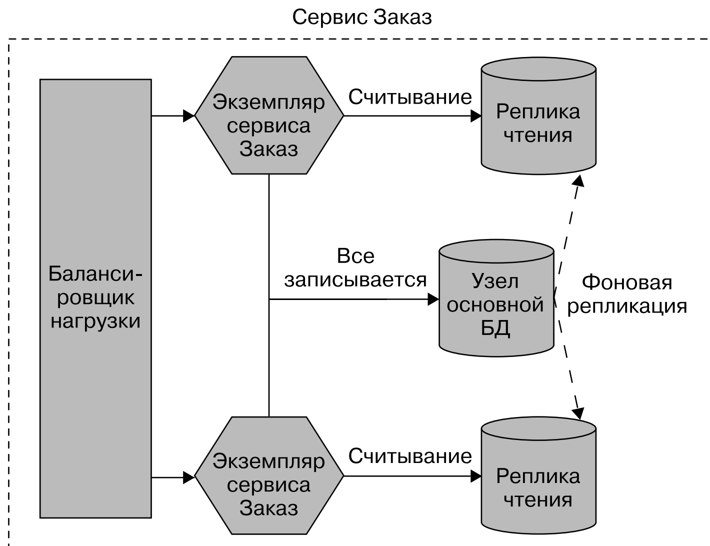


Рис. 8.5. Использование реплик чтения для распределения нагрузки

Весь трафик, доступный только для чтения, направляется на один из узлов реплики, и вы можете дополнительно масштабировать трафик считывания, добавив дополнительные узлы считывания. Из-за того, как работают реляционные базы данных, труднее масштабировать записи, добавляя дополнительные машины (обычно требуются модели сегментирования, что осложняет задачу), поэтому перемещение трафика только для чтения в эти реплики считывания часто может освободить больше места на узле записи, чтобы обеспечить большее масштабирование.

К этой непростой картине добавляется тот факт, что одна и та же инфраструктура базы данных может поддерживать несколько логически изолированных БД. Базы данных для сервисов `Инвойс` и `Заказ` могут быть поданы по одной

и той же базовой СУБД (система управления базами данных) и аппаратной части, как показано на рис. 8.6. Такой подход позволяет объединять аппаратное обеспечение для обслуживания нескольких микросервисов, что снизит затраты на лицензирование, а также поможет сократить работу по управлению самой базой данных.

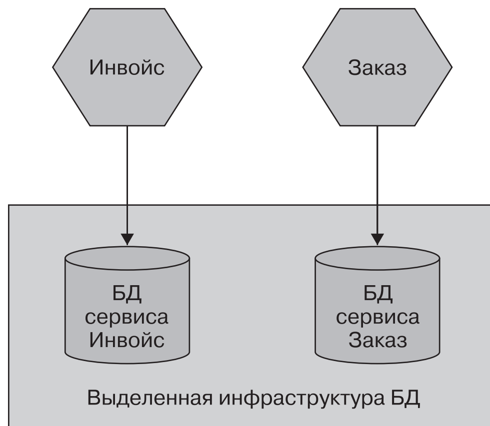


Рис. 8.6. Одна и та же физическая инфраструктура БД, в которой размещены две логически изолированные БД

Здесь важно понимать, что, хотя эти две базы данных могут запускаться с одного и того же оборудования и компонента СУБД, они по-прежнему остаются логически изолированными. Они не мешают друг другу (если вы этого не позволите). Один важный момент, который следует учитывать: если такая инфраструктура общей базы данных выйдет из строя, это повлияет на несколько микросервисов, что может иметь катастрофические последствия.

По моему опыту, организации, управляющие своей собственной инфраструктурой и работающие локально, как правило, с гораздо большей вероятностью будут поддерживать несколько разных баз данных, размещенных в общей инфраструктуре, из-за стоимости. Подготовка оборудования и управление им сопряжены с большими трудностями (по крайней мере, ранее базы данных редко работали в виртуализированной инфраструктуре), поэтому вам стоит свести вероятность работы по этой модели к минимуму.

С другой стороны, команды, работающие на публичных облачных провайдерах, с *высокой* степенью вероятности будут предоставлять выделенную инфраструктуру баз данных для каждого микросервиса, как показано на рис. 8.7. Затраты на создание подобной инфраструктуры и управление ею значительно ниже. Например, сервис реляционных баз данных (Relational Database Service, RDS) от AWS может автоматически решать такие задачи, как резервное копи-

рование, обновление и отказоустойчивость в зонах множественной доступности. Аналогичные продукты существуют и у других поставщиков публичных облачных сервисов. Экономически гораздо эффективнее использовать более изолированную инфраструктуру для вашего микросервиса, предоставляя каждому владельцу микросервиса больше контроля вместо того, чтобы полагаться на общий сервис.

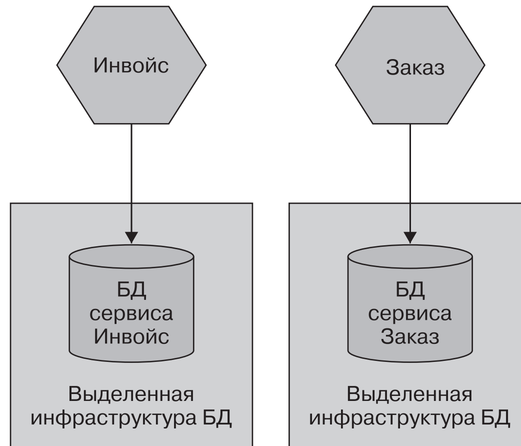


Рис. 8.7. Каждый микросервис использует свою собственную выделенную инфраструктуру БД

Среды выполнения

Когда вы развертываете свое ПО, оно запускается в определенной среде. Каждая среда, как правило, служит разным целям, и точное количество возможно применяемых сред будет сильно различаться в зависимости от способов разработки и развертывания ПО. В одних средах выполнения могут присутствовать эксплуатационные данные, а в других — нет. В каких-то могут быть все сервисы, а в каких-то только небольшое их количество, при этом любые отсутствующие сервисы при тестировании заменяются заглушками.

Обычно программное обеспечение представляется нам проходящим через ряд подготовительных сред, каждая из которых служит определенной цели, позволяя разрабатывать ПО и тестировать его перед запуском в эксплуатацию. Мы исследовали это в подразделе «Компромиссы и среды выполнения» главы 7. От ноутбука разработчика до сервера непрерывной интеграции, интегрированной тестовой среды и многого другого — точный характер и количество ваших сред будет зависеть от множества факторов, но в первую очередь от того, как вы решите разрабатывать программное обеспечение. На рис. 8.8 показан конвейер

для микросервиса `Каталог` системы MusicCorp. Микросервис проходит через различные среды, прежде чем наконец попадет в эксплуатационную среду, где пользователи смогут работать с новым ПО.

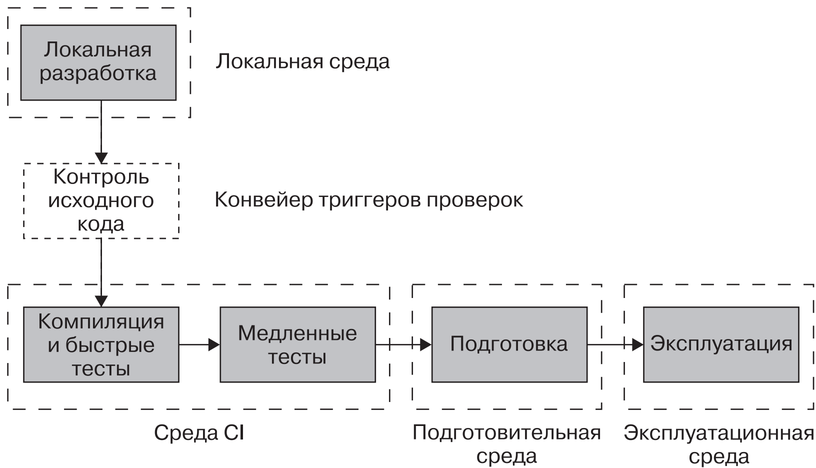


Рис. 8.8. Различные среды, используемые для разных частей конвейера

Первая среда, в которой запускается микросервис, — это то место, где велась работа над кодом до контрольной проверки — возможно, локально. После фиксации кода процесс CI запускается с помощью быстрых тестов. Как быстрые, так и медленные этапы тестирования развертываются в среде CI. Если медленные тесты проходят успешно, микросервис развертывается в подготовительной среде, чтобы обеспечить проверку вручную (что совершенно не обязательно, но все же важно для многих). Если и эта проверка завершается благополучно, микросервис далее развертывается в эксплуатацию.

В идеале каждая среда в этом процессе должна быть точной копией эксплуатационной среды. Это дало бы еще больше уверенности, что наше ПО будет работать, когда поступит клиентам. Однако позволить себе запускать несколько копий всей эксплуатационной среды — очень дорогое удовольствие.

Требуется также настроить среду на ранних этапах, чтобы обеспечить быструю обратную связь. Очень важно, чтобы мы как можно раньше узнали, работает ли наше программное обеспечение, чтобы при необходимости быстро среагировать на поломки. Чем раньше мы узнаем о проблеме, тем быстрее ее исправим и тем меньше последствий от сбоя. Предпочтительнее найти проблему на своем локальном ноутбуке, чем обнаружить ее при тестировании перед выпуском, но точно так же выявление проблемы на предрелизном этапе гораздо лучше, чем обнаружить что-либо после запуска в эксплуатацию (хотя мы рассмотрим некоторые важные компромиссы на эту тему в главе 9).

Это означает, что среды, более близкие к разработчику, будут настроены таким образом, чтобы обеспечить быструю обратную связь, что может поставить под угрозу их «схожесть с эксплуатацией». Однако мы хотим, чтобы они все больше и больше походили на реальные условия эксплуатации, чтобы гарантировать обнаружение проблем.

В качестве простого примера вернемся к нашей предыдущей иллюстрации сервиса `Каталог` и рассмотрим различные среды. На рис. 8.9 на локальном ноутбуке разработчика сервис развернут как единый экземпляр, работающий локально. Программное обеспечение быстро создается, но развертывается как единый экземпляр, работающий на оборудовании, сильно отличающемся от пользовательского. В среде CI развертываются две копии сервиса для тестирования, чтобы убедиться, что логика балансировки нагрузки работает нормально. Мы развертываем оба экземпляра на одной машине — это снижает затраты и ускоряет работу, а также дает достаточную обратную связь на данном этапе процесса.

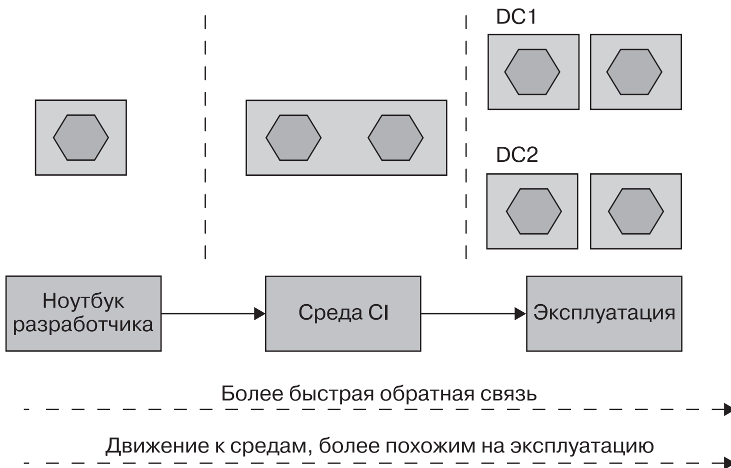


Рис. 8.9. Микросервис может отличаться по способу развертывания в разных средах

Наконец в эксплуатации наш микросервис развертывается в виде четырех экземпляров, распределенных по четырем машинам, которые, в свою очередь, распределены по двум разным дата-центрам.

Это всего лишь пример, как можно использовать среды. Какие настройки понадобятся именно вам, зависит от того, что вы создаете и как происходит развертывание. Например, у вас может быть несколько сред, если требуется развернуть одну копию ПО для каждого клиента.

Ключевым моментом, однако, представляется то, что точная топология микросервиса будет меняться от среды к среде. Поэтому необходимо найти способы изменения количества экземпляров из одной среды в другую, а также

любой конфигурации, зависящей от конкретной среды. Также необходимо создавать экземпляры сервиса только один раз и навсегда, из чего следует, что любая информация, относящаяся к конкретной среде, должна быть отделена от артефакта развернутого сервиса.

То, как вы изменяете топологию своего микросервиса в разных средах, будет во многом зависеть от используемого для развертывания механизма, а также от того, насколько сильно различаются топологии. Если единственное, что меняется от одной среды к другой, — количество экземпляров микросервиса, это может быть так же просто, как параметризация этого значения, чтобы разрешить передачу разных чисел в рамках действия развертывания.

Итак, подведем итог: один логический микросервис допускается развернуть в нескольких средах. В разных средах количество экземпляров одного микросервиса может варьироваться в зависимости от требований конкретной среды.

Принципы развертывания микросервисов

Учитывая, что перед вами так много вариантов развертывания микросервисов, я думаю, важно установить некоторые основные принципы в данной области. Глубокое понимание этих принципов будет полезно независимо от сделанного выбора. Вскоре мы подробно рассмотрим каждый принцип, но для начала приведу основные рассматриваемые идеи.

Изолированное выполнение

Запускайте экземпляры микросервиса изолированным образом, чтобы у них были свои собственные вычислительные ресурсы и их выполнение не могло повлиять на другие экземпляры микросервиса, работающие поблизости.

Сосредоточьтесь на автоматизации

По мере увеличения количества микросервисов автоматизация становится все более важной. Сосредоточьтесь на выборе технологии, обеспечивающей высокую степень автоматизации, и сделайте ее ключевой частью своей культуры производства.

Инфраструктура как код

Представьте конфигурацию вашей инфраструктуры, чтобы упростить автоматизацию и обеспечить обмен информацией. Сохраните этот код в системе управления версиями, чтобы можно было воссоздавать среды.

Развертывание без простоя

Расширьте возможности независимого развертывания и убедитесь, что развертывание новой версии микросервиса может быть выполнено без каких-либо простоев для пользователей вашего сервиса (будь то люди или другие микросервисы).

Управление желаемым состоянием

Используйте платформу, поддерживающую ваш микросервис в определенном состоянии, при необходимости запуская новые экземпляры в случае сбоев или увеличения трафика.

Изолированное выполнение

У вас может возникнуть соблазн, особенно на ранних этапах пути к внедрению микросервисов, просто поместить все экземпляры микросервиса на одну машину (которая может быть физической или виртуальной), как показано на рис. 8.10. Сугубо с точки зрения управления хостом эта модель проще. В мире, в котором одна команда управляет инфраструктурой, а другая — программным обеспечением, рабочая нагрузка команды инфраструктуры часто зависит от количества хостов, которыми ей приходится управлять. Если на одном хосте размещено больше сервисов, рабочая нагрузка на управление узлом не возрастает с увеличением числа сервисов.

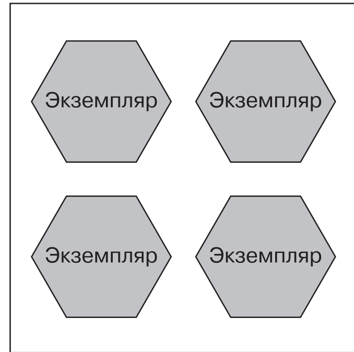


Рис. 8.10. Несколько микросервисов на хост

Однако с данной моделью связаны некоторые проблемы. Первая — это может затруднить мониторинг. Например, при отслеживании работы центрального процессора (ЦП) нужно ли отслеживать ЦП одного сервиса независимо от других? Или стоит проконтролировать ЦП хоста в целом? Избавление от побочных эффектов может оказаться трудной задачей. Если один сервис находится под значительной нагрузкой, это может привести к сокращению ресурсов, доступных другим частям системы. С такой проблемой столкнулся интернет-магазин модной одежды Gilt (https://oreil.ly/yRrWG). Начав с монолита Ruby on Rails, в Gilt решили перейти на микросервисы, чтобы упростить масштабирование приложения, а также лучше приспособиться к растущему числу разработчиков. Первоначально в Gilt много микросервисов сосуществовали в одном блоке, но неравномерная нагрузка на один из микросервисов отрицательно сказывалась на всех остальных. Это также усложнило анализ последствий сбоев хоста — вывод из строя одного хоста может создать мощный эффект домино.

Развертывание сервисов также осложняется, поскольку обеспечить его независимость этого развертывания в данном случае — та еще головная боль. Например, если у каждого микросервиса разные (и потенциально противоречивые) зависимости, которые необходимо установить на общем хосте, как можно заставить это работать?

Такая модель также может препятствовать автономии команд. Если сервисы для разных команд установлены на одном хосте, кто будет настраивать его для их сервисов? По всей видимости, этим займется централизованная команда, а это означает, что для развертывания сервисов потребуется больше координации.

По сути, запуск большого количества экземпляров микросервиса на одной машине (виртуальной или физической) приводит к резкому подрыву одного из ключевых принципов работы микросервисов в целом — независимой развертываемости. Из этого следует, что нам действительно хотелось бы запускать экземпляры микросервиса изолированно, как показано на рис. 8.11.

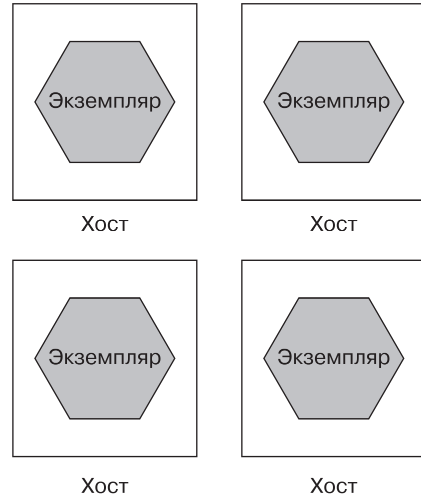


Рис. 8.11. Один микросервис на один хост

Каждый экземпляр микросервиса получает собственную изолированную среду выполнения. Он может устанавливать свои зависимости и иметь набор обособленных ресурсов.

Как выразился мой давний коллега Нил Форд, многие наши методы работы, связанные с развертыванием и управлением хостами, представляют собой попытку оптимизировать работу с учетом нехватки ресурсов. В прошлом, если нам требовалась другая машина для достижения изоляции, единственным вариантом была покупка или аренда еще одной физической машины. Это часто требовало больших затрат времени и приводило к долгосрочным финансовым обязательствам. По моему опыту, клиенты нередко приобретают новые серверы только раз в два-три года и получить дополнительные машины за пределами этих сроков практически нереально. Но вычислительные платформы, доступные по требованию, резко сократили затраты на вычислительные ресурсы, а усовершенствования в технологии виртуализации предлагают большую гибкость даже для инфраструктуры, размещенной внутри компании.

С появлением контейнеризации для создания изолированной среды выполнения появилось больше возможностей, чем когда-либо прежде. Как показано на рис. 8.12, в целом мы переходим от использования дорогих выделенных физических машин, обеспечивающих наилучшую изоляцию, к экономичным и быстрым в реализации контейнерам, предлагающим более слабую изоляцию. Мы вернемся к некоторым особенностям таких технологий, как контейнеризация, позже в этой главе.

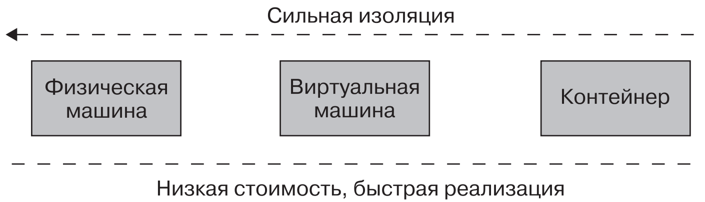


Рис. 8.12. Различные компромиссы в отношении моделей изоляции

При развертывании микросервисов на более абстрактных платформах, например AWS Lambda или Heroku, изоляция была бы обеспечена. В зависимости от характера самой платформы можно ожидать, что экземпляр микросервиса в конечном счете будет работать внутри контейнера или выделенной виртуальной машины незаметно.

В целом изоляция в сфере контейнеров улучшилась в достаточной степени, чтобы сделать их более естественным выбором для рабочих нагрузок микросервисов. Разница в изоляции между контейнерами и виртуальными машинами сократилась до такой степени, что для подавляющего большинства рабочих нагрузок контейнеры «достаточно хороши». Это во многом объясняет, почему они так популярны и почему они, как правило, становятся моим выбором по умолчанию в большинстве ситуаций.

Сосредоточьтесь на автоматизации

По мере добавления новых микросервисов вам придется иметь дело с большим количеством процессов, элементов для настройки, экземпляров для мониторинга. Переход на микросервисы значительно усложняет операционное пространство, и если вы управляете своими операционными процессами в основном вручную, то с увеличением количества сервисов потребуется все больше и больше людей.

Поэтому стоит неустанно уделять внимание автоматизации. Выберите инструментарий и технологию, позволяющие выполнять действия автоматически, в идеале — с перспективой работы с инфраструктурой как с кодом (о чем я вскоре расскажу).

По мере увеличения количества микросервисов автоматизация становится все более важной. Уделите пристальное внимание технологиям, обеспечивающим высокую степень автоматизации, и сделайте ее основной частью своей культуры производства.

Автоматизация — это также способ убедиться, что ваши разработчики попрежнему остаются продуктивными. Предоставление разработчикам возможности самостоятельно предоставлять отдельные сервисы или группы сервисов — ключ к облегчению их рабочего процесса.

Выбранная для обеспечения автоматизации технология начинается с инструментов, используемых для управления хостами. Можете ли вы написать строку кода для запуска или выключения виртуальной машины? Можете ли вы автоматически развернуть написанное вами ПО? Или развернуть изменения в БД без ручного вмешательства? Внедрение культуры автоматизации представляет собой ключевой фактор, если вы хотите держать под контролем сложности микросервисных архитектур.

Два тематических исследования о возможностях автоматизации

Вероятно, будет полезно привести пару конкретных примеров, объясняющих силу хорошей автоматизации. Австралийская компания realestate.com.au (REA) предоставляет списки недвижимости для розничных и коммерческих клиентов в Австралии и других странах Азиатско-Тихоокеанского региона. В течение нескольких лет компания переводила свою платформу в проект на основе распределенных микросервисов. Когда фирма начинала этот путь, ей пришлось потратить много времени на настройку и подбор инструментов для работы с сервисами, чтобы разработчикам было легко получать в свое распоряжение машины, развертывать код и отслеживать сервисы. Это привело к проведению большого количества подготовительных работ перед началом перехода.

За первые три месяца REA смогла ввести в эксплуатацию всего два новых микросервиса, при этом команда разработчиков взяла на себя полную ответственность за всю сборку, развертывание и поддержку сервисов. В течение следующих трех месяцев аналогичным образом заработало от 10 до 15 сервисов. К концу 18-месячного периода у REA было более 70 функционирующих сервисов.

Такого рода закономерность также подтверждается опытом Gilt, о котором я упоминал ранее. Опять же автоматизация, особенно инструменты, помогающие разработчикам, привели к резкому росту использования микросервисов в Gilt. Через год после начала перехода на микросервисы в Gilt существовало около 10 микросервисов. К 2012 году — более 100. А в 2014 году было запущено свыше 450 микросервисов, примерно по три микросервиса на каждого разработчика в Gilt. Такое соотношение микросервисов и разработчиков не представляет собой редкое явление среди организаций, достигших зрелости в использовании микросервисов, и Financial Times является компанией с аналогичным соотношением.

Инфраструктура как код

Продолжим концепцию автоматизации. Инфраструктура как код (infrastructure as code, IAC) — это концепция, при которой ваша инфраструктура настраивается с помощью машиночитаемого кода. Можно определить конфигурацию своего сервиса в файле `chef` или `puppet` или, возможно, написать несколько bash-скриптов для настройки, но какой бы инструмент вы в итоге ни использовали, ваша система может быть приведена в известное состояние с помощью

исходного кода. Возможно, концепцию IAC стоило бы рассматривать как один из способов внедрения автоматизации. Однако я думаю, что допустимо назвать ее чем-то особенным, потому что эта концепция говорит о том, *как* должна осуществляться автоматизация. IAC перенесла концепции разработки ПО в операционное пространство. Определяя инфраструктуру с помощью кода, эту конфигурацию можно контролировать по версиям, тестировать и повторять по желанию. Для получения дополнительной информации по этой теме я рекомендую книгу Кифа Морриса «Инфраструктура как код: динамические системы в эпоху облачных сервисов»[1].

Теоретически можно использовать любой язык программирования для применения идей инфраструктуры в виде кода, но в этой области существуют специализированные инструменты, такие как Puppet, Chef, Ansible и другие, но все они взяли пример с более раннего CFEngine. Это так называемые декларативные инструменты — они позволяют в текстовой форме определить, как должна выглядеть машина (или другой набор ресурсов), а когда эти скрипты применяются, инфраструктура приводится в такое состояние. Более поздние инструменты вышли за рамки настройки компьютера и перешли к изучению способов настройки целых наборов облачных ресурсов — программа Terraform была очень успешной в этой области. И радостно видеть потенциал платформы Pulumi, стремящейся сделать нечто подобное, хотя и позволяющей людям применять обычные языки программирования, а не предметно-ориентированные, часто используемые этими инструментами. AWS CloudFormation и AWS Cloud Development Kit (CDK) представляют собой примеры инструментов, специфичных для конкретной платформы, в данном случае поддерживающих только AWS. Хотя стоит отметить, что, даже если бы я работал только с AWS, я бы предпочел гибкость кроссплатформенного инструмента, например Terraform.

Контроль версий кода вашей инфраструктуры позволяет отследить, кто внес изменения, что нравится аудиторам. Это также облегчает воспроизведение окружающей среды в данный момент времени, что особенно полезно при попытке обнаружить дефекты. Одному из моих клиентов в рамках судебного разбирательства пришлось воссоздать всю работающую систему по состоянию на определенное время несколько лет назад, вплоть до уровней исправлений операционных систем (ОС) и содержимого брокеров сообщений. Если бы конфигурация среды хранилась в системе управления версиями, их работа была бы намного проще — как бы то ни было, в итоге они потратили более трех месяцев на кропотливые попытки повторно собрать зеркальное отображение более ранней эксплуатационной среды, просматривая электронные письма и примечания к выпуску, чтобы попытаться выяснить, что и кем было сделано. Судебное дело, которое шло уже давно, так и не было закрыто к моменту, когда я закончил свою работу с клиентом.

---

[1] *Morris K.* Infrastructure as Code: Dynamic Systems for the Cloud Age. 2 ed. — O'Reilly, 2020.

Развертывание без простоя

Вы, вероятно, уже устали от меня это слышать, но возможность независимого развертывания действительно важна. Однако это также не абсолютное качество. Насколько именно что-то является независимым? До текущей главы мы в первую очередь рассматривали возможность независимого развертывания с точки зрения избегания высокой связанности. Ранее в этой главе мы говорили о важности предоставления экземпляру микросервиса изолированной среды выполнения, чтобы обеспечить ему определенную степень независимости на уровне физического развертывания. Но мы можем пойти дальше.

Реализация возможности развертывания без простоя может стать огромным шагом вперед, позволяющим разрабатывать и развертывать микросервисы. Не применяя развертывание без простоя, я, возможно, буду вынужден координировать свои действия с вышестоящими потребителями при выпуске ПО, чтобы предупредить их о потенциальном отключении.

Сара Уэллс из Financial Times называет возможность развертывания без простоя одним большим преимуществом с точки зрения повышения скорости доставки. Будучи уверенной в том, что релизы не помешают пользователям, компания Financial Times смогла резко увеличить частоту релизов. Кроме того, релиз без простоя гораздо проще выполнить в рабочее время. Помимо того, что это улучшает качество жизни людей, участвующих в выпуске (по сравнению с работой по вечерам и выходным), хорошо отдохнувшая команда, работающая днем, не только с меньшей вероятностью допустит ошибки, но и получит поддержку многих своих коллег, когда понадобится исправить проблемы.

Цель здесь состоит в том, чтобы вышестоящие потребители вообще не замечали, когда вы выпускаете релиз. То, что это возможно, во многом связано с характером вашего микросервиса. Если вы уже используете промежуточное ПО для асинхронной связи между микросервисом и потребителями, это можно тривиально реализовать: отправленные вам сообщения будут доставлены при резервном копировании. Однако если вы используете синхронную связь, это может быть более проблематичным.

Здесь могут пригодиться такие концепции, как последовательные обновления, и это одна из областей, где использование такой платформы, как Kubernetes, значительно облегчает вашу жизнь. При таком обновлении ваш микросервис не отключается полностью до развертывания новой версии, вместо этого количество более старых экземпляров вашего сервиса постепенно сокращается по мере увеличения количества новых, работающих с актуальными версиями вашего ПО. Однако если единственное, что вам нужно, — инструмент для развертывания без простоя, то внедрение Kubernetes, скорее всего, будет огромным излишеством. Что-то простое, например, механизм «сине-зеленого» развертывания (который мы подробнее рассмотрим в подразделе «Отделение развертывания от релиза» далее), работает так же эффективно.

Могут возникнуть дополнительные затруднения с точки зрения решения таких проблем, как долгоживущие соединения и т. п. Безусловно, если вы создаете микросервис с учетом развертывания без простоя, вам, скорее всего, будет гораздо легче, чем если вы возьмете существующую системную архитектуру и попытаетесь приспособить к ней эту концепцию. Независимо от того, сможете ли вы с самого начала реализовать развертывание своих сервисов без простоя или нет, если вы в принципе добьетесь этого, вы, безусловно, оцените такой повышенный уровень независимости.

Управление желаемым состоянием

Управление желаемым состоянием — это возможность указать требования к инфраструктуре, которые вы предъявляете к своему приложению, и поддерживать их без ручного вмешательства. Если работающая система изменяется таким образом, что желаемое состояние больше не поддерживается, базовая платформа предпринимает необходимые шаги для возвращения системы в желаемое состояние.

В качестве простого примера можно указать количество экземпляров, требуемых для микросервиса, а также необходимый этим экземплярам объем памяти и ядер процессора. Некоторая базовая платформа принимает эту конфигурацию и применяет ее, приводя систему в желаемое состояние. Платформа должна, среди прочего, определить, на каких машинах есть свободные ресурсы, которые можно выделить для запуска требуемого количества экземпляров. Как показано на рис. 8.13, если один из этих экземпляров умирает, платформа распознает, что текущее состояние не соответствует желаемому, и предпринимает соответствующие действия, запуская запасной экземпляр.

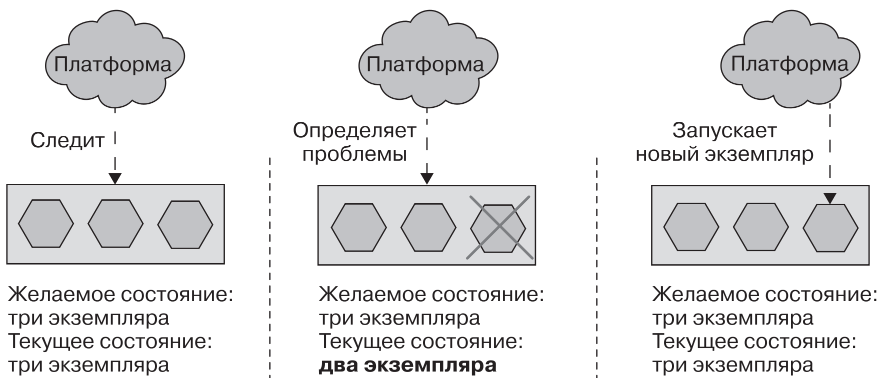


Рис. 8.13. Платформа, обеспечивающая управление желаемым состоянием, запускает новый экземпляр, когда один из них умирает

Вся прелесть управления желаемым состоянием в том, что сама платформа отвечает за поддержку этого самого состояния. Это освобождает как разработчиков, так и операторов от необходимости беспокоиться, как именно все выполняется, — от них просто требуется сосредоточиться на правильном определении желаемого состояния в первую очередь. Это также означает, что в случае возникновения таких проблем, как смерть экземпляра, сбой аппаратного обеспечения или закрытие дата-центра, платформа может решить их без вмешательства человека.

Хотя можно разработать свой собственный набор инструментов для управления желаемым состоянием, обычно используют уже поддерживающую его платформу. Kubernetes — один из таких инструментов. И вы также можете достичь чего-то подобного, используя концепции групп автоматического масштабирования в публичном облаке, например Azure или AWS. Другой платформой, способной обеспечить подобное, является Nomad (https://nomadproject.io). В отличие от Kubernetes, который сосредоточен на развертывании и управлении рабочими нагрузками на основе контейнеров, Nomad содержит очень гибкую модель для осуществления других видов рабочих нагрузок приложений, таких как Java-приложения, виртуальные машины, задачи Hadoop и многое другое. Возможно, стоит рассмотреть такой вариант, если вам требуется платформа для управления смешанными рабочими нагрузками, которая по-прежнему использует управление желаемым состоянием.

Эти платформы владеют информацией о базовой доступности ресурсов и способны сопоставлять запросы на желаемое состояние с имеющимися в наличии ресурсами (или же сообщать, что это невозможно). Как оператор, вы далеки от низкоуровневой конфигурации и можете сказать что-то вроде: «Я хочу, чтобы четыре экземпляра были распределены по обоим центрам данных» — и положиться на свою платформу, зная, что эта задача будет выполнена без вашего участия. Разные платформы обеспечивают разные уровни контроля — вы вольны значительно усложнить определение желаемого состояния.

Использование управления желаемым состоянием иногда может вызвать проблемы, если забыть о нем. Мне вспомнился один случай, когда перед уходом домой я закрывал кластер разработки на AWS. Для уменьшения денежных затрат я отключал экземпляры управляемых виртуальных машин (предоставляемых продуктом EC2 AWS) — их не собирались использовать круглосуточно. Однако при уничтожении одного из экземпляров сразу появился другой. Я не сразу понял, что настроил группу автоматического масштабирования для обеспечения минимального количества машин. AWS видела, как умирает экземпляр, и запускала замену. Мне потребовалось 15 минут такой игры в «ударь крота», прежде чем догадаться, в чем дело. Проблема заключалась в почасовой плате за EC2. Даже если экземпляр работал всего минуту, мы платили за целый час. Так что мои метания в конце дня обошлись мне довольно дорого. В некотором смысле это было признаком успеха (по крайней мере я себя так убеждал):

ранее мы создали группу автоматического масштабирования, и она работала до такой степени хорошо, что мы забыли о ее существовании. Необходимо было всего лишь написать скрипт для отключения группы автоматического масштабирования в рамках завершения работы кластера, чтобы устранить проблему в будущем.

Предварительные условия

Чтобы воспользоваться преимуществами управления желаемым состоянием, платформе нужен какой-то способ автоматического запуска экземпляров микросервиса. Таким образом, наличие полностью автоматизированного развертывания для экземпляров микросервиса представляет собой очевидное предварительное условие для управления желаемым состоянием. Возможно, вам также потребуется тщательно продумать время для запуска экземпляров. Если вы используете управление желаемым состоянием, чтобы обеспечить достаточное количество вычислительных ресурсов для обработки пользовательской нагрузки, то в случае смерти экземпляра необходимо как можно скорее заменить его, чтобы заполнить пробел. Если подготовка нового экземпляра довольно длительная, вам может понадобиться избыточная мощность для обработки нагрузки в случае смерти экземпляра, чтобы иметь достаточно времени для создания новой копии.

Хотя вы могли бы самостоятельно разработать решение для управления состоянием, я считаю это тратой времени. Если вы хотите принять эту концепцию, вам лучше внедрить платформу, рассматривающую ее как первоочередную концепцию. Поскольку это означает знакомство с новой платформой развертывания и всеми связанными с ней идеями и инструментами, вам лучше отложить внедрение управления желаемым состоянием до тех пор, пока у вас не будет запущено хотя бы несколько микросервисов. Это позволит изучить основы микросервисов, прежде чем вы окажетесь перегружены новыми технологиями. Платформы, подобные Kubernetes, действительно помогают, когда требуется управлять множеством процессов, но если у вас всего несколько процессов, требующих внимания, лучше отложить внедрение этих инструментов.

Платформа GitOps

GitOps — довольно молодая разработка компании Weaveworks. Данная платформа объединяет идеи управления желаемым состоянием и IAC. GitOps изначально задумывалась в контексте работы с Kubernetes, и именно на этом сосредоточен соответствующий инструментарий, хотя, возможно, она описывает рабочий поток, который другие использовали и раньше.

В GitOps желаемое состояние вашей инфраструктуры определяется в коде и сохраняется в системе управления версиями. Когда вносятся изменения в это желаемое состояние, некоторые инструменты обеспечивают применение этого

обновленного состояния к работающей системе. Идея состоит в том, чтобы предоставить разработчикам упрощенный рабочий поток для работы с приложениями.

Если вы применяли инструменты настройки инфраструктуры, такие как Chef или Puppet, эта модель покажется вам знакомой. При использовании сервера Chef Server или Puppet Master система была централизованной и способной динамически вводить изменения по мере их появления. Изменение в GitOps заключается в том, что этот инструментарий использует возможности внутри Kubernetes для управления приложениями, а не только инфраструктурой.

Такие инструменты, как Flux (https://oreil.ly/YWS1T), значительно облегчают реализацию подобных идей. Однако хотя инструменты и упрощают переход на иные методы работы, они не могут заставить вас принять новые рабочие подходы. Иными словами, само по себе наличие Flux (или другого инструмента GitOps) не означает, что вы принимаете идеи управления желаемым состоянием или IAC.

Если вы работаете с Kubernetes, внедрение такого инструмента, как Flux, и продвигаемых им потоков работ может значительно ускорить внедрение таких концепций, как управление желаемым состоянием и IAC. Просто убедитесь, что вы не теряете из виду цели, лежащие в основе концепций, и не ослеплены новыми технологиями в этой области!

Варианты развертывания

Когда речь заходит об инструментах и подходах, которые можно использовать для рабочих нагрузок микросервиса, у нас есть *масса* вариантов. Но рассматривать их стоит с точки зрения только что изложенных принципов. Хотелось бы, чтобы микросервисы работали изолированно и в идеале развертывались без простоев. Необходимо, чтобы выбранный инструментарий позволял внедрять культуру автоматизации, определять инфраструктуру и конфигурацию приложений в коде и, как бонус, управлять желаемым состоянием.

Давайте кратко обобщим различные варианты развертывания, прежде чем посмотрим, насколько хорошо они реализуют эти идеи.

Физическая машина

Экземпляр микросервиса развертывается непосредственно на физической машине без виртуализации.

Виртуальная машина

Экземпляр микросервиса развертывается на виртуальной машине.

Контейнер

Экземпляр микросервиса запускается как отдельный контейнер на виртуальной или физической машине. Среда выполнения контейнера может управляться инструментом оркестрации контейнеров, например Kubernetes.

Контейнер приложения

Экземпляр микросервиса запускается внутри контейнера приложения, который управляет другими экземплярами приложения, обычно в той же среде выполнения.

Платформа как услуга (PaaS)

Для развертывания экземпляров микросервисов используется более абстрактная платформа, часто абстрагирующаяся от всех концепций базовых серверов, используемых для запуска микросервисов. Примерами могут служить Heroku, Google App Engine и AWS Beanstalk.

Функция как услуга (FaaS)

Экземпляр микросервиса развертывается как одна или несколько функций, которые запускаются и управляются базовой платформой, такой как AWS Lambda или Azure Functions. Возможно, FaaS — это особый тип PaaS, но он заслуживает изучения сам по себе, учитывая недавно возникшую популярность идеи и вопросы, поднимаемые этой моделью о сопоставлении микросервиса с развернутым артефактом.

Физические машины

Все менее распространенным вариантом становится развертывание микросервисов *непосредственно* на физических машинах. Под «непосредственно» я подразумеваю, что между вами и аппаратным обеспечением нет никаких уровней виртуализации или контейнеризации. Развертывание на физическом оборудовании может привести к снижению загрузки всего вашего аппаратного обеспечения. Однако если на физической машине запущен единственный экземпляр микросервиса, а процессор, память или ввод-вывод (input/output, I/O), предоставляемые оборудованием, используются только наполовину, то оставшиеся ресурсы тратятся впустую. Эта проблема привела к виртуализации большей части вычислительной инфраструктуры, что позволяет сосуществовать нескольким виртуальным машинам на одной физической. Такой подход позволяет значительно повысить эффективность использования инфраструктуры, что дает очевидные экономические преимущества.

Если у вас есть прямой доступ к физическому оборудованию без возможности виртуализации, возникает соблазн разместить несколько микросервисов на одном компьютере. Конечно, это нарушает обсуждавшийся ранее принцип *изолированной среды выполнения* для сервисов. Вы можете использовать такие инструменты, как Puppet или Chef, для настройки машины, помогая реализовать концепцию IAC. Проблема в том, что, если вы работаете только на уровне одной физической машины, реализация таких концепций, как управление желаемым состоянием, развертывание без простоя и т. д., потребует работы на более высоком уровне абстракции. Подобные типы систем

чаще всего используются в сочетании с виртуальными машинами, что мы рассмотрим позже.

В целом прямое развертывание микросервисов на физических машинах почти не встречается в настоящее время, только в случае очень специфических требований (или ограничений). В остальном этот подход не сравнится с повышенной гибкостью, которую может обеспечить виртуализация или контейнеризация.

Виртуальные машины

Виртуализация преобразила центры обработки данных, позволив нам разбить существующие физические компьютеры на более мелкие, но виртуальные машины (ВМ). Традиционная виртуализация, такая как VMware или используемая основными провайдерами облачных услуг управляемая инфраструктура виртуальных машин (например, сервис EC2 от AWS), дала огромные преимущества в повышении эффективности использования вычислительной инфраструктуры при одновременном снижении накладных расходов на управление хостом.

По сути, виртуализация позволяет разделить базовую машину на несколько меньших виртуальных машин, действующих как обычные серверы для ПО, работающего внутри ВМ. Вы можете перенаправить части ресурсов ЦП, памяти, I/O и хранилища каждой виртуальной машине, что в нашем контексте позволяет разместить гораздо больше изолированных сред выполнения для экземпляров микросервисов на одну физическую машину.

Каждая ВМ содержит полноценную операционную систему и набор ресурсов, которые могут использоваться программным обеспечением, запущенным внутри виртуальной машины. Развертывание каждого экземпляра на отдельной ВМ гарантирует очень хорошую степень изоляции между экземплярами. Любой экземпляр микросервиса способен настроить ОС в виртуальной машине в соответствии со своими локальными потребностями. Однако все еще сохраняется проблема аппаратного обеспечения, поддерживающего работу этих виртуальных машин. Если оно выйдет из строя, мы рискуем потерять несколько экземпляров микросервиса. Существуют способы решения данной дилеммы, например управление желаемым состоянием.

Стоимость виртуализации

По мере установки все большего количества виртуальных машин на одном и том же аппаратном обеспечении вы заметите, что получаете все меньшую отдачу с точки зрения вычислительных ресурсов, доступных самим ВМ. Почему так происходит?

Подумайте о нашей физической машине как о ящике для носков. Если положить в ящик много деревянных перегородок, в него поместится больше или меньше носков? Ответ: меньше — сами перегородки тоже занимают место!

Скорее всего, с нашим ящиком стало бы проще работать и организовывать его содержимое, а мы бы даже смогли положить в один из отсеков футболки, но чем больше разделителей, тем меньше общее пространство.

В мире виртуализации есть те же издержки, что и у разделителей ящиков для носков. Чтобы понять, откуда они берутся, давайте посмотрим, как выполняется большая часть виртуализации. На рис. 8.14 показано сравнение двух типов виртуализации. Слева мы видим различные уровни, участвующие в так называемой *виртуализации типа 2*, а справа показана *виртуализация на основе контейнеров*, которую мы рассмотрим подробнее в ближайшее время.

Виртуализация типа 2 — это вид, реализуемый AWS, VMware, vSphere, Xen и KVM (виртуализация типа 1 относится к технологии, при которой ВМ выполняются непосредственно на оборудовании, а не поверх другой ОС). В нашей физической инфраструктуре есть операционная система хоста, в которой мы запускаем нечто называемое *гипервизором*, у которого есть две ключевые задачи. Во-первых, он распределяет ресурсы, такие как процессор и память, из физического хоста на виртуальный. Во-вторых, он действует как уровень управления, позволяя нам управлять самими виртуальными машинами.

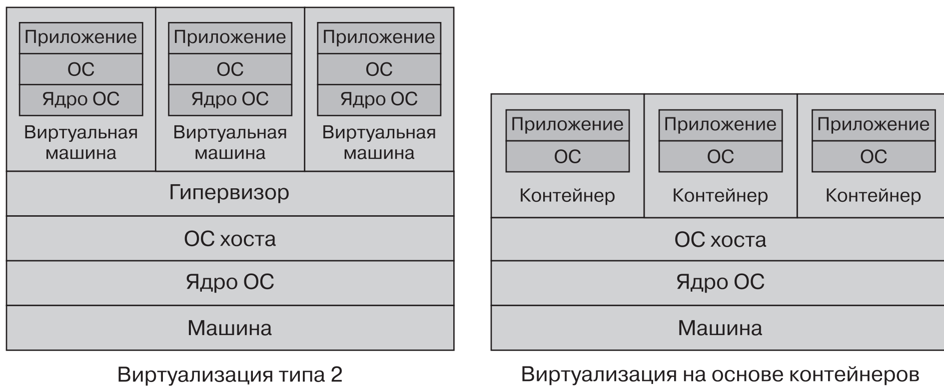


Рис. 8.14. Сравнение стандартной виртуализации типа 2 и облегченных контейнеров

Внутри виртуальных машин мы получаем то, что выглядит как совершенно разные хосты. Они способны запускать собственные операционные системы со своими ядрами. Их можно считать почти герметично запечатанными машинами, которые гипервизор изолирует от базового физического хоста и других ВМ.

Проблема с виртуализацией типа 2 заключается в необходимости гипервизору выделять ресурсы для выполнения своей работы. Это вызывает частичную загрузку процессора, I/O и памяти, которые можно было бы использовать в другом месте. Чем большим количеством узлов управляет гипервизор, тем больше ресурсов ему требуется. В определенный момент эти издержки становятся препятствием для дальнейшего дробления физической инфраструктуры.

На практике оказывается, что отдача от разбиения физического блока на все более мелкие части уменьшается, поскольку накладные расходы гипервизора растут пропорционально.

Хорош ли такой подход для микросервисов

Возвращаясь к нашим принципам, ВМ очень хороши для изоляции, но за все есть своя цена. Простота их автоматизации может варьироваться в зависимости от конкретной используемой технологии. Например, управляемые виртуальные машины в Google Cloud, Azure или AWS легко автоматизировать с помощью API и экосистемы инструментов, основанных на этих API. Кроме того, эти платформы предоставляют такие концепции, как группы автоматического масштабирования, помогая реализовать желаемое управление состоянием. Внедрение развертывания без простоя потребует больше усилий, но, если платформа виртуальной машины предоставляет хороший API, — строительные блоки уже есть в ней. Проблема в том, что многие люди используют управляемые ВМ, которые предоставляются традиционными платформами виртуализации, например VMware. Такие платформы, хотя теоретически и позволяют автоматизировать процесс, обычно не используются в этом контексте. Чаще всего они находятся под централизованным контролем специальной операционной группы, и в результате возможность прямой автоматизации на них может быть ограничена.

Хотя контейнеры в целом становятся все более популярными для рабочих нагрузок микросервисов, многие организации с большим успехом используют виртуальные машины для запуска крупномасштабных микросервисных систем. Netflix, одна из самых популярных микросервисных систем, построила большую часть своих микросервисов на базе управляемых виртуальных машин AWS с помощью EC2. Если вам нужны более строгие уровни изоляции, которые могут обеспечить контейнеры, или у вас нет возможности контейнеризировать свое приложение, виртуальные машины могут стать отличным выбором.

Контейнеры

Со времени выхода первого издания этой книги контейнеры стали доминирующей концепцией в развертывании серверного ПО и для многих кажутся фактическим выбором для упаковки и запуска микросервисных архитектур. Идея контейнера, популяризированная Docker и объединенная с поддерживающей платформой оркестрации контейнеров, такой как Kubernetes, стала для большинства предпочтительным решением в вопросе масштабирования микросервисных архитектур.

Прежде чем мы перейдем к тому, почему это произошло, и к взаимосвязи между контейнерами, Kubernetes и Docker, необходимо сначала понять, что такое контейнер, а также разобраться в разнице между ним и виртуальной машиной.

Изолированный, но по-другому

Контейнеры впервые появились в UNIX-подобных ОС и в течение многих лет были жизнеспособной перспективой только в таких операционных системах, как Linux. Хотя контейнеры в Windows очень популярны, наибольшее влияние контейнеры оказали именно на Linux.

В Linux процессы запускаются определенным пользователем и имеют определенные возможности в зависимости от установленных разрешений. Процессы могут порождать другие процессы. Например, при запуске процесса в терминале он обычно считается дочерним по отношению к терминальному процессу. Работа ядра Linux заключается в поддержании этого дерева процессов, гарантируя, что только разрешенные пользователи могут получить доступ к процессам. Кроме того, ядро Linux способно назначать им ресурсы. Все это представляет собой неотъемлемую часть построения жизнеспособной многопользовательской ОС, где требуется, чтобы действия одного пользователя не поломали всю систему.

Контейнеры, работающие на одном компьютере, используют одно и то же базовое ядро (но есть исключения, о которых мы вскоре поговорим). Вместо того чтобы управлять процессами напрямую, рассматривайте контейнеры как абстракции над поддеревом общего дерева системных процессов, при этом всю тяжелую работу выполняет ядро. Этим контейнерам могут быть выделены физические ресурсы, обрабатываемые ядром. Данный общий подход использовался во многих формах, таких как Solaris Zones и OpenVZ, но именно с LXC эта идея стала основной в операционных системах Linux. Концепция контейнеров Linux получила дальнейшее развитие, когда Docker обеспечил еще более высокий уровень абстракции по сравнению с контейнерами, изначально используя LXC, а затем полностью заменив его.

Если рассмотреть диаграмму стека для хоста, на котором запущен контейнер (см. рис. 8.14), мы увидим несколько отличий в сравнении с виртуализацией типа 2. Во-первых, гипервизор не требуется. Во-вторых, у контейнера, похоже, нет ядра, так как он использует ядро базовой машины. На рис. 8.15 это показано более наглядно. Контейнер способен запускать собственную ОС, но она будет использовать часть общего ядра — именно в нем находится дерево процессов для каждого контейнера. Это означает, что операционная система нашего хоста может запускать Ubuntu, а наши контейнеры — CentOS при условии, что они оба работают как часть одного и того же базового ядра.

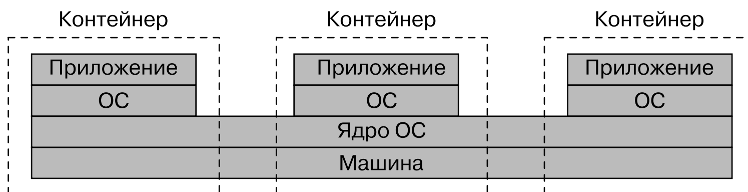


Рис. 8.15. Обычно контейнеры на одном компьютере используют одно и то же ядро

С контейнерами мы выигрываем не только от экономии ресурсов, поскольку гипервизор не требуется, но также за счет обратной связи. Контейнеры Linux предоставляются *гораздо* быстрее, чем полноценные ВМ. Нередко запуск виртуальной машины занимает не одну минуту, в то время как с контейнерами Linux счет идет на секунды. А также есть более тонкий контроль над самими контейнерами с точки зрения распределения ресурсов, что значительно упрощает настройку параметров для получения максимальной отдачи от аппаратного обеспечения.

Из-за более легкой природы контейнеров их можно содержать в гораздо большем количестве при запуске на одном и том же оборудовании, чем это было бы возможно с виртуальными машинами. Развертывая по одному сервису на контейнер, как показано на рис. 8.16, мы получаем определенную степень изоляции от других контейнеров (хотя это и неидеально), и это гораздо экономичнее, чем запускать каждый сервис на отдельной ВМ. Возвращаясь к нашей предыдущей аналогии с ящиками для носков, в контейнерах разделители намного тоньше, чем в виртуальных машинах. Это означает, что большая часть ящика для носков используется для носков.

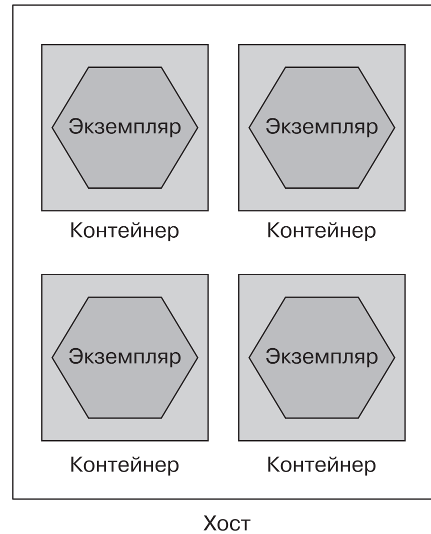


Рис. 8.16. Запуск сервисов в отдельных контейнерах

Контейнеры также можно использовать с полноразмерной виртуализацией (на самом деле это обычное дело). Я видел не один проект, предоставляющий большой экземпляр AWS EC2 и запускающий на нем несколько контейнеров, чтобы получить лучшее из обоих вариантов: эфемерную вычислительную платформу по запросу в форме EC2 в сочетании с очень гибкими и быстрыми контейнерами, работающими поверх нее.

Не идеал

Однако контейнеры Linux не лишены недостатков. Представьте, что у меня есть множество микросервисов, работающих в своих собственных контейнерах на хосте. Как их видят внешние потребители? Нужен какой-то способ направить их к базовым контейнерам, что многие гипервизоры делают за вас при обычной виртуализации. С более ранними технологиями, такими как LXC, с этим приходилось справляться самостоятельно — это одна из областей, где подход Docker к контейнерам очень помог.

Следует также отметить, что эти контейнеры считаются изолированными с точки зрения ресурсов — я могу предоставить каждому контейнеру отдельные наборы ЦП, памяти и т. д. Но это не обязательно та же степень изоляции, которую дают виртуальные машины или, если на то пошло, отдельные физические машины. Раньше существовало несколько задокументированных и известных способов, с помощью которых процесс из одного контейнера мог вырваться наружу и взаимодействовать с другими контейнерами или базовым хостом.

Для решения этих проблем были приложены огромные усилия, а системы оркестрации контейнеров и базовые среды выполнения контейнеров помогли изучить способы запуска рабочих нагрузок контейнеров, чтобы улучшить изоляцию. Лично я рекомендую рассматривать контейнеры как отличный способ изолировать выполнение доверенного ПО. Если вы используете код, написанный другими пользователями, и обеспокоены тем, что злоумышленник пытается обойти изоляцию на уровне контейнера, то вам следует самостоятельно провести более глубокое исследование текущего уровня техники для обработки таких ситуаций. Некоторые из них мы вскоре затронем.

Контейнеры Windows

Исторически сложилось, что пользователи Windows завидовали своим коллегам по цеху из мира Linux, поскольку контейнеров в Windows не было. Однако теперь они стали полноценной частью Windows. Задержка была связана с тем, что подобная функциональность прежде не поддерживалась на уровне ядра операционной системы Microsoft. Но с выходом Windows Server 2016 многое изменилось — технологии контейнеризации стремительно ворвались в мир Windows.

Одним из первых камней преткновения при внедрении контейнеров Windows был размер самой ОС. Помните, что необходимо запустить операционную систему внутри каждого контейнера, поэтому, загружая его образ, вы также загружаете операционную систему. Однако Windows *большая* — настолько большая, что это сделало контейнеры очень тяжеловесными не только с точки зрения размера образов контейнеров, но и из-за ресурсов, необходимых для их запуска.

Microsoft отреагировала на это, создав урезанную операционную систему под названием Windows Nano Server. Идея состояла в том, что у Nano Server должна быть ОС небольшого размера и возможность запускать такие вещи, как экземпляры микросервисов. Наряду с этим Microsoft также развивает более крупное ядро Windows Server, предназначенное для поддержки запуска устаревших приложений Windows в качестве контейнеров. Проблема в том, что эти устройства все еще довольно велики по сравнению с их эквивалентами в Linux — ранние версии Nano Server значительно превышали 1 ГБ. Для сравнения, небольшие операционные системы Linux, такие как Alpine, занимали всего несколько мегабайт.

Хотя Microsoft продолжает попытки уменьшить размер Nano Server, это несоответствие в размерах все еще существует. На практике из-за того, что общие слои в образах контейнеров могут кэшироваться, это не представляет серьезной проблемы.

Особый интерес в мире контейнеров Windows представляет тот факт, что они поддерживают разные уровни изоляции. Стандартный контейнер Windows использует изоляцию процессов, как и его аналоги в Linux. При изоляции процессов каждый контейнер работает в составе одного и того же базового ядра, управляющего изоляцией между контейнерами. С контейнерами Windows у вас также есть возможность обеспечить серьезную изоляцию, запустив контейнеры внутри их собственной виртуальной машины Hyper-V. Это больше похоже на изоляцию при виртуализации, но самое приятное, что разрешается выбирать между Hyper-V и изоляцией процесса при запуске контейнера — образ менять не требуется.

Гибкость при запуске образов в различных типах изоляции имеет преимущества. В некоторых ситуациях модель угроз может диктовать необходимость более сильной изоляции между запущенными процессами, чем простая изоляция на уровне процесса. Например, можно запустить «недоверенный» сторонний код вместе со своими собственными процессами. В такой ситуации очень полезна возможность запускать эти контейнерные рабочие нагрузки в качестве контейнеров Hyper-V. Следует учитывать, что изоляция Hyper-V, скорее всего, с точки зрения времени развертывания и стоимости выполнения будет ближе к обычной виртуализации.

РАЗМЫТЫЕ ЛИНИИ

Растет тенденция к поиску решения, обеспечивающего более надежную изоляцию, как у виртуальных машин, но при этом обладающего легкостью контейнеров. Примерами могут служить контейнеры Microsoft Hyper-V, позволяющие использовать отдельные ядра, и Firecracker (https://oreil.ly/o9rBz), который ошибочно называют виртуальной машиной на основе ядра. Firecracker зарекомендовал себя в качестве детали реализации сервисных предложений, таких как AWS Lambda, где необходима полная изоляция рабочих нагрузок от разных клиентов, но при этом нужно сократить время развертывания и уменьшить операционный след рабочих нагрузок.

Docker

С появлением Docker контейнеризация обрела массовую популярность, хотя до этого применялась в ограниченном количестве. Набор инструментов Docker выполняет большую часть работы с контейнерами. Он управляет подготовкой контейнера, решает некоторые сетевые проблемы и даже предоставляет собственную концепцию реестра, позволяющую хранить приложения Docker. До Docker не существовало концепции образа для контейнеров. Этот аспект, на-

ряду с гораздо более приятным набором инструментов, помог сильно упростить работу с контейнерами.

Абстракция образа Docker полезна, поскольку детали реализации микросервиса скрыты. У нас есть сборки для своего микросервиса, создающие образ Docker в качестве артефакта и сохраняющие его в реестре Docker. При запуске экземпляра образа Docker мы получаем общий набор инструментов для управления им независимо от используемой базовой технологии — микросервисы, написанные на Go, Python, NodeJS или чем-то еще, могут обрабатываться одинаково.

Docker также может устранить некоторые недостатки, связанные с локальным запуском множества сервисов для разработки и тестирования. Раньше я мог использовать такой инструмент, как Vagrant, который давал возможность размещать несколько независимых виртуальных машин на моем рабочем компьютере. Это позволяло бы получить производственную ВМ, запускающую мои экземпляры сервисов локально. Но это довольно ресурсоемкий подход, к тому же я был бы ограничен в количестве доступных для запуска виртуальных машин. Однако сегодня достаточно просто запустить Docker непосредственно на своем компьютере, возможно используя Docker Desktop (https://oreil.ly/g19BE), и создать образ Docker для своего экземпляра микросервиса или извлечь предварительно созданный образ и запустить его локально. Эти образы могут (и должны) быть идентичны образу контейнера, который в итоге будет запущен в эксплуатацию.

Когда Docker впервые появился, его область применения была ограничена управлением контейнерами на одной машине. Но что делать, если вам нужно управлять контейнерами на нескольких машинах? Это важно для поддержания работоспособности системы, если возник сбой в работе компьютера или требуется запустить ряд контейнеров, чтобы справиться с нагрузкой системы. Для решения подобной проблемы компания Docker выпустила два совершенно разных продукта, которые назывались просто Docker Swarm и Docker Swarm Mode (кто сказал, что придумывать названия сложно?). Однако, когда дело доходит до управления большим количеством контейнеров на многих машинах, Kubernetes лидирует. Даже если вы используете инструментарий Docker для создания отдельных контейнеров и управления ими.

Пригодность для микросервисов

Контейнеры как концепция прекрасно подходят для микросервисов, и Docker сделал их значительно более жизнеспособными. Мы получаем изоляцию, но по приемлемой цене, а также возможность скрывать базовые технологии, что позволяет смешивать различные технологические стеки. Но, когда дело доходит до реализации таких идей, как управление желаемым состоянием, нам понадобится что-то вроде Kubernetes.

Kubernetes заслуживает более подробного обсуждения, поэтому мы вернемся к нему чуть позже. А пока просто думайте о нем как о способе управления контейнерами на множестве машин, чего на данный момент достаточно.

Контейнеры приложений

Если вы знакомы с развертыванием приложений .NET в IIS или Java в нечто вроде Weblogic или Tomcat, вам хорошо известна модель, в которой несколько различных сервисов или приложений находятся внутри одного контейнера приложений, который, в свою очередь, расположен на отдельном хосте, как показано на рис. 8.17. Идея заключается в том, что контейнер приложений, содержащий ваши сервисы, дает преимущества в виде улучшенной управляемости, например поддержку кластеризации для группировки нескольких экземпляров вместе, инструменты мониторинга и т. п.

Такой вариант также может помочь сократить издержки на языковую среду выполнения. При запуске пяти Java-сервисов в одном контейнере Java-сервлетов накладные расходы потребуются только на одну JVM. Сравните это с запуском пяти независимых виртуальных машин на хосте при использовании контейнеров. Тем не менее я по-прежнему считаю, что у таких контейнеров приложений достаточно минусов, и вам следует хорошенько подумать, действительно ли они необходимы в вашей системе.

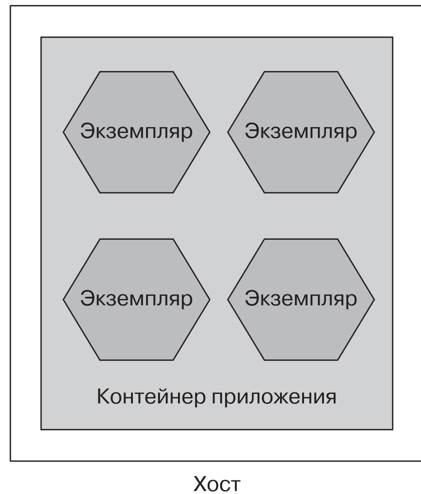


Рис. 8.17. Несколько микросервисов на контейнер приложения

Первый недостаток заключается в ограничении выбора технологии. Потребуется купить стек технологий. Это может сократить не только выбор технологий для реализации самого сервиса, но и варианты автоматизации и управления системами. Одним из способов снижения издержек по управлению несколькими хостами является автоматизация, поэтому ограничение возможностей решения этой проблемы может нанести двойной ущерб.

Я бы также поставил под сомнение ценность некоторых функций, предоставляемых контейнерами приложений. Многие из них заявляют о возможности управлять кластерами для поддержки общего состояния сессий в памяти, чего мы, безусловно, хотим избежать из-за проблем, которые это создает при масштабировании сервисов. А возможностей мониторинга, которые они предоставляют, будет недостаточно, если рассматривать виды объединенного мониторинга, которые хотелось бы выполнять в мире микросервисов, как будет описано в главе 10. Многие из них также имеют довольно длительное время запуска, что влияет на циклы обратной связи с разработчиками.

Есть и другие проблемы. Попытка надлежащего управления жизненным циклом приложений на базе таких платформ, как JVM, может быть проблема-

тичной и более сложной, чем обычный перезапуск JVM. Анализ использования ресурсов и потоков также станет намного сложнее, как только у вас появится несколько приложений, совместно использующих один и тот же процесс. И помните: даже если вы получаете пользу от контейнеров, предназначенных для конкретной технологии, они не бесплатны. Помимо того, что многие из них коммерческие и, следовательно, связаны с затратами, они сами по себе увеличивают затраты ресурсов.

В конечном счете данный подход снова сводится к попытке оптимизировать дефицит ресурсов. Независимо от того, решите ли вы использовать несколько сервисов на хост в качестве модели развертывания, я бы настоятельно рекомендовал рассматривать автономные развертываемые микросервисы как артефакты, где каждый экземпляр микросервиса выполняется как отдельный изолированный процесс.

По сути, отсутствие изоляции, обеспечиваемой этой моделью, стало одной из основных причин, по которой применение такой модели встречается все реже среди специалистов, внедряющих микросервисные архитектуры.

Платформа как услуга (PaaS)

При использовании модели PaaS вы работаете с абстракцией более высокого уровня, чем с одним хостом. Некоторые из этих платформ основаны на использовании технологического артефакта, например WAR-файл Java или Ruby gem, и его автоматической подготовке и запуске. Часть платформ будут прозрачно пытаться управлять масштабированием системы в обоих направлениях. Другие позволят контролировать количество узлов, на которых может работать ваш сервис, но они справятся с остальными задачами.

Сегодня, как и во времена первого издания этой книги, большинство лучших, наиболее отточенных решений PaaS размещены на хостинге. Heroku установил эталон для предоставления удобного интерфейса для разработчиков и, возможно, остается золотым стандартом для PaaS, несмотря на ограниченный рост функций за последние несколько лет. Платформы, подобные Heroku, не просто запускают экземпляр вашего приложения, они также предоставляют такие возможности, как запуск экземпляров базы данных, что может быть очень болезненно делать самостоятельно.

Когда решения PaaS работают — это очень хорошо. Однако когда они работают не совсем так, как вы задумывали, возможности залезть «под капот» и все исправить у вас, скорее всего, не будет. Это часть компромисса, на который вы идете. Я бы сказал, что, по моему опыту, чем умнее пытаются быть решения PaaS, тем чаще возникает проблема. Я использовал несколько моделей PaaS, пытавшихся автоматически масштабироваться в зависимости от использования приложения, но они делали это плохо. Неизменно эвристика, управляющая этими умными функциями адаптирована к среднестатистическому приложению,

а не под ваш конкретный вариант использования. Чем более нестандартно ваше приложение, тем выше вероятность, что оно будет конфликтовать с PaaS.

Поскольку хорошие решения PaaS обрабатывают множество ваших задач, они могут стать отличным способом справиться с возросшими затратами, связанными с большим количеством подвижных элементов. Тем не менее я все еще не уверен, что модели в этой области работают правильно, а ограниченные возможности самостоятельного размещения означают, что такой подход может вам не подойти. При написании первого издания книги я надеялся, что мы увидим внушительный рост в этой сфере, но увы. Мне кажется, что по факту эту потребность начал удовлетворять рост количества бессерверных продуктов, предлагаемых в основном облачными провайдерами. Вместо предложения платформ по типу «черного ящика» для размещения приложения они предоставляют готовые управляемые решения для таких вещей, как брокеры сообщений, базы данных, хранилища и т. п., позволяющие смешивать и сочетать нужные элементы систем, чтобы создать, что нам необходимо. Именно на этом фоне модель «функция как услуга», специфический тип бессерверного продукта, завоевывает все большую популярность.

Оценить пригодность предложений PaaS для микросервисов сложно, поскольку они бывают разных форм и размеров. Например, Heroku сильно отличается от Netlify, но оба они способны работать в качестве платформы развертывания микросервисов в зависимости от характера вашего приложения.

Функция как услуга (FaaS)

За последние несколько лет единственной технологией, которая хотя бы немного приблизилась к Kubernetes по уровню создаваемой шумихи (по крайней мере в контексте микросервисов), стала бессерверная модель. «Бессерверный» на самом деле представляет собой обобщающий термин для множества различных технологий, где, с точки зрения использующего их человека, базовые компьютеры не имеют значения. У вас забирают детали управления и настройки машин. По словам Кена Фромма (https://oreil.ly/hM2uq) (который, насколько я могу судить, придумал термин «бессерверный»):

> Слово «бессерверный» не означает, что серверы больше не задействуются. Идея в том, что разработчикам больше не нужно так много задумываться о них. Вычислительные ресурсы используются в качестве сервисов без необходимости управлять физическими мощностями или ограничениями. Поставщики услуг все чаще берут на себя ответственность за контроль над серверами, хранилищами данных и другими ресурсами инфраструктуры. Разработчики могли бы создавать свои собственные решения с открытым исходным кодом, но это означает, что они должны управлять серверами, очередями и нагрузками.
>
> *Кен Фромм. Почему будущее программного обеспечения и приложений — бессерверные решения*

Модель FaaS стала настолько важной частью бессерверной модели, что для многих эти два термина взаимозаменяемы. Это досадно, поскольку упускается из виду важность других бессерверных продуктов, таких как базы данных, очереди, решения для хранения данных и т. п. Тем не менее это говорит об ажиотаже, вызванном моделью FaaS, которая обсуждалась очень бурно.

Именно продукт AWS Lambda, запущенный в 2014 году, вызвал интерес к FaaS. С одной стороны, концепция восхитительно проста. Вы развертываете некоторый код (функцию), который находится в состоянии бездействия, пока не произойдет что-то, что вызовет этот код. Вы сами решаете, что будет этим триггером: файл, поступающий в определенное местоположение, элемент, появляющийся в очереди сообщений, вызов, поступающий по протоколу HTTP, — что угодно.

При появлении импульса функция запускается, а когда сигнал прекращается — она завершается. Базовая платформа обрабатывает перемещение вызовов этих функций вверх или вниз по запросу и одновременное выполнение ваших функций, чтобы у вас могло быть несколько запущенных копий одновременно, где это уместно.

Преимуществ здесь море. Код, который не выполняется, ничего не стоит в денежном выражении — вы платите только за то, что используете. Это делает FaaS отличным вариантом для ситуаций, когда у вас низкая или непредсказуемая нагрузка. Платформа, лежащая в основе FaaS, выполняет за вас функции, обеспечивая некоторую степень неявной высокой доступности и надежности без необходимости выполнения какой-либо работы. По сути, использование платформы FaaS, как и многих других бессерверных предложений, позволяет резко сократить объем операционных издержек, о которых вам нужно беспокоиться.

Ограничения

Все известные мне реализации FaaS используют некую контейнерную технологию. Это скрыто от пользователя. Обычно вам не приходится беспокоиться о создании запускаемого контейнера, вы просто предоставляете некоторую упакованную форму кода. Однако это означает, что вам не хватает определенной степени контроля над запускаемыми процессами. В результате вам требуется поставщик FaaS для поддержки выбранного вами языка программирования. Функции Azure показали себя здесь лучше всех, поддерживая широкий спектр различных сред выполнения, в то время как собственное предложение облачных функций Google Cloud, для сравнения, поддерживает очень мало языков (на момент написания книги Google поддерживал только Go, некоторые версии Node и Python). Стоит отметить, что AWS теперь разрешает определять собственную пользовательскую среду выполнения для ваших функций, теоретически позволяя реализовать поддержку языков, которые не предоставляются «из коробки». Хотя это становится еще одной частью операционных издержек, требующих вашего обслуживания.

Отсутствие контроля над базовой средой выполнения также распространяется на ресурсы, выделяемые для каждого вызова функции. В Google Cloud, Azure и AWS можно управлять только объемом памяти, выделяемой для каждой функции. Это, в свою очередь, подразумевает, что во время выполнения вашей функции выделяется определенное количество ресурсов ЦП и I/O, но вы не можете контролировать эти аспекты напрямую. В конечном счете придется выделить больше памяти для функции, даже если она в этом не нуждается, просто чтобы получить необходимый объем загрузки процессора. В итоге если вы чувствуете, что требуется провести большую тонкую настройку доступных для ваших функций ресурсов, то на данном этапе FaaS, скорее всего, не станет для вас оптимальным вариантом.

Еще одно ограничение, о котором следует знать, заключается в том, что для вызовов функций могут быть установлены лимиты по времени выполнения. Например, продолжительность выполнения облачных функций Google в настоящее время ограничено 9 минутами, в то время как функции AWS Lambda могут выполняться до 15 минут. Функции Azure при желании могут работать вечно (в зависимости от типа вашего тарифного плана). Лично я думаю, что если ваши функции работают в течение длительного времени, то для решения проблем они не подходят.

Наконец, большинство вызовов функций считаются не имеющими состояния. Концептуально это означает, что функция не может получить доступ к состоянию, оставленному предыдущим вызовом функции, если это состояние не хранится в другом месте (например, в базе данных). Это затруднило объединение нескольких функций в цепочку. Представьте, как одна функция производит оркестрацию серии вызовов других нижестоящих функций. Заметным исключением станет библиотека Azure Durable Functions (https://oreil.ly/I6ZSc), решающая эту проблему действительно интересным способом. Расширение Durable Functions (устойчивые функции) поддерживает возможность приостановить состояние указанной функции и разрешить ей возобновить работу с того места, на котором она остановилась, — все это обрабатывается прозрачно с помощью реактивных расширений. Такое решение, на мой взгляд, значительно удобнее для разработчиков, чем собственные пошаговые функции AWS, связывающие воедино несколько функций с помощью конфигурации на основе JSON.

WEBASSEMBLY (WASM)

Wasm — это официальный стандарт, предоставляющий разработчикам возможность запускать изолированные программы, написанные на различных языках программирования, в пользовательских браузерах. Цель Wasm — обеспечить безопасный и эффективный запуск произвольного кода на клиентских устройствах, определяя как формат упаковки, так и среду выполнения. Это позволяет создавать гораздо более сложные клиентские приложения при использовании обычных браузеров. В качестве конкретного примера рассмотрим eBay. Они использовали Wasm для реализации программного

обеспечения сканирования штрихкодов в Интернете, ядро которого написано на C++. Ранее это ПО было доступно только для нативных приложений eBay на Android или iOS[1].

Системный интерфейс WebAssembly (WASI) был определен как способ, позволяющий Wasm перемещаться из браузера и работать везде, где можно найти совместимую реализацию WASI. Примером может служить возможность запуска Wasm в сетях доставки контента, таких как Fastly или Cloudflare.

Благодаря своей легкой природе и мощным концепциям «песочницы», встроенным в его основную спецификацию, Wasm потенциально может бросить вызов использованию контейнеров в качестве формата развертывания для серверных приложений. В краткосрочной перспективе его сдерживают, скорее всего, серверные платформы, доступные для запуска Wasm. Хотя теоретически Wasm реально запустить, например, в Kubernetes, в конечном счете Wasm будет встроен в контейнеры, что в итоге оказывается несколько бессмысленным, поскольку выполняется более легкое развертывание внутри (сравнительно) более тяжелого контейнера.

Чтобы максимально использовать потенциал Wasm, вероятно, потребуется серверная платформа развертывания со встроенной поддержкой WASI. Гипотетически планировщик, подобный Nomad, был бы лучше приспособлен для поддержки Wasm, поскольку он обеспечивает подключаемую модель драйвера. Время покажет!

Проблемы

Помимо рассмотренных ограничений, есть некоторые иные проблемы, с которыми вы можете столкнуться при использовании FaaS.

Для начала важно рассмотреть вопрос, связанный с понятием времени раскрутки. Концептуально функции вообще не выполняются, если в них нет необходимости. Это означает, что они будут запущены для обслуживания входящего запроса. На данный момент некоторым средам выполнения требуется много времени для запуска. Это называют холодным запуском. JVM и .NET сильно страдают от этого недостатка, поэтому время холодного запуска функций, использующих эти среды выполнения, часто оказывается значительным.

Однако они редко запускаются в «холодном» режиме. По крайней мере, в AWS они поддерживаются в «теплом» состоянии, так что поступающие запросы обслуживаются уже работающими экземплярами. Это происходит до такой степени быстро, что в настоящее время может быть трудно оценить влияние холодного старта из-за проводимой оптимизации поставщиками FaaS. Тем не менее, если это вызывает беспокойство, использование языков с незначительным «временем раскрутки» (Go, Python, Node и Ruby) может эффективно решить данную проблему.

Наконец, аспект динамического масштабирования функций также может стать дилеммой. Функции запускаются при появлении триггера. У всех

[1] *Padmanabhan S., Jha P.* WebAssembly at eBay: A Real-World Use Case // eBay, 22 мая 2019 года. https://oreil.ly/SfvHT.

используемых мной платформ было жесткое ограничение на максимальное количество одновременных вызовов, на что вам, возможно, придется обратить особое внимание. Я общался с несколькими командами, которые столкнулись с проблемой масштабирования функций и перегрузки других частей своей инфраструктуры, не обладавших такими же свойствами масштабирования. Стив Фолкнер из Bustle поделился (https://oreil.ly/tFdCk) одним из таких примеров, когда функции масштабирования перегружали инфраструктуру Redis в Bustle, вызывая трудности в эксплуатации. Если одна часть вашей системы может динамически масштабироваться, а другие — нет, то вы вскоре заметите, что это несоответствие вызывает серьезные проблемы.

Сопоставление с микросервисами

До сих пор в наших обсуждениях различных вариантов развертывания сопоставление экземпляра микросервиса с механизмом развертывания было довольно простым. Один экземпляр микросервиса может быть развернут на виртуальной машине, упакован в виде единого контейнера или даже помещен в контейнер приложений, такой как Tomcat или IIS. С применением FaaS все становится немного запутаннее.

Одна функция на один микросервис. Теперь очевидно, что один экземпляр микросервиса может быть развернут как одна функция, как показано на рис. 8.18. Такой подход сохраняет концепцию экземпляра микросервиса как единицы развертывания — модели, которую до сих пор момента мы изучали наиболее внимательно.

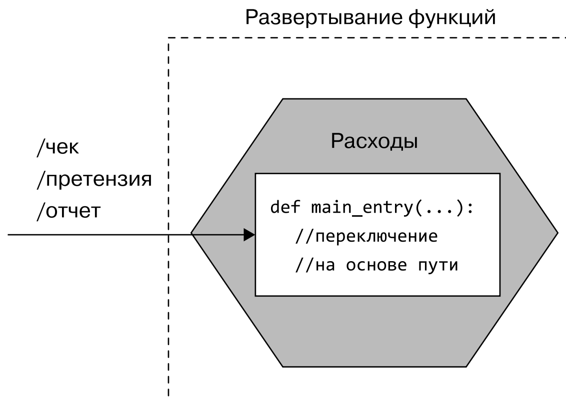


Рис. 8.18. Сервис Расходы реализован как отдельная функция

При вызове платформа FaaS запускает единственную точку входа в вашу развернутую функцию. Это означает, что, если вы собираетесь развернуть единую функцию для всего своего сервиса, вам потребуется каким-то образом диспетчеризировать эту точку входа к различным функциональным частям вашего микросервиса. Если бы вы реализовали сервис `Расходы` как микросервис

на основе REST, у вас могли бы быть открыты различные ресурсы, например `/чек`, `/претензия` или `/отчет`. В этой модели запрос любого из этих ресурсов будет поступать через эту же точку входа, поэтому вам понадобится направить входящий вызов на соответствующую часть функциональности на основе пути входящего запроса.

Одна функция на один агрегат. Итак, как можно разбить экземпляр микросервиса на более мелкие функции? При использовании предметно-ориентированного проектирования, возможно, уже были явно смоделированы агрегаты (набор объектов, управляемых как единая сущность, обычно ссылающихся на концепции реального мира). Если ваш экземпляр микросервиса обрабатывает несколько агрегатов, то одна из моделей заключается в выделении функции для каждого агрегата, как показано на рис. 8.19. Это гарантирует, что вся логика для одного агрегата будет самодостаточной внутри функции, что облегчает обеспечение последовательной реализации управления жизненным циклом агрегата.

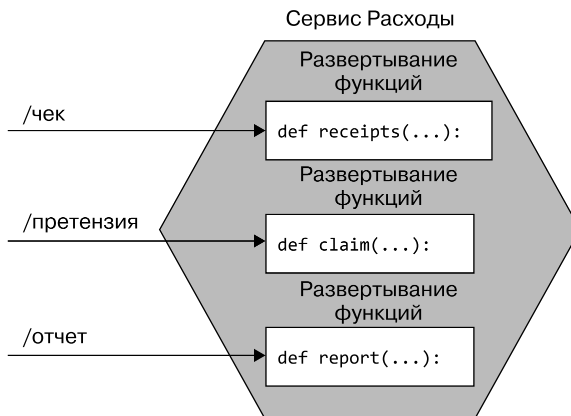


Рис. 8.19. Сервис Расходы развертывается в виде нескольких функций, каждая из которых обрабатывает отдельный агрегат

При использовании такой модели экземпляр микросервиса больше не сопоставляется с одной единицей развертывания. Вместо этого микросервис теперь представляет собой скорее логическую концепцию, состоящую из множества различных функций, которые могут быть развернуты независимо друг от друга (теоретически).

Здесь есть несколько предостережений. Во-первых, я бы настоятельно рекомендовал поддерживать более общий внешний интерфейс. Что касается вышестоящих потребителей, то они все еще общаются с сервисом `Расходы` и не знают, что запросы сопоставляются с агрегатами с меньшим охватом. Это гарантирует, что ваши действия по рекомбинации элементов или даже реструктуризации агрегатной модели не повлияют на вышестоящих потребителей.

Вторая проблема связана с данными. Должны ли эти агрегаты продолжать использовать общую БД? В этом вопросе я несколько расслаблен. Предполагая, что всеми этими функциями управляет одна и та же команда и что концептуально это остается единым «сервисом», я не против, чтобы они по-прежнему использовали одну базу данных, как показано на рис. 8.20.

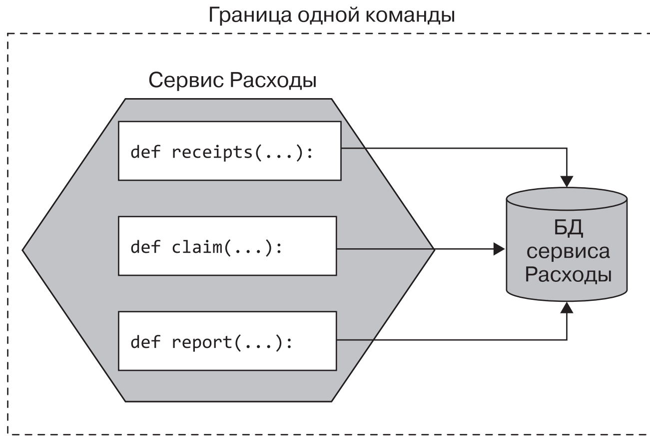


Рис. 8.20. Различные функции, использующие одну и ту же базу данных, поскольку все они логически являются частью одного и того же микросервиса и управляются одной и той же командой

Однако со временем, если потребности каждой агрегатной функции будут различаться, я бы предпочел разделить их использование данных, как показано на рис. 8.21, особенно если вы замечаете, что связанность на уровне данных ухудшает возможность легко изменять их. На этом этапе можно утверждать, что эти функции теперь являются самостоятельными микросервисами, хотя, как я только что объяснил, имеет смысл по-прежнему представлять их как единый микросервис для вышестоящих потребителей.

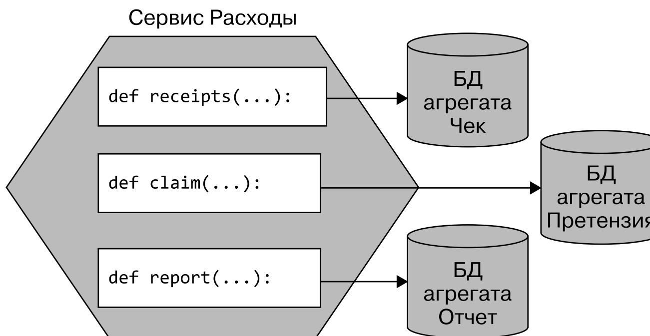


Рис. 8.21. Каждая функция использует свою собственную базу данных

Это сопоставление одного микросервиса с несколькими более детализированными единицами развертывания несколько искажает наше предыдущее определение микросервиса. Обычно мы рассматриваем микросервис как независимо развертываемый модуль, а теперь один микросервис состоит из нескольких различных независимо развертываемых модулей. Концептуально в этом примере микросервис становится скорее логической, чем физической идеей.

Сделайте все еще более детализированным. Если вы хотите перейти на еще меньший масштаб, есть соблазн разбить ваши функции для каждого агрегата на более мелкие части. Здесь стоит быть гораздо осторожнее. Помимо того что это может привести к появлению большого количества функций, это также нарушает один из основных принципов агрегатов: мы хотим рассматривать его как единое целое, чтобы лучше управлять целостностью самого агрегата.

Ранее у меня была идея сделать каждый переход состояния агрегатов отдельной функцией, но я отказался от такого подхода из-за проблем, связанных с несогласованностью. Когда у вас есть разные независимо развертываемые элементы, каждый из которых представляет собой отдельную часть общего перехода состояния, обеспечить правильное выполнение задач становится действительно трудно. Это переносит нас в пространство шаблонов сага, которые мы обсуждали в главе 6. При реализации сложных бизнес-процессов такие концепции, как саги, важны и работа оправданна. Однако я не вижу смысла в добавлении этой сложности на уровне управления одним агрегатом, который может быть легко обработан с помощью одной функции.

Дальнейшие шаги

Я по-прежнему убежден, что будущее для большинства разработчиков — это использование платформы, скрывающей от них большую часть базовых деталей. В течение многих лет Heroku был лучшим примером, на который я мог бы указать в плане того, как найти правильный баланс, но теперь у нас есть FaaS и более широкая экосистема готовых бессерверных предложений, прокладывающих иной путь.

Несмотря на то что FaaS еще предстоит решить ряд проблем, я считаю, что это та платформа, которую в конечном счете будет использовать большинство разработчиков. Не все приложения впишутся в экосистему FaaS, учитывая существующие ограничения, но те, у кого получается, уже дают значительные преимущества. По мере того как все больше и больше ресурсов направляется на создание FaaS-предложений на базе Kubernetes, люди, которые не могут напрямую использовать решения FaaS, предоставляемые основными облачными провайдерами, все чаще получают возможность применить этот новый способ работы.

Таким образом, хотя FaS может подойти не всем, я определенно призываю людей изучить этот подход. А моим клиентам, рассматривающим возможность перехода на облачные решения Kubernetes, я настоятельно рекомендую сначала

изучить FaaS, поскольку он может дать им все необходимое, скрывая при этом значительную сложность и разгружая большой объем работы.

Я вижу, что все больше организаций используют FaaS как часть более широкого решения, выбирая FaaS для конкретных случаев использования, где он хорошо подходит. Удачным примером может служить компания BBC, которая прибегает к лямбда-функциям как части своего основного технологического стека, обеспечивающего работу сайта BBC News. В целом система использует сочетание экземпляров Lambda и EC2, причем экземпляры EC2 часто используются в ситуациях, когда вызовы лямбда-функций оказались бы слишком дорогими[1].

Какой вариант развертывания подходит именно вам

Итак, у нас много вариантов, верно? И я, вероятно, не слишком помог, стараясь изо всех сил делиться множеством плюсов и минусов для каждого подхода. Если вы дочитали до этого места, то можете быть немного сбиты с толку, пытаясь понять, что вам следует делать.

Но если ваши текущие решения работают в вашей ситуации, то *продолжайте* их развивать! Не позволяйте моде диктовать выбор технических решений.

Если считаете, что вам действительно нужно изменить способ развертывания микросервисов, тогда позвольте мне попытаться обобщить многое из того, что мы уже обсудили, и предложить несколько рекомендаций.

Возвращаясь к нашим принципам развертывания микросервисов, одним из наиболее важных аспектов было обеспечение изоляции микросервисов. Но использование только его в качестве руководящего принципа может привести к применению выделенных физических машин для каждого экземпляра микросервиса! Это бы влетело нам в копеечку, и, как мы уже обсуждали, есть несколько очень мощных инструментов, которые бы стали недоступны, если бы мы пошли по такому пути.

Компромиссов здесь предостаточно. Создание баланса между стоимостью и простотой использования, изоляцией, привычностью в работе... может стать невыполнимой задачей. Итак, рассмотрим набор правил, которые я люблю называть «базовые правила Сэма для определения того, где развертывать материал».

---

[1] *Ishmael J.* Optimising Serverless for BBC Online // Technology and Creativity at the BBC (блог), BBC. 26 января 2021 года. https://oreil.ly/gkSdp.

1. Не пытайтесь починить то, что не сломано[1].
2. Отдайте автоматике столько контроля, сколько хочется, а затем отдайте еще немного. Если вы можете переложить всю свою работу на хороший PaaS, такой как Heroku (или платформа FaaS), тогда сделайте это и будьте счастливы. Вам действительно нужно возиться с настройкой до последнего?
3. Контейнеризация микросервисов не безболезненная, но станет действительно хорошим компромиссом между стоимостью изоляции и некоторыми фантастическими преимуществами для локальной разработки, в то же время предоставляя вам определенную степень контроля над происходящим. Ожидайте появления Kubernetes в своем будущем.

Многие люди скандируют: «Kubernetes или провал!» — что, на мой взгляд, бесполезно. Если вы работаете в публичном облаке и ваша проблема соответствует FaaS как модели развертывания, примените ее и пропустите Kubernetes. Ваши разработчики, скорее всего, скажут вам спасибо. Как описывается в главе 16, не позволяйте страху держать вас в ловушке собственного беспорядка.

Нашли потрясающий PaaS, такой как Heroku или Zeit, и у вас есть приложение, подходящее под ограничения платформы? Перенесите всю работу на платформу и уделите больше времени работе над своим продуктом. И Heroku, и Zeit — довольно крутые платформы с шикарным удобством использования для разработчика. Разве ваши разработчики, в конце концов, не заслуживают того, чтобы быть счастливыми?

Для остальных же контейнеризация — это правильный путь. Пришло время поговорить о Kubernetes.

КАКАЯ РОЛЬ ОТВЕДЕНА PUPPET, CHEF И ДРУГИМ ИНСТРУМЕНТАМ

Данная глава значительно изменилась со времен первого издания. Отчасти это связано с развитием отрасли в целом, но также и с новыми, становящимися все более полезными технологиями. Их появление также привело к уменьшению роли других инструментов (например, Puppet, Chef, Ansible и Salt) в развертывании микросервисных архитектур по сравнению с 2014 годом.

Основной причиной этого, по сути, стала популярность контейнера. Сила таких инструментов, как Puppet и Chef, состояла в том, что они дают вам способ привести машину в желаемое состояние, которое определено в некоторой кодируемой форме. Вы можете определить, какое время выполнения вам нужно, где должны находиться файлы конфигурации и т. д., таким образом, чтобы их можно было детерминированно запускать снова и снова на той же самой машине, гарантируя, что она всегда будет приведена в одно и то же состояние.

Большинство людей создают контейнер путем определения файла Dockerfile. Это позволяет определить те же требования, что и в случае с Puppet или Chef, но с некоторыми отличиями. Контейнер разрушается при

[1] Я мог бы и не придумать это правило.

повторном развертывании, поэтому каждое создание контейнера выполняется с нуля (здесь я несколько упрощаю). Большая сложность, присущая Puppet и Chef для работы с этими инструментами, запускаемыми снова и снова на одних и тех же машинах, не требуется.

Puppet, Chef и подобные им по-прежнему невероятно полезны, однако теперь их вытеснили из контейнера и переместили дальше по стеку. Люди используют аналогичные инструменты для управления устаревшими приложениями и инфраструктурой или для создания кластеров, на которых теперь выполняются контейнерные рабочие нагрузки. Но разработчики еще реже, чем в прошлом, вступают в контакт с этими инструментами.

Концепция IAC по-прежнему жизненно важна. Просто изменился тип инструментов, которые разработчики, скорее всего, будут использовать. Для тех, кто работает с облаком, например, такие вещи, как Terraform (http://terraform.io), могут оказаться очень полезным для предоставления облачной инфраструктуры. Недавно я стал большим поклонником Pulumi (http://pulumi.com), который отказывается от использования предметно-ориентированных языков (domain-specific languages, DSL) в пользу обычных языков программирования, чтобы помочь разработчикам управлять своей облачной инфраструктурой. Я вижу большие перспективы для внедрения Pulumi, поскольку команды доставки все больше и больше берут на себя ответственность за операционный мир. И я подозреваю, что Puppet, Chef и т. д., хотя и продолжат играть полезную роль в операциях, скорее всего, постепенно начнут отходить от повседневной деятельности по разработке.

Kubernetes и оркестрация контейнеров

По мере того как контейнеры начали набирать популярность, многие люди стали искать решения для управления контейнерами на нескольких машинах. У Docker было две попытки сделать это (Docker Swarm и Docker Swarm Mode). Такие компании, как Rancher и CoreOS, придумали свои решения, а платформы более общего назначения, например Mesos, использовались для запуска контейнеров наряду с другими видами рабочих нагрузок. Однако, несмотря на большие усилия по созданию этих продуктов, Kubernetes за последние пару лет стал доминировать.

Прежде чем говорить о самом Kubernetes, в первую очередь стоит обсудить, почему существует необходимость в подобном инструменте.

Пример для контейнерной оркестровки

В целом Kubernetes можно описать как платформу оркестрации контейнеров или, если использовать вышедший из моды термин, планировщик контейнеров. Итак, что же это за платформы и зачем они могут понадобиться?

Контейнеры создаются путем изоляции набора ресурсов на базовой машине. Такие инструменты, как Docker, позволяют определить вид контейнера и создать

его экземпляр на компьютере. Но большинство решений требуют, чтобы наше ПО было установлено на нескольких машинах, возможно, для обработки достаточной нагрузки или для обеспечения избыточности системы, позволяющей выдержать отказ одного узла. Платформы оркестрации контейнеров определяют, как и где выполняются рабочие нагрузки контейнеров. Термин «планирование» начинает приобретать больше смысла в этом контексте. Оператор говорит: «Я хочу, чтобы этот блок запустился», и оркестратор планирует эту работу — находит доступные ресурсы, перераспределяет их при необходимости и обрабатывает детали для оператора.

Различные платформы оркестрации контейнеров также обрабатывают управление желаемым состоянием, гарантируя поддержку ожидаемого состояния набора контейнеров (в нашем случае экземпляров микросервиса). Они также позволяют указать, как мы хотим распределять рабочие нагрузки, что помогает оптимизировать ресурсы, задержки между процессами или по соображениям надежности.

Без такого инструмента вам придется управлять распределением контейнеров, что, как я могу вам сказать по собственному опыту, очень быстро надоедает. Написание скриптов для управления запуском и подключением к сети экземпляров контейнеров — занятие не из приятных.

В целом все платформы оркестрации контейнеров, включая Kubernetes, предоставляют подобные возможности в той или иной форме. Если посмотреть на планировщики общего назначения, например Mesos или Nomad, на управляемые решения, такие как ECS AWS, Docker Swarm Mode и т. д., вы увидите аналогичный набор функций. Но по причинам, которые мы вскоре рассмотрим, Kubernetes завоевал эту нишу. В нем также есть одна или две интересные концепции, которые стоит вкратце изучить.

Упрощенный взгляд на концепции Kubernetes

В Kubernetes существует множество других концепций, поэтому простите меня, что я не вдаюсь в описание их всех (они определенно заслуживают отдельной книги). Здесь я постараюсь изложить ключевые идеи, с которыми вам придется столкнуться, когда вы только начнете работать с этим инструментом. Сначала рассмотрим концепцию кластера, она показана на рис. 8.22.

По сути, кластер Kubernetes состоит из двух вещей: набора машин, на которых будут выполняться рабочие нагрузки, называемые узлами, и набора управляющего ПО, контролирующего эти узлы и называющегося плоскостью управления. На узлах можно запустить физические или виртуальные машины. Вместо планирования контейнера Kubernetes планирует то, что он называет *подом* (набором контейнеров). Под состоит из одного или нескольких контейнеров, развертывающихся совместно.

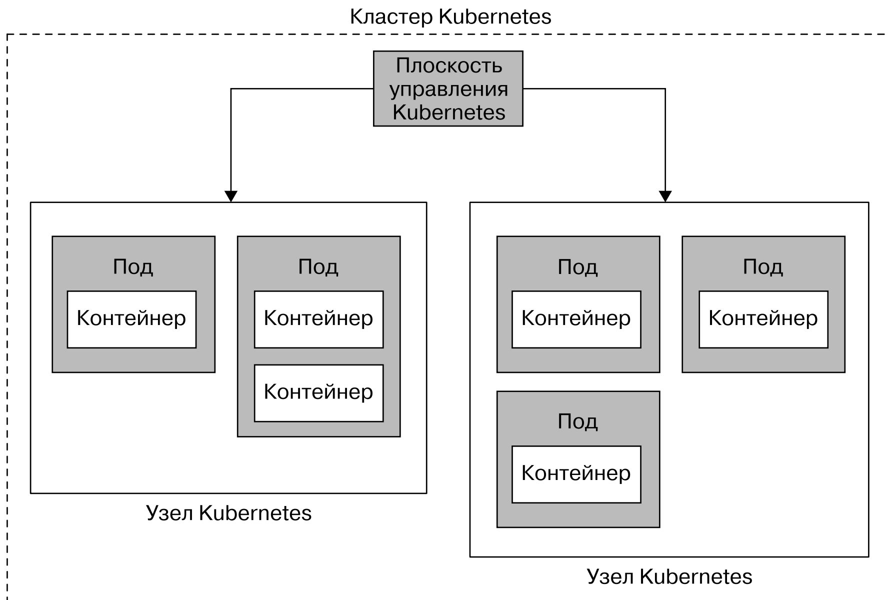


Рис. 8.22. Краткий обзор топологии Kubernetes

Обычно в поде находится всего один контейнер, например экземпляр микросервиса. Однако в некоторых случаях (довольно редких) развертывание нескольких контейнеров вместе может иметь смысл. Например, использование sidecar-прокси, таких как Envoy, часто как часть сервисной сети — тема, которую мы обсуждали в разделе «Сервисные сети и API-шлюзы» в главе 5.

Следующее понятие, о котором полезно знать, называется *сервисом*. В контексте Kubernetes сервис рассматривается как стабильная конечная точка маршрутизации — по сути, способ сопоставления запущенных модулей со стабильным сетевым интерфейсом, доступным в кластере. Kubernetes обрабатывает маршрутизацию внутри кластера, как показано на рис. 8.23.

Идея заключается в том, что определенный под считается эфемерным — он может быть закрыт по целому ряду причин, в то время как сервис в целом продолжает жить. Сервис существует для маршрутизации вызовов в поды и из них может обрабатывать завершение работы подов или запуск новых. Чисто с терминологической точки зрения это может сбить с толку. Мы говорим в более общем плане о развертывании сервиса, но в Kubernetes вы не развертываете сервис — вы развертываете поды, сопоставляемые с сервисом. Вам может потребоваться некоторое время, чтобы разобраться.

Далее у нас есть *набор реплик*. С помощью набора реплик вы определяете желаемое состояние набора подов. Здесь вы говорите: «Я хочу четыре таких пода», а Kubernetes справляется с остальным. На практике от вас больше не требуется

работать с наборами реплик напрямую. Вместо этого они обрабатываются за вас с помощью *развертывания* — последней концепции, которую мы рассмотрим. Развертывание — это способ применения изменений к своим модулям и наборам реплик. При развертывании допускается выполнять такие действия, как выпуск непрерывных обновлений (таким образом, вы постепенно заменяете модули актуальной версией, чтобы избежать простоев), откаты, увеличение числа узлов и многое другое.

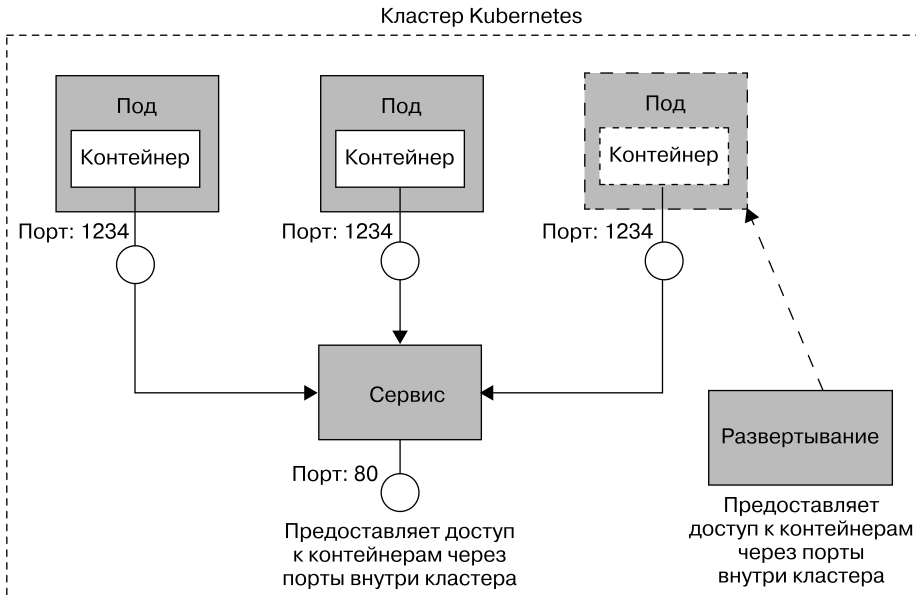


Рис. 8.23. Как под, сервис и развертывание работают вместе

Итак, чтобы развернуть свой микросервис, вы определяете *под*, внутри которого находится ваш экземпляр микросервиса, и *сервис*, который позволит Kubernetes узнать, как будет осуществляться доступ к вашему микросервису. Затем применяете изменения к запущенным подам с помощью *развертывания*. На словах кажется все легко, правда? Скажем так, я опустил здесь довольно много материала для краткости.

Мультиарендность и федерация

С точки зрения эффективности хотелось бы объединить все доступные вычислительные ресурсы в одном кластере Kubernetes и запускать там рабочие нагрузки со всей организации. Это, вероятно, позволит более эффективно использовать базовые ресурсы, поскольку те, что не задействованы, могут быть

свободно перераспределены. Это, в свою очередь, должно соответствующим образом снизить затраты.

Хотя Kubernetes хорошо управляет различными микросервисами для разных целей, у него есть ограничения в отношении того, насколько мультиарендной является платформа. Всевозможным подразделениям вашей организации может потребоваться разная степень контроля над различными ресурсами. Такого рода элементы управления не были встроены в Kubernetes, и это решение кажется разумным с точки зрения попытки несколько ограничить область применения Kubernetes. Чтобы обойти эту проблему, организации рассматривают несколько разных путей.

Первый вариант — использовать построенную поверх Kubernetes платформу. OpenShift от Red Hat, например, обладает богатым набором средств контроля доступа и других возможностей, созданных с учетом потребностей более крупных организаций и которые могут несколько упростить концепцию мультиарендности. Помимо любых финансовых последствий использования подобных платформ, для их работы иногда придется прибегать к абстракциям, предоставленным выбранным вами поставщиком. Это означает, что ваши разработчики должны знать, как использовать не только Kubernetes, но и платформу конкретного поставщика.

Другой подход заключается в рассмотрении федеративной модели, описанной на рис. 8.24. С такой моделью вы получаете несколько отдельных кластеров, поверх которых располагается некоторый слой ПО, что позволяет вносить изменения во все кластеры при необходимости. Во многих случаях пользователи будут работать непосредственно с одним кластером, что предполагает довольно знакомый опыт работы с Kubernetes. Но в отдельных ситуациях может появиться потребность распределить приложение по нескольким кластерам, например, если эти кластеры находятся в разных географических регионах и вы хотите, чтобы приложение было развернуто с возможностью справиться с потерей целого кластера.

Федеративный характер делает объединение ресурсов более сложным. Как показано на рис. 8.24, кластер А полностью задействован, в то время как у кластера Б достаточно неиспользуемых мощностей. Если потребуется выполнить больше рабочих нагрузок в кластере А, это станет возможным только в случае предоставления ему дополнительных ресурсов. Например, переместить пустой узел кластера Б в А. Насколько легко это сделать, будет зависеть от характера используемого ПО федерации, но я вполне могу представить, что это нетривиальное изменение. Имейте в виду, что один узел может быть частью как одного, так и другого кластера и поэтому не способен запускать поды кластеров А и Б.

Стоит отметить, что наличие нескольких кластеров может быть полезным при рассмотрении проблемы модернизации самого кластера. Возможно, будет проще и безопаснее перенести микросервис в недавно обновленный кластер, чем обновлять кластер на месте.

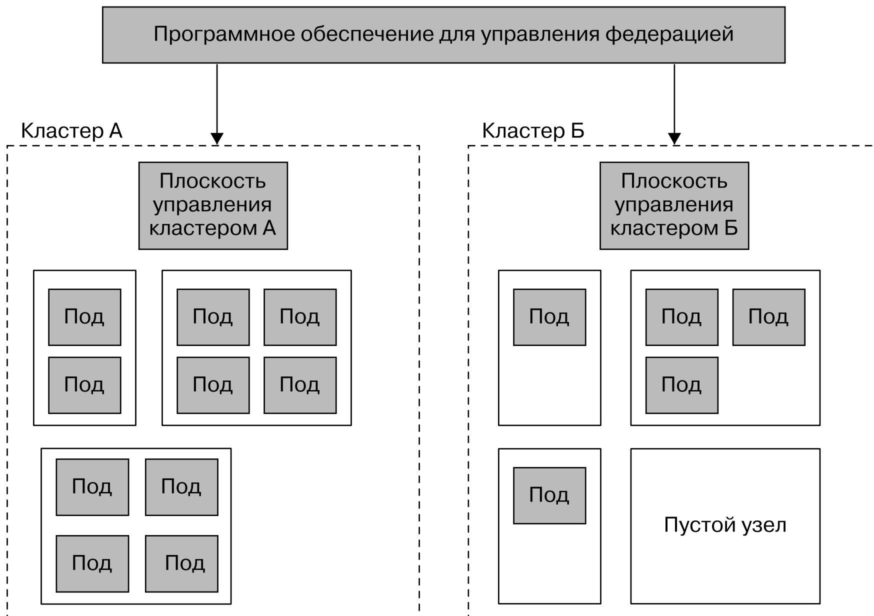


Рис. 8.24. Пример федерации в Kubernetes

По сути, это проблемы масштабирования. В некоторых организациях никогда не возникает таких дилемм, поскольку в них совместно используется один кластер. Для других организаций, стремящихся повысить эффективность в большем масштабе, это, безусловно, требующая более подробного изучения область. Следует отметить, что существует ряд различных представлений о том, как должна выглядеть федерация Kubernetes, и цепочек инструментов для управления кластерами.

ПРЕДЫСТОРИЯ KUBERNETES

Kubernetes начинался как проект с открытым исходным кодом в Google, который черпал вдохновение из более ранних систем управления контейнерами Omega и Borg. Многие из основных концепций Kubernetes основаны на идеях управления рабочими нагрузками контейнеров внутри Google, хотя и преследуют несколько иную цель. Borg управляет системами огромных глобальных масштабов, обрабатывая десятки, если не сотни тысяч контейнеров в дата-центрах по всему миру. Если вам нужна более подробная информация о том, как сравниваются различные взгляды, лежащие в основе этих трех платформ Google (хотя и в контексте Google), я рекомендую статью (https://oreil.ly/fVCSS) Брендана Бернса и др.

Хотя у Kubernetes есть некоторые общие черты с Borg и Omega, работа в подобном масштабе не была главной движущей силой проекта. Как Nomad, так и Mesos (каждый из которых взял пример с Borg) нашли

свою нишу в ситуациях, когда требуются кластеры из тысяч машин, как показано в использовании Mesos компанией Apple для Siri[1] или Nomad в Roblox (https://oreil.ly/tyWof).

В Kubernetes хотели позаимствовать идеи у Google, но обеспечить более удобный для разработчиков интерфейс, чем у Borg или Omega. Можно взглянуть на решение Google вложить много усилий в создание инструмента с открытым исходным кодом с чисто альтруистической точки зрения. И хотя я уверен, что кто-то так и хотел, но реальность такова, что это в значительной степени связано с риском, который Google видела в конкуренции со стороны AWS и т. п.

На рынке облачных сервисов Google Cloud набрал обороты, но по-прежнему отстает на треть от Azure и AWS (которые лидируют), а по некоторым данным, Alibaba Cloud оттесняет его на четвертое место. Несмотря на увеличение доли рынка, это по-прежнему далеко от желаний Google.

Вполне вероятно, что главная проблема заключалась в том, что явный лидер рынка — AWS — в итоге может стать практически монополистом в области облачных вычислений. Более того, опасения по поводу стоимости перехода от одного провайдера к другому означали, что такое доминирующее положение на рынке трудно изменить. А затем появляется Kubernetes с его обещанием предоставить стандартную платформу для запуска контейнерных рабочих нагрузок, которые поддерживаются несколькими поставщиками. Была надежда, что это позволит перейти от одного поставщика к другому и избежать тотального доминирования AWS.

Таким образом, стоит рассматривать Kubernetes как щедрый вклад Google в развитие ИТ-индустрии в целом или как попытку Google сохранить актуальность на быстро развивающемся рынке. Не вижу проблем с тем, чтобы считать обе причины одинаково верными.

Федерация нативных облачных вычислений

Фонд нативных облачных вычислений (Cloud Native Computing Foundation, CNCF) — это ответвление некоммерческой организации Linux Foundation. CNCF сосредоточен на курировании экосистемы проектов, чтобы помочь продвигать облачную разработку. Хотя на практике это означает поддержку Kubernetes и проектов, которые работают с самим Kubernetes или строятся на нем. Сами проекты не создаются и не разрабатываются непосредственно CNCF. Это скорее место, где они размещаются и где могут быть разработаны общие стандарты и обеспечена обратная совместимость.

Таким образом, CNCF напоминает роль проекта Apache Software Foundation, который подразумевает определенный уровень качества и более широкую поддержку сообщества. Все программы, размещенные CNCF, имеют открытый исходный код, хотя их разработка вполне может осуществляться коммерческими организациями.

[1] *Bryant D.* Apple Rebuilds Siri Backend Services Using Apache Mesos // InfoQ. 3 мая 2015 года. https://oreil.ly/NLUMX.

Помимо помощи в разработке этих связанных проектов, фонд также проводит различные мероприятия, предоставляет документацию и учебные материалы, а также определяет программы сертификации для Kubernetes. В фонде есть участники из всех областей ИТ-индустрии, и хотя маленьким группам или независимым лицам трудно играть большую роль в самой организации, степень межотраслевой поддержки (включая многие конкурирующие друг с другом компании) впечатляет.

Как сторонний наблюдатель, CNCF, похоже, добился больших успехов в распространении информации о полезности курируемых им проектов. Он также выступает в качестве места, где эволюция крупных проектов обсуждается открыто, обеспечивая широкий вклад. Фонд сыграл огромную роль в успехе Kubernetes. Скорее всего, без него все еще был бы фрагментированный ландшафт в этой области.

Платформы и мобильность

Часто можно услышать, что Kubernetes описывается как «платформа». Однако на самом деле это не платформа в том смысле, в каком этот термин понимают разработчики. Все, что действительно дает продукт из коробки, — это возможность запускать рабочие нагрузки контейнеров. Большинство пользователей Kubernetes в конечном счете собирают свою собственную платформу, устанавливая вспомогательное ПО, такое как сервисные сети, брокеры сообщений, инструменты агрегации логов и многое другое. В более крупных организациях за это отвечает команда разработчиков платформы, собирающая платформу воедино и управляющая ею, а также помогающая остальным разработчикам эффективно ее использовать.

Это может быть как благом, так и проклятием. Подобный комплексный подход стал возможным благодаря достаточно совместимой экосистеме инструментов (во многом благодаря CNCF). Это означает, что вам разрешается выбрать свои любимые инструменты для конкретных задач. Однако существует риск тирании выбора — можно легко оказаться перегруженными таким количеством доступных вариантов. Такие продукты, как OpenShift от Red Hat, частично лишают нас этого выбора, поскольку дают готовую платформу с некоторыми решениями, уже принятыми за нас.

Хотя на базовом уровне Kubernetes предлагает переносимую абстракцию для выполнения контейнеров, на практике это не так просто, как взять работающее в одном кластере приложение и ожидать, что оно будет работать в другом месте. Ваше приложение, операции и рабочий процесс разработчика вполне могут зависеть от вашей собственной пользовательской платформы. Переход из одного кластера Kubernetes в другой также потребует повторной сборки платформы в новом месте назначения. Я общался с представителями многих организаций,

внедривших Kubernetes в первую очередь по причине беспокойства о привязке к одному поставщику, но эти организации не поняли одного нюанса: построенные на Kubernetes приложения переносятся между кластерами Kubernetes в теории, но не всегда на практике.

Helm, Operator и CRD

Одной из областей, вызывающих постоянную путаницу в пространстве Kubernetes, стало управление развертыванием и жизненным циклом сторонних приложений и подсистем. Подумайте о необходимости запуска Kafka в своем кластере Kubernetes. Вы можете создать свой собственный под, сервис и спецификации развертывания и запустить их самостоятельно. Но как насчет управления обновлением вашей установки Kafka? Или других распространенных задач обслуживания, с которыми вам, возможно, потребуется справиться, например обновления запущенного ПО с отслеживанием состояния?

Появился ряд инструментов, цель которых — дать вам возможность управлять этими типами приложений на более разумном уровне абстракции. Идея в том, что кто-то создает нечто похожее на пакет для Kafka и вы запускаете его в своем кластере Kubernetes в стиле, напоминающем «черный ящик». Два наиболее известных решения — Operator и Helm. Helm позиционирует себя как «отсутствующий менеджер пакетов» для Kubernetes, и, хотя Operator может управлять начальной установкой, он, похоже, больше сосредоточен на текущем регулировании приложения. Как ни странно, хотя Operator и Helm рассматриваются как альтернативы друг другу, их можно применять совместно в некоторых ситуациях (Helm для начальной установки, Operator для операций жизненного цикла).

Более поздняя эволюция в этой области — это нечто называемое пользовательскими определениями ресурсов (custom resource definitions, CRD). С помощью CRD возможно расширить основные API Kubernetes, что позволит подключать новое поведение к вашему кластеру. Самое приятное в CRD — они довольно легко интегрируются в существующий интерфейс командной строки, элементы управления доступом и многое другое, так что ваше пользовательское расширение не кажется чужеродным дополнением. CRD в основном позволяют реализовывать свои собственные абстракции Kubernetes. Подумайте о подах, наборах реплик, сервисах и абстракциях развертывания, которые мы обсуждали ранее, — с CRD вы способны добавить свои собственные элементы в этот набор.

CRD подходят для всего: от организации небольших фрагментов конфигурации до управления сервисными сетями, например Istio, или кластерным программным обеспечением, таким как Kafka. С такой гибкой и мощной концепцией трудно понять, где лучше всего использовать CRD. Даже среди экс-

пертов, с которыми я беседовал, нет общего мнения. Все в этой сфере, похоже, не успокаивается так быстро, как я надеялся, да и единодушия не так много, как хотелось бы, — тенденция в экосистеме Kubernetes.

Knative

Knative (https://knative.dev) — это проект с открытым исходным кодом, цель которого — предоставить разработчикам методы работы в стиле FaaS с использованием Kubernetes под капотом. По сути, Kubernetes не очень удобен, особенно если сравнить его с Heroku, например, или аналогичными платформами. Цель Knative — привнести опыт разработки FaaS в Kubernetes, скрыв сложность Kubernetes от разработчиков. В свою очередь, это должно означать, что командам разработчиков станет легче управлять полным жизненным циклом своего ПО.

Мы уже обсуждали сервисные сети, в частности, я упоминал Istio в главе 5. Они необходимы для запуска Knative. В то время как Knative теоретически позволяет подключать различные сервисные сети, только Istio считается стабильным в настоящее время (поддержка Ambassador и Gloo все еще на стадии альфа-версии). На практике получается, что, если вы хотите применить Knative, необходимо приобрести Istio.

Проектам Kubernetes и Istio, в значительной степени поддерживаемым компанией Google, потребовалось очень много времени, чтобы достичь стадии, когда их можно считать стабильными. Kubernetes все еще претерпевал серьезные изменения после выпуска версии 1.0, и только совсем недавно Istio, лежащий в основе Knative, был полностью перепроектирован. Такой послужной список стабильных, готовых к эксплуатации проектов заставляет задуматься, что для Knative потребуется гораздо больше времени, чтобы быть выйти на большую сцену. Хотя некоторые организации используют его, и вы, вероятно, тоже могли бы, но опыт подсказывает, что пройдет достаточно времени, прежде чем произойдут какие-то серьезные изменения, которые потребуют болезненной миграции. Отчасти по этой причине я предложил более консервативным организациям, рассматривающим FaaS для своего кластера Kubernetes, поискать иные проекты, подобные Open-FaaS. Такие решения уже используются организациями по всему миру и не требуют базовой сервисной сети. Но если вы прямо сейчас запрыгнете в поезд Knative, не удивляйтесь, если в будущем вас ждет резкий сход с рельсов.

Еще одно замечание: было обидно видеть, что Google решила не делать Knative частью CNCF. Можно только предположить, что причиной стало желание Google управлять направлением развития самого инструмента. Kubernetes при запуске приводил в замешательство многих отчасти потому, что отражал взгляды Google, как следует управлять контейнерами. Компания получила

огромную выгоду от участия в проектировании более широкого круга представителей ИТ-индустрии, и очень жаль, что на данном этапе в Google решили, что не заинтересованы в таком же подходе для Knative.

Будущее

Забегая вперед, я не вижу никаких признаков того, что неудержимый Kubernetes остановится в ближайшее время, и я точно ожидаю увидеть больше организаций, внедряющих свои собственные кластеры Kubernetes для частных облачных сервисов или использующих управляемые кластеры в настройках публичного облака. Однако я думаю, что наблюдаемое нами сейчас, когда разработчикам приходится учиться напрямую использовать Kubernetes, будет относительно недолгим проявлением. Kubernetes отлично справляется с управлением рабочими нагрузками контейнеров и предоставляет платформу для создания других приложений. Однако он не представляет собой то, что принято считать удобным для разработчиков. Сам Google показал нам это, продвигая Knative, и я думаю, что мы продолжим наблюдать скрытое под слоями абстракции более высокого уровня применение Kubernetes. Поэтому в будущем стоит ожидать появления Kubernetes повсюду. Вы просто не будете этого знать.

Это не значит, что разработчики забудут, что они создают распределенную систему. Они все равно столкнутся с множеством проблем, которые несет с собой этот тип архитектуры. Просто им не придется так сильно беспокоиться о деталях того, как их ПО сопоставляется с базовыми вычислительными ресурсами.

Стоит ли вам его использовать

Итак, для тех из вас, кто еще не является полноправным членом клуба Kubernetes, стоит ли вам присоединиться? Начнем с того, что внедрение и управление собственным кластером Kubernetes не для слабонервных — это серьезное мероприятие. Качество опыта, который получат ваши программисты, работая с установкой Kubernetes, во многом будет зависеть от эффективности команды, управляющей кластером. По моим сведениям, по этой причине ряд крупных организаций, пойдя по пути внедрения Kubernetes on-prem, передали эту работу на аутсорсинг специализированным компаниям.

Еще лучше — прибегнуть к полностью управляемому кластеру. Если есть возможность использовать публичное облако, то применяйте полностью управляемые решения, подобные тем, которые предоставляют Google, Azure и AWS. Однако я бы сказал, что если в вашем распоряжении есть публичное облако, то подумайте, действительно ли Kubernetes — то, что вам нужно. Если требуется удобная для разработчиков платформа управления развертыванием и жизненным циклом ваших микросервисов, то FaaS — отличное решение. Также стоит

рассмотреть и подобные PaaS предложения, такие как Azure Web Apps, Google App Engine, или некоторых из более мелких поставщиков, например Zeit или Heroku.

Прежде чем принять решение об эксплуатации Kubernetes, попросите некоторых ваших администраторов и разработчиков попользоваться им. Программисты могут на своих ноутбуках запустить что-то легковесное, например minikube или MicroK8s. Это даст им представление о работе Kubernetes. А вот людям, которые должны управлять платформой, возможно, потребуется более глубокое погружение. На Katacoda (https://www.katacoda.com) есть несколько отличных онлайн-руководств для ознакомления с основными концепциями, а CNCF публикует множество учебных материалов в этой области. Убедитесь, что люди, которые действительно будут использовать этот материал, поэкспериментируют с ним, прежде чем вы примете решение.

Не попадайтесь в ловушку, думая, что у вас должен быть Kubernetes, «потому что все остальные так делают». Это такое же опасное оправдание выбора Kubernetes, как и микросервисов. Каким бы хорошим ни был Kubernetes, он не для всех — проведите свою личную оценку. И давайте будем откровенны: если у вас есть горстка разработчиков и всего несколько микросервисов, Kubernetes, скорее всего, станет обузой, даже если вы используете полностью управляемую платформу.

Поэтапная доставка

За последнее десятилетие или около того мы стали грамотнее внедрять ПО для наших пользователей. Появились новые методы, обусловленные рядом различных вариантов использования и пришедшие из самых разных частей ИТ-индустрии. Но в первую очередь все они были сосредоточены на том, чтобы сделать процесс внедрения нового программного обеспечения гораздо менее рискованным. И если релиз ПО становится менее рискованным, значит, можно выполнять его чаще.

Перед вводом программного обеспечения в эксплуатацию мы проводим множество мероприятий, помогающих выявить проблемы до того, как они повлияют на реальных пользователей. Предварительное тестирование — важнейшая часть данного процесса. Однако в главе 9 мы обсудим, как далеко это нас может завести.

В своей книге «Ускоряйся!»[1] Николь Форсгрен, Джез Хамбл и Джин Ким приводят четкие доказательства, полученные в результате обширных исследований, что компании с высокой производительностью внедряют новые решения

[1] *Форсгрен Н., Хамбл Дж., Ким Дж.* Ускоряйся! Наука DevOps: Как создавать и масштабировать высокопроизводительные цифровые организации.

чаще, чем их коллеги из компаний с низкой производительностью, и в то же время получают *гораздо более низкую частоту отказов изменений*.

Фраза «Поспешишь — людей насмешишь», похоже, неприменима, когда дело доходит до доставки ПО: частые поставки и более низкие показатели отказов идут рука об руку, и осознавшие это организации изменили свое отношение к выпуску программного обеспечения.

Эти организации используют такие методы, как переключение функций, канареечные релизы, параллельные прогоны и многое другое, о чем мы подробно расскажем в текущем разделе. Это изменение в подходе к выпуску функциональности относится к так называемой *поэтапной доставке*. Функциональность предоставляется пользователям контролируемым образом. Вместо масштабного развертывания можно разумно подойти к тому, кто и какую функциональность видит, например, путем развертывания новой версии ПО для подмножества наших пользователей.

По сути, в основе всех этих методов лежит простое изменение наших представлений о доставке программного обеспечения. А именно: что мы можем отделить концепцию развертывания от концепции выпуска.

Отделение развертывания от релиза

Джез Хамбл, соавтор книги «Непрерывная доставка», приводит доводы в пользу разделения этих двух идей и делает это основным принципом для выпусков программного обеспечения с низким уровнем риска (https://oreil.ly/VzpLc).

> Развертывание — это то, что происходит, когда вы устанавливаете некоторую версию своего программного обеспечения в определенную среду (часто подразумевается эксплуатационная среда). Релиз — это когда вы делаете систему или какую-то ее часть (например, функциональную возможность) доступной для пользователей.

Джез утверждает, что, разделив эти две идеи, можно гарантировать, что наше ПО будет работать в условиях эксплуатации без сбоев, замеченных пользователями. Сине-зеленое развертывание — один из простейших примеров данной концепции в действии: у вас есть одна версия программного обеспечения в режиме реального времени (синяя), а затем вы развертываете новую вместе со старой в среду эксплуатации (зеленая). Вы проверяете, работает ли новая версия должным образом, и, если это так, вы перенаправляете клиентский трафик в сторону новой версии вашего ПО. Если проблема будет обнаружена до полного перехода, это не повлияет ни на одного клиента.

Хотя сине-зеленые развертывания представляют собой один из самых простых примеров, существует ряд более сложных методов. Их можно использовать при принятии этой концепции.

Переходим к поэтапной доставке

Джеймс Гавернор, соучредитель отраслевой аналитической фирмы RedMonk, ориентированной на разработчиков, впервые придумал (https://oreil.ly/1nFrg) термин «поэтапная доставка» для охвата ряда различных методов, используемых в этой области. Далее он описал ее как «непрерывную доставку с полным контролем радиуса взрыва» (https://oreil.ly/opHOq). Получается, что это расширение непрерывной поставки, но также и технология, дающая нам возможность контролировать потенциальное воздействие недавно выпущенного ПО.

Подхватывая эту тему, Адам Зимман из LaunchDarkly описывает (https://oreil.ly/zeeNc), как поэтапная доставка влияет на «бизнес». С этой точки зрения нам требуется изменить представление о том, как новые функциональные возможности достигают наших клиентов. Это больше не единовременное развертывание — теперь это поэтапное действие. Однако важно отметить, что поэтапная доставка может расширить возможности владельца продукта, как выразился Адам, «делегируя контроль над функцией владельцу, который несет наибольшую ответственность за результат». Однако для того, чтобы это сработало, владелец продукта, о котором идет речь, должен понимать механику используемой технологии поэтапной доставки, подразумевая несколько технически подкованного владельца продукта или поддержку со стороны достаточно грамотных людей.

Мы уже коснулись сине-зеленых развертываний как одного из поэтапных методов доставки. Давайте кратко рассмотрим еще несколько стратегий.

Переключатели функций

С помощью переключателей функций (иначе известных как флаги функций) мы скрываем развернутую функциональность за переключателем, который можно использовать для выключения или включения функциональности. Этот метод чаще всего используется как часть магистральной разработки, когда еще не готовая функция может быть проверена и развернута, продолжая быть скрытой от конечных пользователей. Это может быть полезно для включения функции в указанное время или отключения ее, если она вызывает проблемы.

Возможно также использовать переключатели функций, используя более тонкий подход, позволяя флагу получить другое состояние в зависимости от характера пользователя, делающего запрос. Так, например, одна группа клиентов (или группа бета-тестирования) видит функцию включенной, в то время как все остальные — отключенной. Такой подход поможет вам внедрить канареечное развертывание, о котором мы поговорим далее. Существуют полностью управляемые решения для контроля переключателей функций, в том числе

LaunchDarkly (http://launchdarkly.com) и Split (https://www.split.io). Какими бы впечатляющими ни были эти платформы, я думаю, стоит начать с чего-то попроще, например, на первых порах хватит обыкновенного файла конфигурации, а когда начнете продвигать желаемые методы использования переключателей, посмотрите в сторону упомянутых технологий.

Для более глубокого погружения в мир переключателей я могу от всей души порекомендовать статью Пита Ходжсона «Переключатели функций (они же флаги функций)» (https://oreil.ly/5B9ie), в которой подробно рассказывается, как их реализовать и использовать.

Канареечный релиз

> Человеку свойственно ошибаться, но чтобы по-настоящему все испортить, нужен компьютер[1].

Мы все совершаем ошибки, и компьютеры позволяют нам совершать их быстрее и в большем масштабе, чем когда-либо прежде. Учитывая, что они неизбежны (это бесспорно), имеет смысл ограничить влияние этих ошибок. Одним из таких методов стал канареечный релиз.

Он назван в честь канареек, которых берут в шахты в качестве систем раннего предупреждения шахтеров о присутствии опасных газов. Идея канареечного развертывания заключается в том, что ограниченной группе клиентов доступны новые функции. Если возникает проблема с внедрением, это затрагивает только данную часть наших клиентов, но если функция работает бесперебойно, то ее можно распространить на большее количество пользователей, пока она не станет доступна всем.

Для микросервисной архитектуры переключатель настраивается на уровне отдельного микросервиса, включая (или выключая) функциональность для запросов к ней из внешнего мира или других микросервисов. Иной метод заключается в создании двух разных версий микросервиса, работающих параллельно, и использовании переключателя для маршрутизации от старой версии к новой и наоборот. Здесь канареечная реализация должна находиться где-то на пути маршрутизации/сети, а не в одном микросервисе.

Когда я впервые прибегнул к канареечному релизу, мы контролировали развертывание вручную и могли настраивать процент нашего трафика, видящего новую функциональность. В течение недели мы постепенно увеличивали этот показатель, пока новая функциональность не стала доступна для всех. На про-

[1] Эту цитату часто приписывают биологу Полу Эрлиху, но ее фактическое происхождение неясно (https://oreil.ly/3SOop).

тяжении недели мы следили за частотой ошибок, отчетами об ошибках и т. п. В настоящее время чаще всего данный процесс обрабатывается автоматически. Инструменты, подобные Spinnaker (https://www.spinnaker.io), например, получили возможность без внешнего вмешательства увеличивать количество вызовов на основе метрик, таких как увеличение процента вызовов новой версии микросервиса, если частота ошибок находится на приемлемом уровне.

Параллельное выполнение

При применении канареечного релиза запрос на часть функциональности будет обслуживаться либо старой, либо новой версией продукта. Это означает, что мы не можем сравнить, как два варианта функциональности будут обрабатывать один и тот же запрос. Но это важно, если требуется убедиться, что новая версия работает точно так же, как старая.

При параллельном запуске вы одновременно запускаете две разные модификации одной функциональности и отправляете запрос к обеим ее реализациям. При микросервисной архитектуре наиболее очевидным подходом может быть отправка вызова двум разным версиям одного и того же сервиса и сравнение результатов. Альтернативой может стать сосуществование обеих реализаций функциональности внутри одного сервиса, что часто упрощает сравнение.

При одновременном выполнении обеих реализаций важно понимать, что вам, скорее всего, нужны только результаты одного из вызовов. Одна реализация считается источником истины — реализация, которой вы в настоящее время доверяете, и, как правило, это существующая версия реализации. В зависимости от характера сравниваемой при параллельном запуске функциональности вам, возможно, придется тщательно продумать данный нюанс — вы же не хотите отправлять клиенту два одинаковых обновления заказа или, например, дважды оплачивать счет!

Более подробно о шаблоне параллельного выполнения можете прочесть в главе 3 моей книги «От монолита к микросервисам»[1]. Там я рассматриваю его использование для помощи в переносе функций из монолитной системы в микросервисную архитектуру, где необходимо убедиться, что наш новый микросервис ведет себя так же, как и в монолите. В другом контексте GitHub использует такой шаблон при переработке основных частей своей кодовой базы и выпустил инструмент с открытым исходным кодом Scientist (https://oreil.ly/LXtNJ), чтобы помочь в этом процессе. Здесь параллельный запуск выполняется в рамках одного процесса, а Scientist помогает сравнивать вызовы.

[1] *Ньюмен С.* От монолита к микросервисам. Эволюционные шаблоны для трансформации монолитной системы.

Рассмотрев сине-зеленое развертывание, переключатели функций, канареечные релизы и параллельные запуски, мы только слегка коснулись поверхности области поэтапной доставки. Эти идеи могут хорошо работать вместе (мы уже говорили, как использовать переключатели функций для реализации канареечного развертывания), но вы, вероятно, захотите облегчить себе задачу. Для начала просто не забудьте разделить концепции развертывания и релиза. Затем начните искать способы, которые позволят вам развертывать ПО чаще, но безопаснее. Поработайте с владельцем вашего продукта или другими заинтересованными сторонами бизнеса, чтобы понять, как некоторые из этих методов помогут вам работать быстрее, а также уменьшить количество сбоев.

Резюме

Итак, мы подняли здесь много вопросов. Давайте кратко подведем итоги, прежде чем двигаться дальше. Сначала напомним себе об изложенных ранее принципах развертывания.

Изолированное выполнение

Запускайте экземпляры микросервиса изолированным образом, чтобы у них были свои собственные вычислительные ресурсы и их выполнение не могло повлиять на другие экземпляры микросервиса, работающие поблизости.

Сосредоточьтесь на автоматизации

По мере увеличения количества микросервисов автоматизация становится все более важной. Сосредоточьтесь на выборе технологии, обеспечивающей высокую степень автоматизации, и сделайте ее ключевой частью своей культуры производства.

Инфраструктура как код

Представьте конфигурацию вашей инфраструктуры, чтобы упростить автоматизацию и обеспечить обмен информацией. Сохраните этот код в системе управления версиями, чтобы можно было воссоздавать среды.

Развертывание без простоя

Расширьте возможности независимого развертывания и убедитесь, что развертывание новой версии микросервиса может быть выполнено без каких-либо простоев для пользователей вашего сервиса (будь то люди или другие микросервисы).

Управление желаемым состоянием

Используйте платформу, поддерживающую ваш микросервис в определенном состоянии, при необходимости запуская новые экземпляры в случае сбоев или увеличения трафика.

Кроме того, я поделился своими собственными рекомендациями по выбору правильной платформы развертывания.

1. Не пытайтесь починить то, что не сломано.
2. Отдайте автоматике столько контроля, сколько хочется, а затем отдайте еще немного. Если вы можете переложить всю свою работу на хороший PaaS, такой как Heroku (или платформа FaaS), тогда сделайте это и будьте счастливы. Вам действительно нужно возиться с настройкой до последнего?
3. Контейнеризация микросервисов не безболезненная, но станет действительно хорошим компромиссом между стоимостью изоляции и некоторыми фантастическими преимуществами для локальной разработки, в то же время предоставляя вам определенную степень контроля над происходящим. Ожидайте появления Kubernetes в своем будущем.

Важно также понимать *свои* цели и потребности. Kubernetes отлично подошел бы вам, но, возможно, что-то попроще сработает не хуже. Не стыдитесь, что выбрали более простое решение, а также не беспокойтесь слишком сильно о том, что нужно переложить работу на кого-то другого. Если я могу перенести работу в публичное облако, я это сделаю, что в результате позволит мне сосредоточиться на собственных делах.

Прежде всего, эта сфера переживает время перемен. Я надеюсь, что дал вам некоторое представление о ключевых технологиях, а также поделился некоторыми принципами, которые, вероятно, переживут нынешнее изобилие популярных технологий. Что бы ни случилось дальше, надеюсь, вы теперь гораздо лучше подготовлены к всевозможным нововведениям.

В следующей главе мы углубимся в тему, которую кратко затронули здесь: тестирование микросервисов.

ГЛАВА 9

Тестирование

Мир автоматизированного тестирования получил значительное развитие с тех пор, как я начал писать код, и почти каждый месяц появляется какой-то новый инструмент или техника, совершенствующие тестирование. Но остаются проблемы, связанные с эффективностью проверок кода, охватывающего распределенную систему. В этой главе рассматриваются вопросы, связанные с тестированием более детализированных систем, и представлены некоторые решения, помогающие выпускать новые функции с уверенностью в их надежности.

Тестирование охватывает множество областей. Даже когда мы говорим *только* об автоматизированных тестах, нужно учитывать их большое количество. С использованием микросервисов мы добавили дополнительный уровень сложности. Понимание того, какие типы тестов можно выполнять, важно, чтобы помочь сбалансировать иногда противоположные силы: как можно быстрее выпустить свое ПО и убедиться, что оно достаточно качественное. Учитывая масштабы тестирования в целом, я не собираюсь широко исследовать эту тему. Вместо этого в текущей главе основное внимание уделим рассмотрению различий между тестированием микросервисной архитектуры и менее распределенной системой, такой как монолитное приложение с одним процессом.

Этап разработки, на котором проводилось тестирование, также изменился с момента выхода первого издания книги. Ранее оно осуществлялось в основном до выпуска ПО. Однако сейчас мы все чаще обращаем внимание на тестирование наших приложений в эксплуатации, что еще больше стирает границы между разработкой и мероприятиями по запуску продукта. Эти вопросы мы рассмотрим в текущей главе, прежде чем более подробно изучить тестирование в главе 10.

Типы тестов

Как и многие консультанты, я грешил использованием квадрантов для категоризации мира, и я забеспокоился, что в этой книге их не будет. К счастью, Брайан Марик придумал фантастическую систему категоризации тестов, которая идеально подходит. На рис. 9.1 показан вариант квадранта Марика из книги

Лайзы Криспин и Джанет Грегори «Agile-тестирование»[1], который помогает классифицировать различные типы тестов.

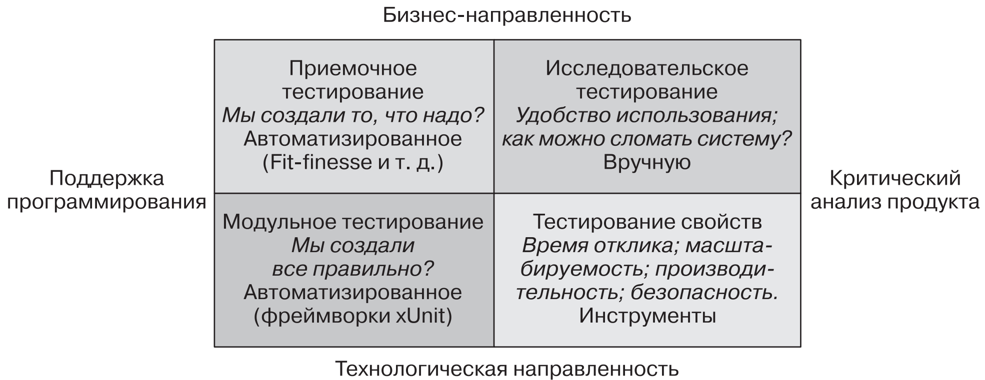


Рис. 9.1. Тестовый квадрант Брайана Марика. Лиза Криспин и Джанет Грегори, «Agile-тестирование: Практическое руководство для тестировщиков и Agile-команд» © 2009

В нижней части квадранта расположены тесты *технологической направленности*, то есть тесты, которые в первую очередь помогают разработчикам в создании системы. Тесты свойств, такие как тесты производительности и модульные тесты с небольшим охватом, попадают в эту категорию, все они обычно автоматизированы. Верхняя половина квадранта включает в себя те тесты, которые помогают нетехническим заинтересованным сторонам понять, как работает ваша система. Мы называем их *тестами бизнес-направленности*. Это могут быть комплексные тесты с широким охватом, как показано в квадранте «Приемочное тестирование» вверху слева, или тестирование вручную (типичным примером которого будет пользовательское тестирование, выполняемое в системе UAT), как показано в квадранте «Исследовательское тестирование».

На данном этапе стоит обратить внимание на тот факт, что подавляющее большинство таких тестов сосредоточено на *продакшен-валидации*. В частности, мы используем их, чтобы убедиться в достаточном качестве ПО перед его развертыванием в эксплуатационной среде. Как правило, успешность (или неудача) подобных тестов становится решающим условием для принятия решения о том, следует ли развертывать ПО.

Мы все чаще убеждаемся в ценности тестирования программного обеспечения, как только фактически начинаем работать в эксплуатационной среде. Подробнее о балансе между этими двумя идеями мы поговорим позже в текущей главе, а сейчас стоит обратить внимание на одно ограничение квадранта Марика.

---

[1] *Crispin L., Gregory J.* Agile Testing: A Practical Guide for Testers and Agile Teams. — Addison-Wesley, 2008.

Каждый тип теста, показанный в квадранте, имеет свое место. Точный объем каждого выполняемого тестирования будет зависеть от характера вашей системы, но ключевым моментом является наличие нескольких вариантов с точки зрения способа тестирования системы. В последнее время наблюдается тенденция отказа от любого крупномасштабного тестирования, выполняемого вручную, в пользу автоматизации как можно большего количества повторяющихся тестов, и я, безусловно, согласен с таким подходом. Если вы в настоящее время часто занимаетесь тестированием вручную, я бы посоветовал вам решить эту проблему, прежде чем идти дальше по пути микросервисов, поскольку вы не получите многих преимуществ этой архитектуры, если не сможете быстро и эффективно проверить свое ПО.

ИССЛЕДОВАТЕЛЬСКОЕ ТЕСТИРОВАНИЕ ВРУЧНУЮ

В целом переход от монолитной архитектуры к микросервисной окажет минимальное влияние на исследовательское тестирование, помимо любых более широких организационных изменений, которые могут произойти. Как описано в разделе «На пути к потоковым командам» главы 14, мы ожидаем, что пользовательский интерфейс приложения также будет разбит по командным линиям. В таком контексте ответственность за тестирование вручную вполне может измениться.

Возможность автоматизировать некоторые задачи, такие как проверка внешнего вида чего-либо, ранее ограничивалась исключительно исследовательским тестированием вручную. Развитие инструментария, позволяющего создавать визуальные утверждения, позволило начать решать задачи без непосредственного участия человека. Однако не стоит рассматривать это как причину отказа от тестирования вручную. Рассмотрите это как шанс освободить время тестировщиков, чтобы сосредоточиться на *исследовательском* тестировании, например.

Если все сделано правильно, исследовательское тестирование вручную в значительной степени становится открытием. Выделение времени на изучение приложения конечным пользователем может выявить проблемы, которые в противном случае были бы незаметны. Тестирование вручную также может быть жизненно важным, когда автоматизированный тест невозможно реализовать, например, из-за стоимости написания теста. Автоматизация заключается в устранении повторяющихся задач, чтобы освободить людей для выполнения более творческих, разовых действий. Поэтому думайте об автоматизации как о способе высвободить интеллектуальные ресурсы для того, что мы делаем лучше всего.

При работе над целями данной главы мы в основном пропускаем исследовательское тестирование вручную. Это не значит, что такой тип тестирования неважен, просто цель главы — сосредоточиться в первую очередь на том, чем тестирование микросервисов отличается от тестирования более типичных монолитных приложений. Но, когда речь заходит об автоматизированных тестах, сколько тестов каждого вида нам требуется провести? Следующая модель поможет нам ответить на этот вопрос и понять, какие различные компромиссы нас поджидают.

Охват тестирования

В своей книге «Преуспевайте с Agile»[1] Майк Кон описывает модель под названием «пирамида тестов», которая помогает объяснить, какие типы автоматизированных тестов необходимы. Пирамида помогает продумать не только охват тестов, но и пропорции различных типов тестов, к которым стоит стремиться. Оригинальная модель Кона разделила автоматизированные тесты на модульные, сервисные и UI-тесты, как показано на рис. 9.2.

Рис. 9.2. Пирамида тестов Майка Кона

При чтении пирамиды учитывайте, что по мере того, как мы поднимаемся по пирамиде, объем тестирования увеличивается, как и наша уверенность в работоспособности функций. С другой стороны, время цикла обратной связи растет вместе с длительностью выполнения тестов и в случае сбоя теста будет сложнее определить, что нарушено. Если мы спускаемся по пирамиде, в целом тесты становятся намного быстрее, как и циклы обратной связи. Мы быстрее находим неработающие функции, наши непрерывные интеграционные сборки так же выполняются легче, и меньше шансов перейти к новой задаче, прежде чем мы обнаружим, что что-то сломали. Когда эти тесты с меньшим охватом терпят неудачу, мы, скорее всего, будем знать, что сломалось, часто вплоть до точной строки кода, — каждый тест лучше *изолирован*, что облегчает понимание и устранение неисправностей. С другой стороны, нет большой уверенности в том, что система в целом работает, если мы протестировали только одну строку кода!

[1] *Cohn M.* Succeeding with Agile. — Addison-Wesley, 2009.

Проблема подобной модели, что все эти термины означают разные вещи для разных людей. Понятие «сервис» особенно перегружено, также существует множество определений модульного теста. Тестирование одной строки — модульный тест? Я бы сказал, что да. А тестирование нескольких функций или классов? Я бы сказал «нет», но многие не согласятся! Я склонен придерживаться названий «модули» и «сервисы», несмотря на их неоднозначность, но предпочитаю называть *UI-тесты сквозными тестами*, что я и буду делать с этого момента.

Практически каждая команда, с которой я работал, использовала для тестов иные названия, чем Кон в своей пирамиде. Как бы вы их ни называли, вам понадобятся функциональные автоматизированные тесты разного объема для различных целей.

Давайте посмотрим на отработанный пример. На рис. 9.3 показаны приложение службы поддержки и основной веб-сайт. Они оба взаимодействуют с микросервисом `Покупатель` для получения, просмотра и редактирования сведений о потребителе. Микросервис `Покупатель`, в свою очередь, взаимодействует с сервисом `Лояльность`, где наши клиенты накапливают баллы, покупая компакт-диски Джастина Бибера. Возможно. Очевидно, что это фрагмент общей системы MusicCorp, но это достаточно показательный фрагмент для погружения в несколько различных сценариев, которые может потребоваться протестировать.

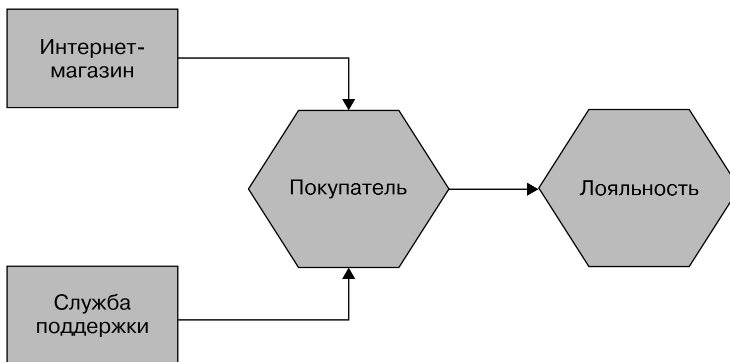


Рис. 9.3. Часть музыкального магазина в стадии тестирования

Модульное тестирование

Модульные тесты (или юнит-тесты) обычно проверяют один вызов функции или метода. В эту категорию попадают тесты, созданные в качестве побочного эффекта *разработки через тестирование* (test-driven design, TDD), как и тесты, созданные с помощью таких методов, как тестирование на основе свойств.

Мы не запускаем здесь микросервисы и ограничиваем использование внешних файлов или сетевых подключений. В общем, потребуется большое количество подобных проверок. Если все сделано правильно, они работают молниеносно, а на современном оборудовании тысячи подобных тестов запустятся менее чем за минуту. У многих людей эти тесты выполняются автоматически при локальном изменении файлов. С интерпретируемыми языками это может дать очень быстрые циклы обратной связи.

Юнит-тесты помогают разработчикам и ориентированы на технологии, а не на бизнес (согласно терминологии Марика). Кроме того, именно с их помощью мы надеемся обнаружить большинство своих ошибок.

Итак, в нашем примере модульные тесты в отношении микросервиса `Покупатель` будут охватывать небольшие части кода изолированно, как показано на рис. 9.4.

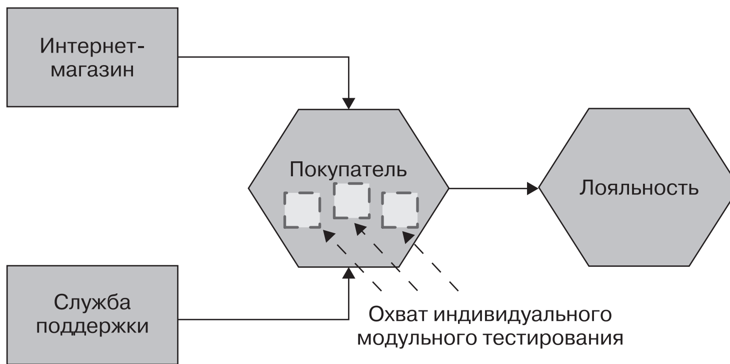


Рис. 9.4. Охват модульных тестов в нашем примере системы

Основная цель этих тестов — дать нам очень быструю обратную связь о качестве созданной функциональности. Юнит-тесты также важны для поддержки рефакторинга кода, так как позволяют реструктурировать код по мере продвижения. Появляется уверенность, что тесты с небольшим охватом подстрахуют нас в случае ошибки.

Сервисное тестирование

Сервисные тесты предназначены для обхода UI и непосредственного тестирования микросервисов. В монолитном приложении можно было бы просто тестировать набор классов, предоставляющих *сервис* пользовательскому интерфейсу. Для системы, состоящей из нескольких микросервисов, подобный тест будет проверять возможности отдельного микросервиса.

Выполняя таким образом тесты для одного микросервиса, мы больше уверены в том, что сервис будет вести себя так, как ожидается, но область тестирования все еще остается несколько изолированной. Причина появления сбоя теста должна быть в рамках только тестируемого микросервиса. Чтобы добиться такой изоляции, необходимо отключить все внешние связанные блоки, чтобы в поле зрения попал только сам микросервис, как показано на рис. 9.5.

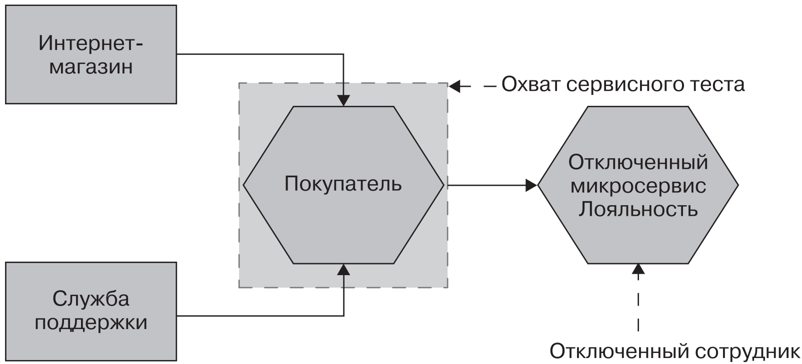


Рис. 9.5. Охват сервисных тестов на нашем примере системы

Некоторые подобные тесты могут быть такими же быстрыми, как и юнит-тесты с небольшим охватом. Но если вы решите протестировать их на реальной БД или перейти по сети к заблокированным нижестоящим блокам, время тестирования, скорее всего, увеличится. Эти тесты также охватывают больший объем, чем простой модульный, поэтому, когда они терпят неудачу, определить, что именно привело к провалу, может быть сложнее. Однако у них значительно меньше активных частей, и, следовательно, такое тестирование менее уязвимое, чем тесты с большим охватом.

Сквозное тестирование

Сквозные тесты выполняются для всей вашей системы целиком. Часто они управляют графическим интерфейсом через браузер, но также могут легко имитировать другие виды взаимодействия с пользователем, например загрузку файла.

Эти тесты охватывают большой объем кода, как показано на рис. 9.6. Поэтому вы почувствуете облегчение, когда они успешно завершатся, гарантируя, что тестируемый код будет работать при запуске в эксплуатацию. Но такой увеличенный охват имеет свои недостатки, и, как мы вскоре увидим, сквозные тесты могут быть очень сложными.

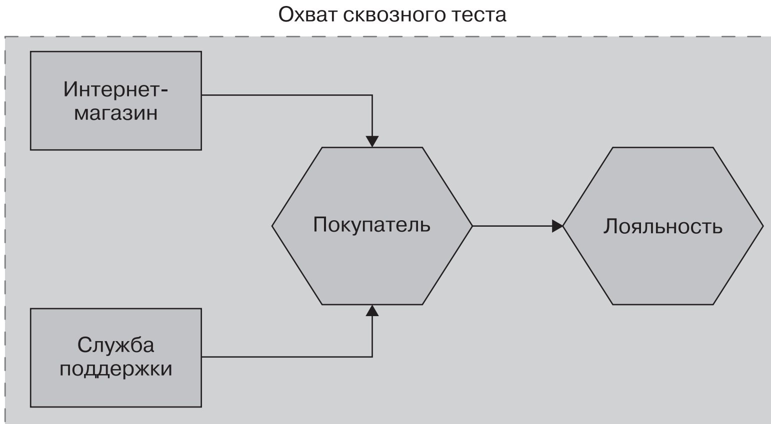


Рис. 9.6. Охват сквозных тестов на нашем примере системы

А КАК НАСЧЕТ ИНТЕГРАЦИОННЫХ ТЕСТОВ?

Наверняка вы заметили, что я не стал явно описывать интеграционные тесты. Это сделано специально. Я обнаружил, что данный термин часто используется разными людьми для описания разных типов тестирования. Для некоторых интеграционный тест — это только взаимодействие между двумя сервисами или, возможно, привязка между кодом и БД. Для других это то же самое, что и полноценные сквозные тесты. В текущей главе я старался использовать более четкие определения. Надеюсь, вам будет несложно сопоставить то, что вы называете интеграционным тестом, с терминами, которые я использую здесь.

Компромиссы

То, к чему мы стремимся, используя различные типы тестов, которые охватывает пирамида, — это разумный баланс. Нам нужны быстрая обратная связь и гарантии работоспособности системы.

Юнит-тесты невелики по охвату, поэтому, когда они терпят неудачу, можно быстро найти проблему. Они также быстры в написании и *очень* быстры в запуске. По мере увеличения охвата тестов мы становимся более уверенными в нашей системе, но обратная связь начинает ухудшаться, поскольку продолжительность выполнения тестов возрастает. Кроме того, их написание и обслуживание обходятся дороже.

Часто приходится искать баланс количества требующихся тестов каждого типа, чтобы найти золотую середину. Обнаружили, что запуск вашего набора тестов занимает слишком много времени и терпит неудачу? Напишите юнит-тест, чтобы быстрее выявить причину. Попробуйте заменить некоторые тесты с большим охватом (и более медленные) на более быстрые с меньшим. С другой

стороны, когда с ошибкой сталкивается потребитель, это может быть признаком, что вы упустили какой-то тест.

Итак, если у всех этих тестов есть компромиссы, сколько всего их нужно? Хорошее эмпирическое правило заключается в том, что вам, вероятно, будет требоваться на порядок больше тестов по мере спуска по пирамиде. Важно знать, что есть различные типы автоматических тестов, и понимать, вызывает ли у вас проблемы текущий баланс тестов!

Например, я работал над одной монолитной системой, где у нас было 4000 модульных тестов, 1000 сервисных и 60 сквозных. Мы решили, что с точки зрения обратной связи у нас было слишком много сервисных и сквозных тестов (последние из которых были злостными нарушителями, влияющими на циклы обратной связи), поэтому мы усердно работали над заменой тестового покрытия тестами с меньшим охватом.

Распространенным антипаттерном является так называемый *тестовый снежный конус* или перевернутая пирамида. Здесь практически отсутствуют тесты с малым охватом, так как все покрытие обеспечивается тестами с большим охватом. У таких проектов часто чрезвычайно медленные тестовые прогоны и очень длительные циклы обратной связи. Если такие проверки выполняются в рамках непрерывной интеграции, у вас не будет много сборок, а характер времени сборки означает, что система может оставаться неработающей в течение длительного периода, когда что-то действительно ломается.

Внедрение сервисных тестов

Внедрение юнит-тестов — довольно простое дело, и существует множество документов, где объясняется, как их писать. Сервисные и сквозные тесты наиболее интересны, особенно в контексте микросервисов, так что сосредоточимся на них.

Нашим сервисным тестам требуется протестировать часть функций во всем микросервисе, и только в нем. Итак, если потребуется написать тест микросервиса `Покупатель` с рис. 9.3, мы развернем экземпляр микросервиса `Покупатель` и, как обсуждалось ранее, «отрежем» микросервис `Лояльность`, чтобы получить гарантии, что сбой при выполнении теста можно сопоставить с проблемой самого микросервиса `Покупатель`.

Как рассматривалось в главе 7, как только мы проверим необходимое ПО, одним из первых действий нашей автоматизированной сборки будет создание бинарного артефакта для микросервиса, например формирование образа контейнера для данной версии ПО. Так что развернуть это довольно просто. Но как быть с заглушками нижестоящих потребителей?

Нашему набору сервисных тестов необходимо отключить нижестоящие сервисы и настроить тестируемый микросервис для подключения к сервисам-заглушкам. Затем нам потребуется настроить заглушки для отправки ответов, имитирующих реальные микросервисы.

Макетирование или заглушки

Когда я говорю об отключении ненужных функций, я имею в виду, что мы создаем микросервис-заглушку, отвечающую заготовленными ответами на известные запросы от тестируемого микросервиса. Например, я мог бы сообщить заблокированному микросервису `Лояльность`, что при запросе баланса клиента 123 он должен вернуть значение 15 000. Тесту все равно, вызывается ли заглушка 0, 1 или 100 раз. Разновидностью этого подхода является использование макета вместо заглушки.

Использование макета позволяет убедиться, что вызов был сделан. Если ожидаемый вызов не выполняется, тест завершается неудачей. Реализация такого подхода требует больше «сообразительности» от наших псевдосервисов, и при чрезмерном использовании это может привести к уязвимости тестов. Однако, как уже отмечалось, заглушке все равно, вызывается ли она 0, 1 или много раз.

Иногда, однако, макеты могут быть очень полезны, чтобы гарантировать появление ожидаемых побочных эффектов. Например, если требуется проверить, устанавливается ли для вновь созданного покупателя баланс баллов. Баланс между вызовами заглушек и макетов достаточно хрупкий и так же опасен в сервисных тестах, как и в модульных. В целом, однако, я использую заглушки гораздо чаще, чем макеты для сервисных тестов. Для более глубокого обсуждения этого компромисса взгляните на книгу «Развитие объектно-ориентированного программного обеспечения под влиянием тестов» Стива Фримена и Нэта Прайса[1].

В общем, я редко использую макеты для такого рода тестирования. Но наличие инструмента, способного реализовывать и макеты, и заглушки, полезно.

Хотя заглушки и макеты на самом деле довольно сильно различаются, я знаю, что это различие может сбить некоторых с толку, особенно тех, кто использует другие термины, такие как *подделки*, *шпионы* и *манекены*. Джерард Месзарос называет все эти вещи, в том числе заглушки и макеты, тестовыми дублерами (https://oreil.ly/8Pp2y).

Более самостоятельная сервис-заглушка

Обычно я сам развертывал сервисы-заглушки. Для их запуска я использовал все: от веб-сервера Apache или nginx до встроенных контейнеров Jetty или даже веб-серверов Python, запускаемых из командной строки. Вероятно, я снова и снова выполнял одну и ту же работу при создании этих заглушек. Мой старый коллега по Thoughtworks, Брэндон Байарс, потенциально избавил многих из нас от огромного количества работы с помощью своего сервера заглушек/макетов под названием mountebank (http://www.mbtest.org).

[1] *Freeman S., Pryce N.* Growing Object-Oriented Software, Guided by Tests. — Addison-Wesley, 2009.

Mountebank можно расценивать как небольшое программное приспособление, программируемое через HTTP. Ни один вызывающий сервис не поймет, что оно написано на NodeJS. Когда mountebank запускается, вы отправляете команды, указывающие ему создать один или несколько «самозванцев», которые будут отвечать на заданный порт с помощью определенного протокола (в настоящее время поддерживаются TCP, HTTP, HTTPS и SMTP), и то, какие ответы эти самозванцы должны отправлять при отправке запросов. Эта программа также поддерживает настройку ожиданий, если вы хотите использовать ее в качестве макета. Поскольку один экземпляр mountebank поддерживает создание нескольких самозванцев, допускается использовать его для отключения нескольких нижестоящих микросервисов.

Mountebank действительно может применяться за пределами автоматизированного функционального тестирования. Например, компания Capital One использовала mountebank для замены существующей макетной инфраструктуры для своих крупномасштабных тестов производительности[1].

Одним из ограничений mountebank стало отсутствие поддержки заглушек для протоколов обмена сообщениями. Например, если вы хотите убедиться, что событие было правильно отправлено (и, возможно, получено) через брокер, вам придется искать решение в другом месте. Это одна из областей, где мог бы помочь инструмент под названием Pact (который мы рассмотрим чуть позже).

Поэтому, если требуется запускать тесты только для микросервиса `Покупатель`, можно запустить и его, и экземпляр mountebank, выступающий в качестве микросервиса `Лояльность`, на той же машине. И если эти тесты пройдут успешно, можно сразу же развернуть сервис `Покупатель`! Или нельзя? Как насчет вызывающих микросервис `Покупатель` сервисов — службы поддержки и интернет-магазины? Не внесли ли мы изменения, способные нарушать их работу? Конечно, мы забыли о важных тестах на вершине пирамиды — сквозных тестах.

Внедрение (этих сложных) сквозных тестов

В микросервисной системе возможности, которые мы предоставляем через UI, поставляются рядом микросервисов. Цель сквозных тестов, описанных в пирамиде Майка Кона, — сопоставить функциональность этих пользовательских интерфейсов со всем, что находится под ними, чтобы дать нам некоторую обратную связь о качестве системы в целом.

---

[1] *Valentino J. D.* Moving One of Capital One’s Largest Customer-Facing Apps to AWS // Capital One Tech. 24 мая 2017 года. https://oreil.ly/5UM5W.

Итак, чтобы реализовать сквозной тест, необходимо развернуть несколько микросервисов вместе, а затем запустить для них тест. Очевидно, что у этого теста гораздо больший охват, что придает уверенности в работоспособности нашей системы! С другой стороны, такие тесты медленнее и затрудняют диагностику сбоя. Давайте углубимся в них подробнее, используя предыдущий пример, чтобы увидеть, как эти тесты вписываются в общую картину.

Представьте, что требуется выпустить новую версию микросервиса `Покупатель`. Нам надо провести изменения в рабочую версию как можно скорее, но мы сомневаемся, что внесенное изменение не нарушит работу службы поддержки или веб-магазина. Нет проблем — давайте развернем все сервисы сразу и проведем несколько тестов в службе поддержки и интернет-магазине, чтобы узнать, не закралась ли ошибка. Наивно было бы просто добавить эти тесты в конец конвейера обслуживания клиентов, как показано на рис. 9.7.

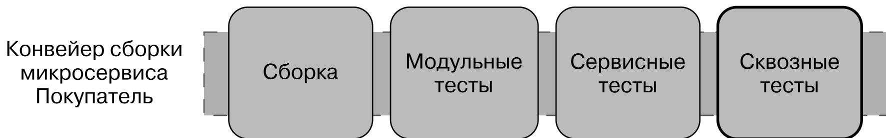


Рис. 9.7. Добавление этапа сквозных тестов: правильный подход?

Пока все идет хорошо. Но нам стоит сразу же задать себе вопрос: какую версию других микросервисов следует использовать? Должны ли мы запускать тесты с версиями службы поддержки и веб-магазина, которые находятся в эксплуатации? Это разумное предположение, но что, если новая версия службы поддержки или интернет-магазина стоит в очереди на запуск в эксплуатацию? Что же нам тогда делать?

Вот еще одна проблема: если у нас есть набор сквозных тестов сервиса `Покупатель`, которые развертывают множество микросервисов и выполняют тесты с ними, как насчет сквозных тестов, выполняющихся другими микросервисами? Если они тестируют одно и то же, мы можем обнаружить, что покрываем много одинаковых областей и дублируем значительную часть работы по развертыванию всех этих микросервисов.

Чтобы прекрасно справиться с обеими этими проблемами, достаточно подключить несколько конвейеров «веером» к одному этапу сквозного тестирования. В таком случае, когда запускается одна из нескольких различных сборок, это приведет к запуску общих шагов сборки. Например, на рис. 9.8 показано, что успешная сборка для любого из четырех микросервисов приведет к запуску этапа общих сквозных тестов. Некоторые инструменты CI с улучшенной поддержкой конвейера сборки позволяют создавать подобные веерные модели «из коробки».

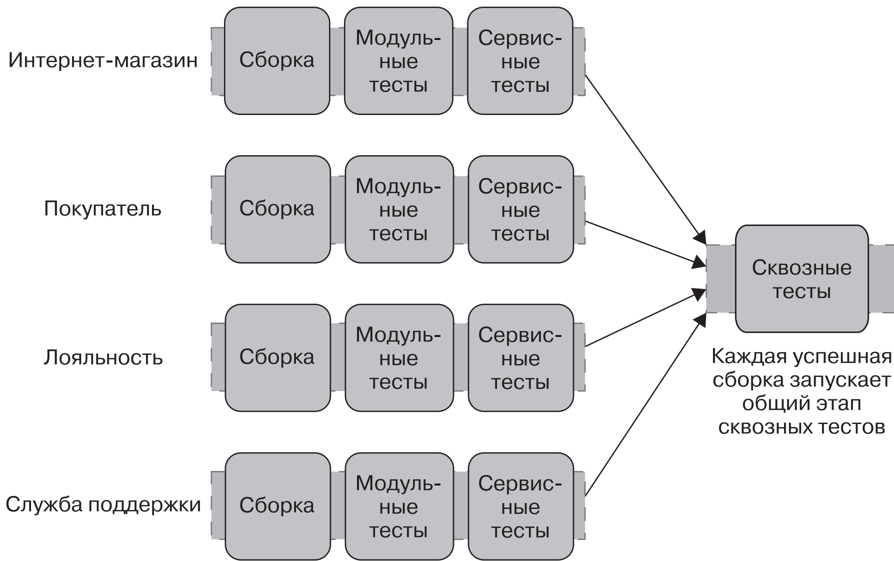


Рис. 9.8. Стандартный способ обработки сквозных тестов в разных сервисах

Поэтому каждый раз, когда меняется один из сервисов, мы запускаем локальные тесты для этого сервиса. Если проверки проходят успешно, мы запускаем интеграционные тесты. Здорово же? Что ж, к сожалению, у сквозного тестирования есть много недостатков.

Хрупкие и flaky-тесты

По мере расширения охвата тестирования увеличивается и количество подвижных элементов, которые могут привести к сбоям в тестировании. Такие элементы могут привести к сбоям в тестировании, не отображающим, что тестируемая функциональность нарушена, но указывают на то, что произошла какая-то другая проблема. Например, если у нас есть тест для проверки возможности разместить заказ на один компакт-диск и мы запускаем его на четырех или пяти микросервисах, если какой-либо из них не работает, может произойти сбой, не имеющий ничего общего с характером самого теста. Аналогично временный сбой в сети может привести к провалу теста, не дав информации о тестируемой функциональности.

Чем больше подвижных элементов, тем более хрупкими могут быть тесты и тем они менее детерминированы. Если некоторые из ваших тестов *иногда* заканчиваются неудачей, но их просто перезапускают, потому что позже они

выполняются успешно, значит, у вас есть flaky-тесты (ненадежные, нестабильные). И в этом виноваты не только тесты, охватывающие множество различных процессов. Тесты, которые охватывают функциональность, выполняемую в нескольких потоках (и в нескольких процессах), также часто становятся проблематичными. Сбой может означать состояние гонки, тайм-аут, или что функциональность действительно нарушена. Flaky-тесты — это враг. Их запуск мало что говорит нам. Мы повторно запускаем CI-сборки в надежде, что они позже пройдут тест успешно, но только для того, чтобы увидеть, как накапливаются фиксации кода, и внезапно мы оказываемся с кучей неработающих функций.

При обнаружении flaky-тестов очень важно сделать все возможное, чтобы исключить их. В противном случае мы начинаем терять уверенность в наборе тестов, которые «всегда так работают». Набор нестабильных тестов может стать жертвой того, что Диана Воган называет «нормализацией отклонений». И со временем мы рискуем настолько привыкнуть к беспорядку, что будем воспринимать это как норму, а не как проблему[1]. Это человеческий фактор. Нам нужно как можно скорее найти и устранить эти flaky-тесты, прежде чем мы привыкнем к провальным тестам.

В книге «Искоренение недетерминизма в тестах»[2] Мартин Фаулер отстаивает подход, согласно которому, если у вас есть нестабильные тесты, вы должны их отследить и, если не получается немедленно исправить, убрать из набора. Подумайте, что можно с ними сделать: переписать, заменить другим тестом с меньшим охватом или сделать базовую среду более стабильной. В некоторых случаях изменение тестируемого ПО, чтобы упростить его тестирование, также может быть правильным шагом к решению проблемы.

Кто пишет эти сквозные тесты

Тесты, которые выполняются как часть конвейера для конкретного микросервиса, разумно будет написать владеющей сервисом команде (подробнее о владении сервисом мы поговорим в главе 15). Но если принять во внимание, что может быть задействовано несколько команд и этап сквозных тестов теперь эффективно распределяется между ними, кто будет писать эти тесты?

Я сталкивался с подобными ситуациями. Такие тесты становятся открытыми для всех, и командам предоставляется доступ к добавлению тестов без какого-либо понимания работоспособности всего набора. Это часто приводит

---

[1] *Vaughan D.* The Challenger Launch Decision: Risky Technology, Culture, and Deviance at NASA. — Chicago: University of Chicago Press, 1996.

[2] *Fowler M.* Eradicating Non-Determinism in Tests // martinfowler.com. 14 апреля 2011 года. https://oreil.ly/7Ve7e.

к всплеску количества тестовых примеров, иногда достигая кульминации в виде тестового снежного конуса, о котором мы говорили ранее. Я также видел ситуации, в которых из-за отсутствия фактической очевидной принадлежности этих тестов кому-то конкретному их результаты игнорировались. Когда они завершаются неудачей, все предполагают, что это чья-то чужая проблема, поэтому их не волнует, успешно ли выполняется тестирование.

Одно из решений, которое я встречал, — это назначить определенные сквозные тесты ответственностью конкретной команды, даже если они могут пересекаться с микросервисами, над которыми работают несколько разных команд. Впервые я узнал об этом подходе от Эмили Баше[1]. Идея заключается в том, что, хотя мы используем «веерный» этап в нашем конвейере, набор сквозных тестов разделяется на группы функций, которые принадлежат разным командам, как показано на рис. 9.9.

В данном примере изменение в сервисе `Интернет-магазин`, прошедшее стадию сервисного тестирования, приведет к запуску связанных сквозных тестов, причем набор тестов будет принадлежать команде, владеющей сервисом `Интернет-магазин`. Аналогично любые изменения только в сервисе `Служба поддержки` приведут к запуску соответствующих сквозных тестов. Но изменения в сервисе `Покупатель` или `Лояльность` запускают *оба* набора тестов. Это может привести нас к ситуации, когда изменение, внесенное в микросервис `Лояльность`, нарушит оба набора сквозных тестов, что потенциально потребует от команд, владеющих ими, обратиться к владельцу микросервиса `Лояльность` за исправлением. Хотя в случае Эмили эта модель помогла, очевидно, что у нее все еще есть свои проблемы. По сути, сомнительно возлагать на команду ответственность за тесты, когда действия людей из другой команды могут привести к сбою этих тестов.

Иногда организации выделяют специальную команду для написания тестов. Это может иметь катастрофические последствия. Команда, разрабатывающая ПО, все больше отдаляется от тестов своего кода. Время цикла увеличивается, поскольку владельцы сервисов в итоге ждут, пока команда тестирования напишет сквозные тесты для только что увидевшей свет функциональности. Поскольку эти тесты составляет другая команда, написавшая сервис команда меньше вовлечена в работу и, следовательно, с меньшей вероятностью знает, как запускать и исправлять эти тесты. Хотя, к сожалению, это все еще распространенная организационная модель, я вижу значительный вред от того, что команда дистанцируется от написания тестов для своего кода.

---

[1] *Bache E.* End-to-End Automated Testing in a Microservices Architecture — Emily Bache // Конференция NDC, 5 июля 2017 года, видео на YouTube, 56:48, NDC Oslo 2017, https://oreil.ly/QX3EK.

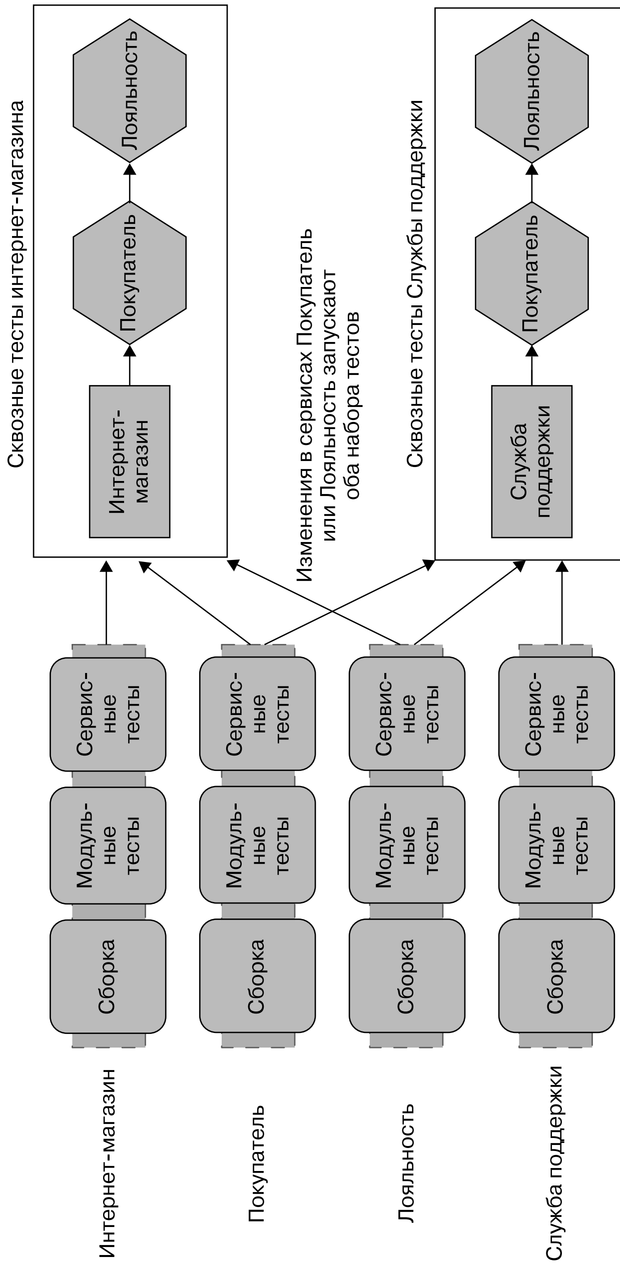


Рис. 9.9. Стандартный способ обработки сквозных тестов в разных сервисах

Разобраться в этом действительно сложно. Не следует дублировать выполняемую работу и полностью централизовать данный процесс до такой степени, чтобы команды, создающие сервисы, стали слишком далеки от вопроса тестирования. Если вы можете найти простой способ делегировать сквозные тесты определенному коллективу, то сделайте это. Если нет и если невозможно найти способ удалить сквозные тесты и заменить их чем-то другим, то вам, вероятно, придется рассматривать сквозной набор тестов как общую кодовую базу с совместным владением. Команды могут свободно производить фиксации кода в этом наборе тестов, но ответственность за работоспособность набора должна быть разделена между командами, разрабатывающими сами сервисы. Если вы хотите широко использовать сквозные тесты с несколькими командами, я думаю, что такой подход необходим. И все же я видел, что это делается очень редко и всегда с некоторым списком проблем. В конечном счете я убежден, что на определенном уровне развития организации вам стоит отойти от сквозных тестов, выполняемых командами совместно.

Как долго должны выполняться сквозные тесты

Сквозные тесты могут занять некоторое время. Я видел, как их запуск занимал до одного дня, если не больше, а в одном проекте, над которым я работал, полный цикл регрессионных тестов занял шесть недель! Редко встречаются команды, действительно курирующие свои наборы сквозных тестов, чтобы уменьшить дублирование в тестовом покрытии, или выделяющие достаточно времени, чтобы сделать тесты быстрыми.

Эта медлительность в сочетании с тем, что тесты часто рискуют оказаться ненадежными, может стать серьезной проблемой. Тесты, которые занимают весь день и часто завершаются сбоями, не имеющими ничего общего с нарушенной функциональностью, — это катастрофа. Даже если ваша функциональность нарушена, вам может потребоваться много часов, чтобы выяснить это. В этот момент вы, скорее всего, уже перешли бы к другой работе вместо того, чтобы возвращаться к началу и исправлять проблему.

Можно улучшить некоторые из тестов, выполняя их параллельно. Например, с помощью Selenium Grid. Однако такой подход не заменяет реального понимания того, что необходимо протестировать, и активного *удаления* ненужных тестов.

Удаление тестов иногда чревато и сродни отмене определенных мер безопасности в аэропортах. Независимо от того, насколько неэффективны меры безопасности, любой разговор об их отмене часто сопровождается резкой реакцией, якобы вы не заботитесь о безопасности людей или хотите победы террористов. Трудно вести конструктивное обсуждение о том, какую пользу приносит то или иное мероприятие, по сравнению с тем бременем, которое оно влечет за собой. Это также может быть трудным компромиссом между риском и прибылью.

Вас поблагодарят, если вы удалите тест? Возможно. Но вас наверняка обвинят, если удаленный тест допустит появление ошибки. Однако, когда дело доходит до наборов тестов с более широким охватом, это именно то, что нужно уметь делать. Если одна и та же функция рассматривается в 20 различных тестах, возможно, получится избавиться от половины проверок, чтобы сэкономить время! Для этого требуется качественная оценка рисков, в чем люди, как известно, плохо разбираются. В результате такое интеллектуальное курирование и управление тестами с большим охватом и высокой нагрузкой происходит невероятно редко. Желать, чтобы люди делали это чаще, и заставить их — не одно и то же.

Великое нагромождение

Длительные циклы обратной связи, связанные со сквозными тестами, представляют собой проблему не только для продуктивности разработчиков. При длинном наборе тестов любые сбои отвлекают на себя внимание, что увеличивает общее время прохождения сквозных тестов. Если мы развертываем только то ПО, которое успешно прошло все тесты (что мы и должны делать!), это означает, что меньшее количество наших сервисов доходит до потребителя.

Это может привести к нагромождению. Пока исправляются проблемы отказавшего этапа интеграционного тестирования, может появиться больше изменений от вышестоящих команд. Помимо того что это усложнит исправление сборки, так еще и объем развертываемых изменений увеличивается. Идеальный способ справиться с этим — не позволять людям осуществлять фиксации, если сквозные тесты терпят неудачу, но, учитывая длительное время их выполнения, это часто оказывается непрактично. Попробуйте сказать: «Вы, 30 разработчиков: никаких фиксаций изменений, пока мы не исправим эту семичасовую сборку!» Однако, если вы разрешаете провести фиксации кода на неработающей сборке, она может оставаться такой дольше, что подрывает ее эффективность как способа быстрой обратной связи о качестве кода. Правильный ответ — сделать набор тестов быстрее.

Чем больше масштаб развертывания и чем выше вероятность выпуска, тем больше шансов что-то сломать. Поэтому хочется быть уверенными, что мы сможем часто выпускать небольшие, хорошо протестированные изменения. Когда сквозные тесты замедляют нашу способность выпускать небольшие изменения, они приносят больше вреда, чем пользы.

Метаверсия

На этапе сквозных тестов легко задуматься: «Я знаю, что все эти сервисы в текущих версиях работают вместе, так почему бы не развернуть их разом?» Что, в конце концов, очень быстро превращается в: «Так почему бы не использовать

номер версии для всей системы?» Цитируя Брэндона Байарса (https://oreil.ly/r7Mzz): «Теперь у вас 2.1.0 проблем».

Объединяя изменения, внесенные в несколько сервисов, мы фактически принимаем идею о том, что изменение и развертывание нескольких сервисов одновременно является приемлемым. Это становится нормой. При этом мы упускаем одно из главных преимуществ микросервисной архитектуры: возможность независимого развертывания.

Слишком часто подход, допускающий одновременное развертывание нескольких сервисов, приводит к ситуации, когда сервисы становятся связанными. Вскоре эти отдельные сервисы сильнее переплетутся между собой, и вы этого не заметите, потому что не пытаетесь развернуть их по отдельности. В итоге у вас возникает путаница, когда приходится организовывать развертывание нескольких сервисов одновременно. Как мы обсуждали ранее, такого рода связанность может привести к худшему положению, чем при использовании одного монолитного приложения.

Это плохо.

Отсутствие возможности независимого тестирования

Мы часто возвращались к теме независимого развертывания, которая является важным свойством, помогающим командам выполнять задачи более автономно, позволяя эффективнее поставлять ПО. Если ваши команды работают независимо, значит, они должны иметь возможность проводить и независимое тестирование. Как мы уже видели, сквозные тесты могут снизить автономию команд и способствуют повышению уровня взаимодействия, что может повлечь за собой определенного рода проблемы.

Стремление к независимой тестируемости распространяется и на использование инфраструктуры, связанной с тестированием. Часто я вижу, как людям приходится использовать общие среды тестирования, в которых выполняются тесты от нескольких команд. Такая среда часто сильно ограничена, и любая проблема может привести к серьезным последствиям. В идеале, если вы хотите, чтобы ваши команды вели разработку и тестирование независимо, обеспечьте их собственными средами тестирования.

Исследование, обобщенное в книге «Ускоряйся!», показало, что высокопроизводительные команды с большей вероятностью «проводят большую часть тестирования по запросу, не требуя интегрированной тестовой средой»[1].

[1] *Форсгрен Н., Хамбл Дж., Ким Дж.* Ускоряйся! Наука DevOps: Как создавать и масштабировать высокопроизводительные цифровые организации. — Альпина PRO.

Следует ли избегать сквозных тестов

Несмотря на только что описанные недостатки, для многих пользователей сквозные тесты все еще могут быть управляемы при небольшом количестве микросервисов, и в этих ситуациях они по-прежнему хороши. Но что происходит с 3, 4, 10 или 20 сервисами? Очень быстро эти наборы тестов становятся чрезвычайно раздутыми, и в худшем случае они приведут к декартову взрыву в тестируемых сценариях.

На самом деле даже при небольшом количестве микросервисов эти тесты усложняются, если у вас есть несколько команд, которые используют их совместно. С помощью общего сквозного набора тестов вы подрываете свою цель независимого развертывания. Теперь вместо развертывания микросервисов вашей команде требуется проходить набор тестов, который используют не только они.

Какую ключевую проблему мы пытаемся решить с помощью сквозных тестов? Мы стараемся гарантировать, что наши изменения не повредят потребителям. В подразделе «Структурные и семантические разрывы контрактов» в главе 5 подробно описано, как наличие явных схем для микросервисных интерфейсов поможет выявить структурные нарушения и уменьшить необходимость в более сложных сквозных тестах.

Однако схемы не могут улавливать семантические сбои, а именно изменения в *поведении*, которые приводят к нарушениям из-за обратной несовместимости. Сквозные тесты, безусловно, помогут выявить эти семантические сбои, но за это приходится платить. В идеале хотелось бы иметь какой-то тип теста, способный улавливать семантические изменения и выполняться с уменьшенным охватом, улучшая изоляцию теста (и, следовательно, скорость обратной связи). Именно здесь на помощь приходят контрактные тесты и ориентированные на потребителя контракты.

Контрактное тестирование

При использовании *контрактных тестов* команда, чей микросервис использует внешний сервис, готовит тесты с описанием своих ожиданий поведения внешнего сервиса. Одна из основных причин, по которой подобные тесты могут быть полезны, заключается в том, что их можно запускать с любыми используемыми заглушками или макетами, представляющими внешние сервисы, — ваши контрактные тесты должны успешно проходить при запуске с вашими собственными заглушками точно так же, как и при работе с реальным внешним сервисом.

Контрактные тесты становятся очень полезными при использовании в рамках *контрактов, ориентированных на потребителя* (consumer-driven contracts,

CDC). Контрактные тесты, по сути, это явное программное представление того, какого поведения потребитель (вышестоящий микросервис) ожидает от производителя (нижестоящего). С помощью CDC команда сервиса-потребителя гарантирует, что эти контрактные тесты будут переданы команде сервисов-производителей, чтобы последние убедились, что их микросервис соответствует определенным ожиданиям. Как правило, для этого команда производителя запускает потребительские контракты для всех потребляющих микросервисов как часть своего набора тестов, который будет выполняться при каждой сборке. Очень важно с точки зрения обратной связи, что эти тесты необходимо запускать только с одним производителем изолированно, чтобы они могли быть быстрее и надежнее сквозных тестов и смогли заменить их собой.

В качестве примера вернемся к нашему предыдущему сценарию. У микросервиса `Покупатель` два отдельных потребителя: служба поддержки и интернет-магазин. Оба этих приложения-потребителя запрограммированы на определенный ответ на действия микросервиса `Покупатель`. В этом примере создается набор тестов для каждого потребителя: один представляет ожидания службы поддержки, а другой набор — интернет-магазина. Нам нужно только запустить сам микросервис `Покупатель`, ведь у нас тот же эффективный охват тестирования, что и у сервисных тестов. Контракты получили бы схожие характеристики производительности и потребовали бы от нас запуска только самого микросервиса `Покупатель` с отключением любых внешних зависимостей.

Хорошей практикой здесь будет привлечение кого-то из команд производителей и потребителей к совместной работе над созданием тестов, поэтому, возможно, люди из интернет-магазина и службы поддержки объединяются с людьми из отдела обслуживания клиентов. Можно утверждать, что CDC способствуют установлению четких линий связи и сотрудничества, где это необходимо, между микросервисами и командами-потребителями. Вообще внедрение CDC просто делает более явной существующую коммуникацию между командами. В межкомандном сотрудничестве CDC представляют собой явное напоминание о законе Конвея.

CDC находятся на том же уровне в пирамиде тестов, что и сервисные тесты, хотя и с совершенно иной направленностью, как показано на рис. 9.10. Эти тесты сосредоточены на том, как потребитель будет использовать сервис, и триггер при их поломке отличается от триггера сервисных тестов. Если один из этих CDC ломается во время сборки сервиса `Покупатель`, становится очевидным, на какого потребителя это повлияет. На данном этапе можно либо устранить проблему, либо начать обсуждение введения критического изменения, как мы обсуждали в разделе «Обработка изменений между микросервисами» главы 5.

Таким образом, с помощью CDC можно идентифицировать критические изменения до выпуска ПО, не прибегая к потенциально дорогостоящему сквозному тестированию.

Рис. 9.10. Интеграция ориентированных на потребителя тестов в пирамиду тестирования

Pact

Pact (https://pact.io) — это ориентированный на потребителя тестовый инструмент, который изначально был разработан внутри компании realestate.com.au, но теперь распространяется как продукт с открытым исходным кодом. Первоначально предназначенный только для Ruby и ориентированный на протоколы HTTP, теперь Pact поддерживает несколько языков и платформ, таких как JVM, JavaScript, Python и .NET, а также может использоваться для взаимодействия с сообщениями.

Применяя Pact, вы начинаете с определения ожиданий производителя, используя DSL на одном из поддерживаемых языков. Затем запускаете локальный сервер Pact и инициируете это ожидание с ним, чтобы создать файл спецификации Pact. Файл Pact — это просто формальная спецификация JSON. Очевидно, вы могли бы написать ее вручную, но использовать SDK для конкретного языка намного проще.

Очень приятным свойством текущей модели стало то, что локально запущенный макет сервера, используемый для генерации файла Pact, также работает как локальная заглушка для нижестоящих микросервисов. Определяя свои ожидания локально, вы решаете, как должен реагировать данный локальный сервис-заглушка. Это может заменить необходимость в таких инструментах, как mountebank (или ваших собственных решениях для заглушек или макетов).

На стороне производителя затем проверяется соответствие этой спецификации потребителя с помощью спецификации JSON Pact для управления вызовами вашего микросервиса и проверки ответов. Чтобы это сработало, производителю необходим доступ к файлу Pact. Как обсуждалось ранее в разделе «Сопоставление исходного кода и сборок с микросервисами» главы 7, мы ожидаем, что

потребитель и производитель находятся в разных сборках. Поэтому нам нужен какой-то способ дать производителю доступ к файлу JSON, сгенерированному для сборки потребителя.

Можно поместить файл Pact в репозиторий артефактов вашего инструмента CI/CD или же использовать Pact Broker (https://oreil.ly/kHTkY), позволяющий хранить несколько версий спецификаций Pact. Такое решение позволит вам запускать свои ориентированные на потребителя контрактные тесты с несколькими различными версиями потребителей, если вы хотите протестировать, скажем, эксплуатационную и совсем недавно созданную версию потребителя.

Pact Broker на самом деле обладает множеством полезных возможностей. Помимо того что он служит местом хранения контрактов, он также дает информацию, когда контракты были проверены. Кроме того, поскольку Pact Broker знает об отношениях между потребителем и производителем, он покажет вам зависимости между микросервисами.

Другие варианты

Pact — не единственный способ обойти CDC. Например, есть Spring Cloud Contract (https://oreil.ly/fjufx). Однако стоит отметить, что, в отличие от Pact, который с самого начала был разработан для поддержки различных технологических стеков, Spring Cloud Contract действительно полезен только в чистой экосистеме JVM.

Дело в разговорах

В Agile пользовательские истории часто служат в качестве фундамента для обсуждений. Контракты CDC — то же самое. Они упорядочивают набор дискуссий о том, как должен выглядеть API сервиса, и когда контракты нарушаются, они становятся отправной точкой для обсуждений дальнейшего развития этого API.

Важно понимать, что CDC требуют хорошей коммуникации и доверия между сервисом-потребителем и сервисом-производителем. Если обе стороны управляются одной командой (или одним человеком!), это не должно вызвать проблем. Однако если вы пользуетесь взаимодействий, предоставляемыми третьей стороной, у вас может не хватить частоты коммуникации или степени доверия, чтобы заставить CDC работать. В таких ситуациях вам, возможно, придется довольствоваться ограниченными интеграционными тестами с более широким охватом только вокруг *ненадежного* компонента. В качестве альтернативы, если вы создаете API для тысяч потенциальных пользователей, например, с помощью публичного API веб-сервиса, вам, возможно, придется самому играть роль потребителя (или работать с подмножеством ваших потребителей) при определении этих тестов. Отключение огромного количества внешних потребителей — довольно плохая идея, так что важность CDC возрастает!

Заключительное слово

Как было подробно описано ранее, у сквозных тестов большое количество недостатков, которое значительно возрастает по мере добавления тестируемых подвижных элементов. Из разговоров с людьми, которые уже некоторое время внедряют микросервисы в больших масштабах, я узнал, что большинство из них со временем полностью устраняют необходимость в сквозных тестах в пользу других механизмов проверки качества своего ПО. Например, применяются явные схемы и CDC, тестирование в эксплуатации или, возможно, некоторые из поэтапных методов доставки, вроде канареечных релизов.

Вы можете рассматривать выполнение сквозных тестов перед релизом в качестве тренажера. Пока вы изучаете, как работают контракты CDC, и совершенствуете методы мониторинга и развертывания, эти сквозные тесты могут стать полезной страховочной сеткой, позволяющей снизить риск в обмен на более высокие затраты времени. Но по мере совершенствования в других областях тестирования и увеличения относительной стоимости создания сквозных тестов вы можете начать уменьшать свою зависимость от сквозных тестов вплоть до полного отказа от них. Легкомысленный и поспешный отказ от них, вероятно, плохая идея.

Очевидно, что вы лучше меня понимаете профиль рисков вашей собственной организации, но я бы посоветовал вам долго и упорно думать, сколько сквозного тестирования действительно нужно провести.

Опыт разработчика

Одна из серьезных проблем, возникающих по мере того, как разработчики сталкиваются с необходимостью работать со все большим количеством микросервисов, заключается в том, что запуск возросшего количества микросервисов локально негативно влияет на опыт разработчиков. Это часто проявляется в ситуациях, когда программисту требуется запустить тест с большим охватом, объединяющий несколько микросервисов без сервисов-заглушек.

Насколько быстро это станет проблемой, будет зависеть от ряда факторов: сколько микросервисов необходимо запускать локально, в каком технологическом стеке написаны эти микросервисы, мощность локального компьютера — все это может сыграть свою роль. Некоторые технологические стеки более ресурсоемки с точки зрения их первоначального объема (например, микросервисы на основе JVM). В то время, как другие стеки могут привести к тому, что микросервисы будут быстрее и менее требовательны к ресурсам, что, возможно, позволит запускать гораздо больше микросервисов локально.

Одним из решений данной проблемы будет перенос разработки и тестирования в облачную среду. У вас может быть гораздо больше ресурсов, доступных для запуска необходимых микросервисов. Такая модель требует наличия

постоянного подключения к облачным ресурсам, и, помимо этого, могут пострадать циклы обратной связи. Если требуется внести локальное изменение в код и загрузить новую версию (или локально созданный артефакт) в облако, это приведет к значительным задержкам в циклах разработки и тестирования, особенно если вы работаете в части мира с нестабильным доступом к Интернету.

Полноценная разработка в облаке — это одно из возможных решений проблемы циклов обратной связи. Облачные IDE, такие как Cloud9, на данный момент принадлежащие AWS, показали, что это реально. Однако это больше похоже на технологии будущего, которые сегодня не так распространены.

По сути, я действительно считаю, что использование облачных сред для разработки и тестирования микросервисов приводит к большей сложности, чем необходимо, в дополнение к более высоким затратам. В идеале стоит стремиться к тому, чтобы разработчик запускал только те микросервисы, над которыми он действительно работает. Если инженер состоит в команде, владеющей пятью микросервисами, то у него должна быть возможность запускать эти микросервисы максимально эффективно, и для получения быстрой обратной связи предпочтительно, чтобы они запускались локально.

Но что, если пять микросервисов, которыми владеет ваша команда, захотят обратиться к другим системам и микросервисам, принадлежащим другим командам? Без них локальная среда разработки и тестирования не будет функционировать, не так ли? И здесь на помощь снова приходят заглушки. Я должен быть в состоянии поддерживать локальные заглушки, имитирующие микросервисы, выходящие за рамки моей команды. Единственные реальные микросервисы, которые вы должны запускать локально, — те, над которыми вы работаете. Если вы трудитесь в организации, где от вас ожидают работы с сотнями различных микросервисов, что ж, тогда вам придется иметь дело с гораздо более серьезными проблемами — эту тему мы рассмотрим в разделе «Сильное или коллективное владение» главы 15.

От предварительного тестирования к эксплуатационному

Исторически сложилось так, что основное внимание при тестировании уделялось тестированию систем *до* релиза продукта. Тесты позволяют определить серию моделей, с помощью которых необходимо получить доказательства, работает ли наша система так, как нам хотелось бы, как функционально, так и нефункционально. Но если модели несовершенны, мы столкнемся с проблемами, когда использование системы будет вызывать у клиентов злость. Ошибки проскальзывают в рабочую версию, обнаруживаются новые режимы сбоев, и потребители используют систему, как никто не мог предположить.

Одной из реакций на это часто становится определение все большего количества тестов и уточнение моделей, чтобы выявить проблемы на ранней стадии и уменьшить их количество в уже выпущенной системе. Однако в определенный момент придется признать, что при таком подходе снижается отдача. При тестировании перед развертыванием невозможно свести вероятность сбоя к нулю.

Сложность распределенной системы такова, что может оказаться невозможным выявить все потенциальные проблемы, прежде чем мы запустим систему в эксплуатацию.

Проще говоря, цель теста — подтвердить, что наше ПО достаточно качественное и отвечает завяленным требованиям. В идеале требуется как можно скорее получить обратную связь и возможность определить, есть ли проблема с нашим продуктом, прежде чем конечный пользователь столкнется с ней. Вот почему перед выпуском программного обеспечения проводится большое количество тестов.

Однако, ограничиваясь только предварительным тестированием, мы сокращаем количество мест, в которых можно обнаружить проблемы, а также исключаем возможность тестирования качества нашего ПО в самом важном месте — там, где оно будет использоваться.

Мы можем и должны также стремиться применять тестирование в эксплуатационной среде. Это можно реализовать безопасным способом. Такое тестирование обеспечит более качественную обратную связь, чем предварительное.

Виды эксплуатационных тестов

Существует длинный список различных тестов, которые можно провести на стороне потребителя, начиная от простых и заканчивая сложными. Для начала давайте подумаем о простой проверке соединения (ping), чтобы убедиться, что микросервис запущен. Такая проверка — это уже своего рода тест. Мы просто не рассматриваем ее в качестве теста, поскольку это действие, которым обычно занимаются операторы.

Дымовое тестирование — еще один подходящий пример. Обычно проводимый в рамках действий по развертыванию, дымовой тест гарантирует, что развернутое ПО работает правильно. Такие тесты обычно проводятся на реальном работающем программном обеспечении, прежде чем оно будет доступно для пользователей (подробнее об этом — в ближайшее время).

Канареечные релизы, которые мы рассмотрели в главе 8, также можно рассматривать как механизм, который связан с тестированием. Мы выпускаем новую версию ПО для небольшой части пользователей, чтобы «протестировать», правильно ли оно работает. Если релиз пройдет успешно, можно открыть доступ к продукту большему числу клиентов, возможно, полностью автоматизированным способом.

Другой пример эксплуатационного тестирования представлен внедрением фейкового поведения пользователя в систему, чтобы убедиться, что она работает должным образом, например размещение заказа для псевдоклиента или регистрация нового (искусственного) пользователя в реальной системе. Этот тип тестирования иногда может быть отвергнут из-за беспокойства о его влиянии на выпущенную систему. Поэтому, если вы создаете подобные тесты, убедитесь, что они безопасны.

Обеспечение безопасности тестирования в эксплуатации

Если вы решите проводить тестирование в эксплуатации (а вы должны это сделать!), важно, чтобы тесты не вызывали сбоев в работающей системе, будь то из-за нестабильности системы или искажения данных. Такая простая операция, как проверка соединения с экземпляром микросервиса, скорее всего, будет безопасной. Но если это вызывает нестабильность системы, у вас, вероятно, есть довольно серьезные проблемы, требующие решения, если только вы случайно не подвергли свою систему DoS-атаке.

Дымовые тесты, как правило, безопасны, поскольку их операции часто выполняются до релиза. В подразделе «Отделение развертывания от релиза» главы 8 рассматривалось, что разделение концепции развертывания от релиза может быть невероятно полезным. Когда дело доходит до тестирования в пользовательской среде, тесты, проводимые на развертываемом в эксплуатацию ПО до его выпуска, должны быть безопасными.

Люди, как правило, больше всего беспокоятся о безопасности таких вещей, как внедрение в систему фейкового поведения пользователя. На самом деле нам не требуется, чтобы заказ был отправлен или оплата произведена. Этот момент требует должной заботы и внимания, и, несмотря на трудности, данный тип тестирования может быть чрезвычайно полезным. Мы вернемся к этому в подразделе «Семантический мониторинг» главы 10.

Среднее время восстановления превышает среднее время между отказами?

Итак, рассматривая методы сине-зеленого развертывания или канареечный релиз, мы находим способ тестирования ближе к (или даже в) пользовательской среде, а также создаем инструменты, помогающие справиться со сбоем, если он произойдет. Использование этих подходов стало молчаливым признанием невозможности обнаружить и устранить все проблемы до фактического выпуска ПО.

Иногда затраты ресурсов на более качественное устранение возникающих проблем могут быть значительно полезнее, чем добавление большего числа автоматизированных функциональных тестов. В мире веб-операций это часто

называют компромиссом между оптимизацией *среднего времени наработки на отказ* (mean time between failures, MTBF) и *среднего времени восстановления* (mean time to repair, MTTR).

Методы сокращения времени восстановления могут быть такими же простыми, как мгновенный откат в сочетании с хорошим мониторингом (который мы обсудим в главе 10). Если мы способны обнаружить проблему в эксплуатации и откатить ее на ранней стадии, это уменьшит негативное влияние на наших клиентов.

Для разных организаций соотношение между MTBF и MTTR будет разным, и во многом это зависит от понимания истинных последствий сбоев. Однако большинство компаний, затрачивающих время на создание наборов функциональных тестов, часто практически не прилагают усилий для улучшения мониторинга и восстановления после сбоя. Таким образом, хотя они могут сократить количество дефектов, возникающих в первую очередь, такие организации не устранят их все и не готовы иметь дело с дефектами в эксплуатации.

Существуют и другие компромиссы. Например, вместо того, чтобы пытаться выяснить, действительно ли кто-то будет использовать ваше ПО, может быть гораздо разумнее сделать что-то сейчас, чтобы доказать идею или бизнес-модель, прежде чем создавать надежное программное обеспечение. В среде, где это имеет место, тестирование может оказаться излишним, поскольку влияние незнания того, работает ли ваша идея, гораздо выше, чем наличие дефекта в пользовательской среде. В таких ситуациях может быть вполне разумно вообще избегать тестирования до релиза продукта.

Кросс-функциональное тестирование

Основная часть данной главы была посвящена тестированию конкретных функциональных возможностей и их отличиям при тестировании систем на основе микросервисов. Однако есть еще одна категория тестирования. *Нефункциональные требования* — это обобщающий термин, используемый для описания тех характеристик, которыми обладает ваша система, но они не могут быть просто реализованы как обычная функция. Они включают в себя такие аспекты, как допустимая задержка веб-страницы, количество поддерживаемых страницей пользователей, доступность UI для людей с ограниченными возможностями или безопасность данных клиентов.

Термин «нефункциональный» никогда мне не нравился. Некоторые вопросы, охватываемые этим термином, кажутся очень функциональными по своей природе! Моя бывшая коллега, Сара Тарапоревалла, придумала вместо этого выражение «кросс-функциональные требования» (cross-functional requirements, CFR), которое мне по душе. Это больше говорит о том, что такое поведение системы на самом деле возникает только в результате большой сквозной работы.

Многие, если не большинство, требований CFR действительно могут встречаться только после выпуска в эксплуатацию. Тем не менее мы можем определить стратегии тестирования, помогающие по крайней мере увидеть, движемся ли мы к достижению этих целей. Такого рода тесты относятся к квадранту «Тестирование свойств». Отличным примером будет тест производительности. Его мы вскоре обсудим более подробно.

Возможно, вам захочется отслеживать некоторые CFR на уровне отдельного микросервиса. Например, вы можете решить, что срок службы платежного сервиса значительно выше требуемого, но вас устраивает большое время простоя сервиса рекомендаций музыки, поскольку ваш основной бизнес может выжить без рекомендаций исполнителей, похожих на Metallica, в течение десяти или более минут. Эти компромиссы в конечном счете окажут большое влияние на проектирование и развитие системы, и опять же детализированный характер микросервисной системы дает вам гораздо больше шансов найти эти компромиссы. При рассмотрении требований CFR, за которыеможет отвечать данный микросервис или команда, они обычно оказываются частью целей команды на уровне услуг (service-level objective, SLO). Эту тему мы изучим далее, в подразделе «Все ли у нас в порядке?» главы 10.

Тесты, связанные с CFR, также должны следовать модели пирамиды. Некоторые тесты должны быть сквозными, например нагрузочные, для остальных же это не обязательно. Например, как только вы обнаружили узкое место в производительности при сквозном нагрузочном тестировании, напишите тест с меньшим охватом, который поможет выявить проблему в будущем. Другие CFR довольно легко подходят для более быстрых тестов. Я помню, как работал над проектом, где наша HTML-разметка использовала надлежащую функциональность, чтобы помочь людям с ограниченными возможностями пользоваться веб-сайтом.

Проверка сгенерированной разметки на наличие соответствующих элементов управления может быть выполнена очень быстро, не требуя каких-либо сетевых обходов.

Довольно часто разработчики слишком поздно задумываются о CFR. Я настоятельно рекомендую изучить ваши CFR как можно раньше и регулярно пересматривать их.

Тесты производительности

Тесты производительности стоит проводить в явном виде, чтобы убедиться, что некоторые из наших кросс-функциональных требований могут быть выполнены. При декомпозиции систем на более мелкие микросервисы мы увеличиваем количество вызовов, выполняемых через границы сети. Если раньше операция включала в себя один вызов базы данных, то теперь — три или четыре вызова через границы сети к другим сервисам с соответствующим количеством вызовов базы данных. Все это снижает скорость работы систем.

Особенно важно отслеживать источники задержек. Если какая-либо часть цепочки вызовов из нескольких синхронных вызовов начинает действовать медленно, это влияет на все и потенциально может привести к значительным последствиям. Это делает наличие какого-либо способа тестирования производительности приложений еще важнее, чем это могло бы быть в случае более монолитной системы. Часто причина, по которой такого рода тестирование затягивается, заключается в том, что изначально для тестирования недостаточно одной лишь системы. Я понимаю эту проблему, но нередко это приводит к тому, что все идет наперекосяк, а тестирование производительности часто проводится только непосредственно перед первым запуском, если вообще проводится! Не попадайтесь в эту ловушку.

Как и в случае с функциональными тестами, вам может понадобиться сочетание подходов. Вы можете выполнить тесты производительности, изолирующие отдельные сервисы, но стоит начать с тестов, проверяющих основные маршруты запросов в вашей системе. Вероятно, у вас получится взять сквозные тесты перемещений и просто запустить их на полную мощность.

Чтобы получить стоящие результаты, обычно приходится запускать определенные сценарии с постепенно увеличивающимся числом имитируемых клиентов. Такая модель тестирования дает возможность увидеть, как изменяется задержка вызовов с увеличением нагрузки. Это значит, что тесты производительности могут занять некоторое время. Кроме того, потребуется, чтобы система максимально соответствовала эксплуатационной версии. Такой подход гарантирует, что полученные результаты отображают производительность, которую вы можете ожидать от реально работающей системы. Скорее всего, вам потребуется получить больший объем данных, аналогичных пользовательской среде, и, возможно, понадобится больше компьютеров для соответствия инфраструктуре. Подобные задачи достаточно сложны. Даже если вам трудно сделать среду тестирования производительности похожей на эксплуатационную, тесты все равно могут выявить узкие места. Просто имейте в виду, что вы рискуете получить ложноотрицательные результаты или, что еще хуже, ложноположительные.

Из-за времени, необходимого для выполнения тестов производительности, не всегда есть возможность запускать их при каждой фиксации кода. Обычной практикой стало выполнение подгруппы тестов каждый день и большего набора каждую неделю. Какой бы подход вы ни выбрали, убедитесь, что проводите тесты регулярно. Чем дольше вы обходитесь без тестов производительности, тем сложнее отследить виновника возникновения сбоев. Проблемы производительности особенно трудно устранить, поэтому, если вам удастся сократить количество коммитов, требующих просмотра для изучения новых неполадок, вы облегчите себе жизнь.

И обязательно посмотрите на результаты! Я был очень удивлен количеством встречавшихся мне команд, которые проделали внушительную работу по

внедрению и запуску тестов, но фактически никогда не проверяли цифры. Обычно это происходит потому, что люди не знают, как выглядит «хороший» результат. У вас действительно должны быть цели. При предоставлении микросервиса для использования в рамках более широкой архитектуры обычно есть конкретные ожидания, и вы обязуетесь их выполнить — SLO, о которых я упоминал ранее. Если в рамках этого соглашения вы обеспечиваете определенный уровень производительности, тогда имеет смысл, чтобы любой автоматический тест давал вам обратную связь о том, достигнете ли вы (и, надеюсь, превзойдете) этой цели.

Вместо конкретных целевых показателей производительности автоматизированные тесты по-прежнему могут быть очень полезны, помогая вам увидеть, как меняется эффективность микросервиса с каждой модификацией. Это может стать страховкой, если ваши правки приводят к резкому снижению производительности. Таким образом, альтернативой конкретной цели может быть провал теста, если разница в производительности от одной сборки к другой слишком сильно варьируется.

Тестирование производительности должно проводиться совместно с пониманием реальной эффективности системы (о чем мы подробнее поговорим в главе 10), и в идеале для визуализации поведения системы в среде тестирования производительности следует использовать те же инструменты, что и в эксплуатационной среде. Такой подход может значительно упростить сравнение подобного с подобным.

Тесты надежности

Архитектура микросервиса часто настолько надежна, насколько надежно ее самое слабое звено. Поэтому в микросервисы обычно встраивают механизмы, позволяющие повысить их надежность, а значит, и надежность всей системы. Мы подробнее рассмотрим эту тему в разделе «Шаблоны стабильности» главы 12, но в качестве примера можно привести запуск нескольких экземпляров микросервиса за балансировщиком нагрузки, чтобы сбой экземпляра был допустимым явлением, или использование автоматических выключателей для программной обработки ситуаций, в которых невозможно связаться с нижестоящими микросервисами.

В таких ситуациях может быть полезно применять тесты, позволяющие воссоздавать определенные сбои, чтобы убедиться, что ваш микросервис продолжает работать как единое целое. По своей природе эти тесты могут быть сложнее в реализации. Например, если вы захотите создать искусственный сетевой тайм-аут между тестируемым микросервисом и внешней заглушкой. Тем не менее они полезны, особенно при создании общей функциональности, которая будет использоваться в нескольких микросервисах, например при реализации сервисной сети по умолчанию для обработки прерывания цепи.

Резюме

Подытожим: в данной главе я изложил целостный подход к тестированию, который, надеюсь, даст вам некоторые общие рекомендации, как действовать при тестировании собственных систем. Давайте повторим основы.

- Для быстрой обратной связи оптимизируйте и соответствующим образом разделите типы тестов.
- Избегайте необходимости в сквозных тестах, охватывающих более одной команды, — вместо этого попробуйте использовать CDC.
- Используйте CDC, чтобы создать отправные точки для дискуссий между командами.
- Постарайтесь найти компромисс между тем, чтобы прилагать больше усилий для тестирования, и тем, чтобы быстрее обнаруживать проблемы в процессе производства (баланс оптимизаций MTBF и MTTR).
- Дайте шанс тестированию в эксплуатации!

Если вам интересно узнать больше о тестировании, я рекомендую книгу «Agile-тестирование» Лизы Криспин и Джанет Грегори (Addison-Wesley), в которой, помимо прочего, подробнее рассматривается квадрант тестирования. Для более глубокого погружения в тестовую пирамиду, наряду с некоторыми примерами кода и ссылками на инструменты, я также рекомендую книгу «Практическая тестовая пирамида» Хэма Воке[1].

Данная глава была посвящена в основном проверке работоспособности нашего кода до его попадания в рабочую среду. Но мы также начали рассматривать тестирование приложения после его запуска в эксплуатацию. Это то, что необходимо изучить более подробно. Оказывается, микросервисы создают множество проблем для понимания того, как ПО ведет себя у конечного пользователя, — тема, которую мы рассмотрим подробнее далее.

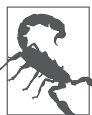
[1] *Vocke H.* The Practical Test Pyramid // martinfowler.com. 26 февраля 2018 года. https://oreil.ly/J7lc6.

ГЛАВА 10

От мониторинга к наблюдаемости

Я надеюсь, что разбиение нашей системы на более мелкие, детализированные микросервисы даст множество преимуществ. Однако это добавляет и значительные сложности. Когда речь идет о понимании поведения наших систем в пользовательской среде, эта возросшая сложность становится более очевидной. Вы очень быстро заметите, что инструменты и методы, хорошо сработавшие для относительно простых однопроцессных монолитных приложений, не работают так хорошо для микросервисной архитектуры.

В текущей главе мы рассмотрим проблемы, связанные с мониторингом микросервисной архитектуры. Я покажу, что хотя новые инструменты и могут помочь, вам потребуется полностью изменить свое мышление, особенно когда речь заходит о выяснении того, что за неразбериха происходит в эксплуатации. Мы также поговорим о возросшем внимании к концепции наблюдаемости — пониманию того, как получить возможность задавать вопросы нашей системе, чтобы узнать, что происходит не так.



Проблемы эксплуатации

Вы не сможете оценить истинный масштаб проблем от применения микросервисной архитектуры, пока не запустите систему в работу, и она не начнет обслуживать реальный трафик.

Сбой, паника и замешательство

Представьте себе картину: тихий вечер пятницы и команда с нетерпением ждет возможности пораньше ускользнуть в паб, чтобы начать выходные вдали от работы. И внезапно приходят электронные письма. Веб-сайт ведет себя неправильно! «Твиттер» пылает от сообщений об ошибках вашей компании, ваш босс нервничает, а перспективы спокойного уик-энда тают на глазах.

Мало что так хорошо подводит итог этой проблеме, как следующий твит.

> Мы заменили наш монолит на микросервисы, чтобы каждое отключение могло быть больше похоже на загадочное убийство[1].

Отслеживание происшествия и его причин — наша первая остановка. Но это непросто, если у нас длинный список подозреваемых.

В мире монолитных приложений с одним процессом у нас по крайней мере есть очень очевидное место, с которого можно начать расследование. Сайт медленный? Это монолит. Сайт выдает странные ошибки? Монолит. Процессор нагружен на 100 % — монолит. Запах гари? Ну, вы поняли идею[2]. Наличие единой точки отказа делает расследование сбоев намного проще!

Теперь подумаем о нашей микросервисной системе. Возможности, которые мы предлагаем нашим пользователям, обслуживаются из нескольких микросервисов, некоторые, в свою очередь, взаимодействуют с еще большим количеством микросервисов. У такого подхода есть много преимуществ (и это хорошо, в противном случае данная книга была бы пустой тратой времени), но в мире мониторинга мы сталкиваемся с проблемой посложнее.

Вот у нас есть несколько серверов для мониторинга, несколько файлов логов для просмотра и несколько мест, где задержка в сети может вызвать проблемы. Площадь поражения увеличилась, а вместе с ней и круг «подозреваемых». Итак, как нам быть? Стоит разобраться в том, что в противном случае могло бы оказаться полным беспорядком — последнее, с чем любой из нас хотел бы иметь дело в пятницу вечером (да когда угодно!).

Для начала нам нужно мониторить мелочи и обеспечивать агрегирование, чтобы увидеть общую картину. Затем необходимо убедиться, что у нас есть доступные инструменты, требующиеся в рамках нашего расследования. Наконец, стоит разумнее подходить к тому, как мы размышляем о работоспособности системы, используя такие концепции, как эксплуатационное тестирование. В текущей главе мы обсудим каждое из этих необходимых условий. Давайте начнем.

Один микросервис — один сервер

На рис. 10.1 представлен очень простой вариант установки: один хост, на котором запущен один экземпляр микросервиса. Теперь нам нужно следить за ним, чтобы знать, когда что-то пойдет не так. Это даст нам возможность исправить проблему. Так на что же следует обратить внимание?

[1] Honestly Black Lives Matter (@honest_update), October 7, 2015, 7:10 p.m., https://oreil.ly/Z28BA.

[2] Пожар как причина отключения системы — не совсем надуманная идея. Однажды я помогал в ликвидации последствий производственного сбоя, вызванного возгоранием сети хранения данных (SAN). То, что нам потребовалось несколько дней, чтобы сообщить о пожаре, — это уже другая история.

Сначала надо получить информацию от самого хоста. Процессор, память — все это пригодится. Далее понадобится доступ к логам из самого экземпляра микросервиса. Если пользователь сообщает об ошибке, мы должны увидеть ее в логах, что прольет свет на произошедшее. На данный момент, с единственным хостом, мы, вероятно, можем обойтись простым локальным входом в систему и использованием инструментов командной строки для просмотра логов.

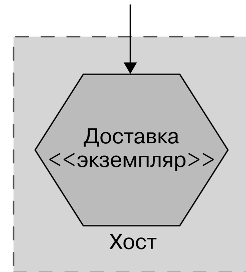


Рис. 10.1. Один экземпляр микросервиса на одном хосте

Наконец, мы могли бы наблюдать извне за самим приложением. Как минимум хорошей идеей будет мониторинг времени отклика микросервиса. Если у вас есть веб-сервер, подключенный к экземпляру микросервиса, возможно, стоило бы просто просмотреть логи веб-сервера. Или продвинуться немного дальше и использовать что-то вроде контрольной конечной точки проверки работоспособности, чтобы убедиться, что микросервис работает и «здоров» (мы рассмотрим, что это значит, позже).

Проходит время, нагрузки увеличиваются, и мы понимаем, что систему нужно масштабировать...

Один микросервис — несколько серверов

Теперь у нас есть несколько копий сервиса, запущенных на отдельных хостах, как показано на рис. 10.2, а запросы к разным экземплярам распределяются через балансировщик нагрузки. Теперь все немного усложняется. Нам по-прежнему надо отслеживать все те же вещи, что и раньше, но делать это таким образом, чтобы стало возможным изолировать проблему. Когда загрузка процессора высока, является ли это проблемой, которую мы наблюдаем на всех хостах, что указывает на неполадки с самим сервисом? Или он изолирован в пределах одного хоста, подразумевая, что проблема присутствует на самом хосте — возможно, неконтролируемый процесс ОС?

На данный момент все еще необходимо отслеживать показатели на уровне хоста и, возможно, даже предупреждать о превышении ими некоего порога. Но теперь стоит посмотреть, каковы эти показатели на всех хостах. Другими словами, мы хотим агрегировать их, сохранив при этом возможность детализации. Поэтому нам нужен инструмент, способный собирать все эти показатели со всех хостов и позволять нам разбирать их по крупицам.

В конце концов, у нас есть логи. Поскольку сервис работает на нескольких серверах, вероятно, будет очень утомительно заходить каждый хост для просмотра данных. Однако при наличии всего нескольких хостов можно использовать такие инструменты, как мультиплексоры SSH, позволяющие запускать одни

и те же команды на множестве хостов. С помощью большого монитора и запуска программы `grep "Error"` в логе микросервиса вполне реально найти виновника. Да, это не очень практично, но на какое-то время должно быть *вполне достаточно*. Однако подобный подход довольно быстро устареет.

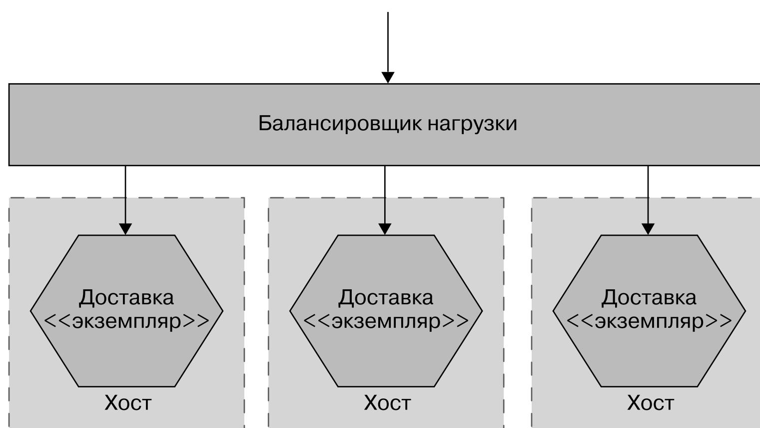


Рис. 10.2. Один сервис, распределенный по нескольким хостам

Для таких задач, как отслеживание времени отклика, можно фиксировать время отклика в балансировщике нагрузки для нисходящих вызовов микросервисов. Однако также следует учитывать, что произойдет, если балансировщик окажется узким местом в системе, — может потребоваться фиксировать время отклика как балансировщика нагрузки, так и самих микросервисов. На данный момент нас, вероятно, гораздо больше волнует, как выглядит исправный сервис, поскольку балансировщик нагрузки будет настраиваться для удаления неисправных узлов из нашего приложения. Надеюсь, к тому времени, как мы доберемся до этого места, у нас будет хотя бы некоторое представление, как выглядит здоровый сервис.

Несколько микросервисов — несколько серверов

На рис. 10.3 все становится гораздо интереснее. Несколько сервисов сотрудничают для предоставления возможностей нашим пользователям, и эти сервисы работают на нескольких хостах, будь то физические или виртуальные. Как вы найдете искомую ошибку в тысячах строк логов на нескольких хостах? Как определить, ведет ли себя сервер неправильно, или это системная проблема? И как отследить ошибку, обнаруженную глубоко в цепочке вызовов между несколькими хостами, и выяснить, что ее вызвало?

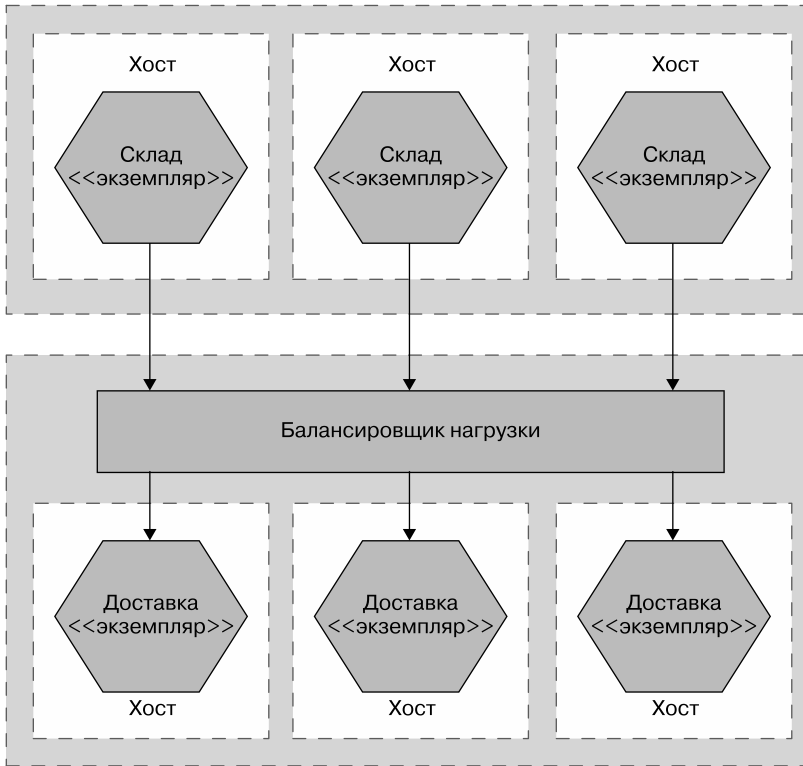


Рис. 10.3. Несколько сотрудничающих сервисов, распределенных по нескольким хостам

Агрегирование информации — метрик и логов — играет жизненно важную роль в обеспечении данного процесса. Но это не единственное, что нам нужно учитывать. Необходимо решить, как просеять этот огромный поток данных и попытаться во всем разобраться. Прежде всего речь идет о смене мышления: от довольно статичного мониторинга к более активному миру наблюдаемости и эксплуатационного тестирования.

Наблюдаемость и мониторинг

Мы собираемся углубиться в решение некоторых из только что изложенных проблем. Но прежде, чем мы это сделаем, я думаю, важно изучить термин, который приобрел большую популярность с тех пор, как я написал первое издание, — *наблюдаемость*.

Как это часто бывает, концепция наблюдаемости существует уже несколько десятилетий, но только недавно появилась в сфере разработки ПО. Наблюдаемость системы — это степень, в которой можно понять внутреннее состояние

системы по внешним выводам. Обычно это требует более целостного понимания вашего программного обеспечения — представления его скорее в виде *системы*, чем набора разрозненных сущностей.

На практике чем более наблюдаема система, тем легче оценить ситуацию, когда что-то идет не так. Наше понимание внешних результатов помогает быстрее выявить основную проблему. Парадокс заключается в том, что часто требуется создавать эти внешние результаты и использовать различные типы инструментов для понимания результатов.

С другой стороны, мониторинг — это то, чем мы занимаемся. *Следим* за системой. Все пойдет наперекосяк, если вы сосредоточитесь только на мониторинге — на самой деятельности, — не думая о том, чего следует ожидать от этой деятельности.

Более традиционные подходы к мониторингу заставили бы вас заранее подумать, что может пойти не так, и определить механизмы оповещения, чтобы сообщить вам, когда это произойдет. Но по мере того, как система становится все более распределенной, вы столкнетесь с проблемами, которые никогда бы вам в голову не пришли. При работе в системе с высокой наблюдаемостью у вас будет набор внешних выходных данных, которые можно опрашивать различными способами. Конкретный результат наличия наблюдаемой системы — вы можете задавать вопросы своей эксплуатируемой системе, которые вы никогда бы не подумали задать раньше.

Таким образом, можно рассматривать мониторинг как деятельность — нечто, что мы делаем, а наблюдаемость представляет собой свойство системы.

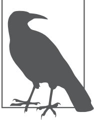
Столпы наблюдаемости? Не так быстро

Некоторые люди пытались свести идею наблюдаемости к нескольким основным концепциям. Часть из них сосредоточились на «трех столпах» наблюдаемости в виде метрик, ведения логов и распределенной трассировки. Компания New Relic даже ввела термин MELT (metrics, event, logs, traces), который на самом деле не прижился, но по крайней мере в New Relic пытались. Хотя эта простая модель поначалу мне очень понравилась (а я обожаю аббревиатуры!), со временем я отошел от этого мышления, считая его чрезмерно упрощенным, а также потенциально упускающим суть.

Во-первых, сведение свойства системы к деталям реализации таким образом кажется мне отстающим. Наблюдаемость — это свойство, и есть много способов, которыми можно было бы его достичь. Чрезмерное внимание к конкретным деталям реализации сопряжено с риском сосредоточения внимания на деятельности, а не на результате. Это аналогично нынешнему миру ИТ, где сотни, если не тысячи организаций влюбились в создание систем на основе микросервисов, на самом деле не понимая, чего они пытаются достичь!

Во-вторых, всегда ли существуют конкретные границы между этими понятиями? Я бы сказал, что многие из них пересекаются. Можно при желании

поместить метрики в файл логов. Аналогично можно построить распределенную трассировку из серии строк лога, что обычно и делается.



Наблюдаемость — это степень, в которой можно понять, что делает система, основываясь на внешних входных данных. Логи, события и метрики могут помочь вам сделать процессы наблюдаемыми, но обязательно сосредоточьтесь на том, чтобы сделать систему понятной, а не на использовании множества инструментов.

Цинично, но я мог бы предположить, что продвижение этого упрощенного описания — это способ продать вам инструменты. Для метрик нужен один инструмент, для логов — другой и еще один для трассировок! И вам необходимо отправлять всю эту информацию по-разному! Гораздо проще продавать функции для галочки, когда вы пытаетесь вывести продукт на рынок, чем говорить о результатах. Как я уже сказал, я *мог бы* предположить это с циничной точки зрения, но сейчас 2021 год, и я стараюсь мыслить более позитивно[1].

Можно утверждать, что все эти три (или четыре!) идеи на самом деле представляют собой лишь конкретные примеры более обобщенной концепции. По сути, мы вольны рассматривать любую часть информации, получаемой от нашей системы, в общем виде как событие. Определенное событие может содержать небольшой или внушительный объем информации: данные о частоте процессора, сведения о неудачном платеже, факт входа клиента в систему или множество иной информации. Мы способны спроецировать из этого потока событий трассировку (предполагая, что можно соотнести эти события), индекс с возможностью поиска или совокупность чисел. Хотя в настоящее время мы решили собирать эту информацию разными способами, используя различные инструменты и протоколы. Текущие цепочки инструментов не должны ограничивать наше мышление в плане того, как наилучшим образом получить необходимую информацию.

Когда речь заходит о том, чтобы сделать систему наблюдаемой, подумайте о выходных данных, которые вам требуется получить от системы, с точки зрения событий, доступных для сбора и опроса. Возможно, в данный момент вам придется использовать разные инструменты для отображения различных типов событий, но в будущем это может измениться.

Строительные блоки для наблюдаемости

Так что же нам нужно? Сделать пользователей нашего ПО счастливыми. Если есть проблема, мы должны знать о ней — в идеале до того, как потребители сами обнаружат ее. Когда проблема все-таки возникает, нам потребуется решить, что можно сделать, чтобы система снова заработала. И как только пыль осядет,

[1] Я не говорил, что справляюсь.

надо иметь под рукой достаточно информации, чтобы понять, что пошло не так и какие предупреждающие мероприятия провести.

В оставшейся части текущей главы мы рассмотрим, как все это осуществить. Мы познакомимся с рядом строительных блоков, способных помочь улучшить наблюдаемость архитектуры вашей системы.

Агрегация логов

Сбор информации от нескольких микросервисов — жизненно важный строительный блок любого решения для мониторинга или обеспечения наблюдаемости.

Агрегация метрики

Сбор необработанных данных из наших микросервисов и инфраструктуры для выявления проблем, планирования пропускной способности и, возможно, даже масштабирования наших приложений.

Распределенная трассировка

Отслеживание потока вызовов через границы нескольких микросервисов для определения того, что пошло не так, и получения точной информации о задержке.

У вас все в порядке?

Анализ бюджетов ошибок, SLA, SLO и т. д., чтобы понять, как их можно использовать для обеспечения того, чтобы микросервис удовлетворял потребности своих потребителей.

Оповещение

О чем следует оповещать? Как выглядит хорошее оповещение?

Семантический мониторинг

Нестандартно мыслить о работоспособности наших систем и о том, что должно разбудить нас среди ночи.

Тестирование в эксплуатации

Краткое описание различных техник.

Начнем, пожалуй, с самого простого, но многократно окупаемого — агрегации логов.

Агрегация логов

При наличии большого количества серверов и экземпляров микросервисов даже в скромной микросервисной архитектуре ведение логов на машинах или мультиплексирование SSH для их извлечения на самом деле не самое лучшее занятие. Вместо этого мы планируем использовать специализированные подсистемы для сбора логов и предоставления централизованного доступа к ним.

Логи очень быстро станут одним из самых важных механизмов, помогающих вам понять, что происходит в эксплуатируемой системе. В более простых архитектурах развертывания файлы логов, информация, помещаемая в них, и то, как мы с ними обращаемся, часто не учитываются. В условиях повышения степени распределенности системы они станут незаменимым инструментом, который не только поможет вам диагностировать, что пошло не так, но и сообщит вам, что возникла проблема, требующая первоочередного внимания.

Вскоре мы обсудим, что в этой области существует множество инструментов, но все они в основном работают одинаково, как показано на рис. 10.4. Процессы (например, экземпляры микросервисов) создают логи в своей локальной файловой системе. Локальный процесс-демон периодически собирает и пересылает данный лог в какое-то хранилище, к которому могут обращаться операторы. Одним из приятных аспектов таких систем стало то, что микросервисной архитектуре не обязательно знать о них. Вам не нужно изменять код, чтобы использовать какой-то специальный API, — вы просто создаете лог в локальной файловой системе. Однако вам необходимо понимать режимы сбоев в этом процессе доставки логов, особенно если вы хотите понять, в каких ситуациях логи могут потеряться.

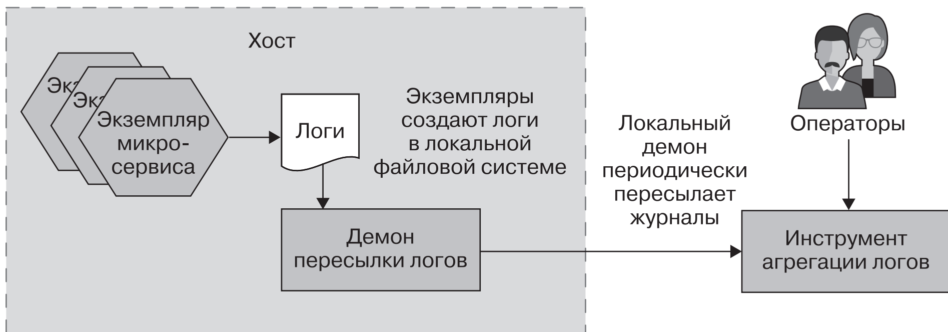


Рис. 10.4. Обзор того, как логи собираются в рамках агрегации логов

Теперь, я надеюсь, вы поняли, что я стараюсь избегать догматизма в отношении многих вещей. Вместо того чтобы просто сказать, что вы *должны* сделать X или Y, я попытался дать вам контекст и рекомендации, а также объяснить нюансы определенных решений. То есть я попытался дать вам инструменты для принятия правильного выбора в вашем контексте. Но что касается агрегации логов, я буду максимально близок к тому, чтобы дать универсальный совет: вам стоит рассматривать внедрение инструмента агрегации логов как *необходимое условие* для реализации микросервисной архитектуры.

У меня есть две причины для такой точки зрения. Во-первых, агрегация логов невероятно полезна. Для тех из вас, кто относится к своим лог-файлам

как к свалке дезинформации, это станет неожиданностью. Но, поверьте мне, при правильном выполнении агрегация логов может быть невероятно ценной, особенно при использовании с идентификаторами корреляции.

Во-вторых, реализовать агрегацию логов не так уж сложно по сравнению с другими источниками проблем, которые может принести микросервисная архитектура. Если ваша организация не в состоянии успешно реализовать простое решение для агрегирования логов, она, скорее всего, сочтет другие аспекты микросервисной архитектуры слишком сложными для обработки. Поэтому подумайте об использовании такого решения как о способе проверки готовности вашей организации к последующим трудностям.

Прежде всего остального

Прежде чем делать что-либо еще для построения микросервисной архитектуры, запустите инструмент агрегации логов. Считайте это необходимым условием для построения микросервисной архитектуры. Позже вы меня поблагодарите.

Также верно сказать, что у агрегации логов есть свои ограничения, и со временем вы захотите изучить более утонченные инструменты, чтобы дополнить или даже заменить некоторые из возможностей агрегации логов. Все это, как бы то ни было, по-прежнему остается тем, с чего стоит начать.

Общий формат

Если вы собираетесь агрегировать свои логи, вам потребуется выполнять запросы по ним для извлечения полезной информации. Чтобы это сработало, важно выбрать разумный стандартный формат логов, в противном случае ваши запросы будет трудно или даже невозможно записать. Необходимо, чтобы дата, время, имя микросервиса, уровень лога и т. д. находились в согласованных местах в каждом файле логов.

Некоторые агенты пересылки логов предоставляют возможность переформатировать логи перед их пересылкой в центральное хранилище. Лично я бы избегал этого везде, где возможно. Проблема в том, что переформатирование логов может быть трудоемким с точки зрения вычислений. Я встречал реальные проблемы в эксплуатируемых системах, вызванные привязкой процессора к выполнению данной задачи. Гораздо лучше приводить логи к тому виду, в каком они записываются самим микросервисом. Я бы использовал агенты пересылки для переформатирования логов в тех местах, где нет возможности изменить их исходный формат, — например, в устаревшем или стороннем ПО.

Мне казалось, что за время, прошедшее с тех пор, как я написал первое издание, общий отраслевой стандарт ведения логов наберет обороты, но, похоже, этого еще не произошло. По-видимому, существует множество вариаций. Обычно они включают в себя использование стандартного формата логов,

поддерживаемого веб-серверами, такими как Apache и nginx, и расширение его за счет добавления дополнительных столбцов данных. Однако в рамках вашей собственной микросервисной архитектуры вы сами выбираете формат, который стандартизируете внутри компании.

При использовании довольно простого формата лога будут выдаваться только примитивные строки текста, содержащие определенные фрагменты информации в определенных местах лога. В примере 10.1 показан образец формата.

Пример 10.1. Несколько примеров логов

```
15-02-2020 16:00:58 Order INFO [abc-123] Customer 2112 has placed order 988827
15-02-2020 16:01:01 Payment INFO [abc-123] Payment $20.99 for 988827 by cust 2112
```

Инструменту агрегации логов необходимо располагать информацией, как проанализировать строку, чтобы извлечь данные, которые мы, возможно, захотим запросить, — например, временную метку, имя микросервиса или уровень лога. В текущем примере это выполнимо, поскольку необходимые фрагменты находятся в статических местах логов: дата — это первый столбец, время — второй столбец и т. д. Однако поиск строк, связанных с определенным клиентом, будет более проблематичным: идентификатор клиента находится в обеих строках лога, но отображается в разных местах. Именно здесь можно было бы задуматься о написании более структурированных строк лога, возможно, с использованием формата JSON, чтобы появилась возможность искать, например, идентификатор клиента или заказа в одном и том же месте. Опять же необходимо будет сконфигурировать инструмент агрегации логов для анализа и извлечения необходимой информации. И еще кое-что: если вы ведете логи в формате JSON, то без дополнительных инструментов человеку будет затруднительно его считывать, а просмотр лога в обыкновенном текстовом редакторе может оказаться не очень удобным.

Согласование строк лога

При большом количестве взаимодействующих сервисов, предоставляющих любые заданные возможности конечному пользователю, один инициирующий вызов может в итоге сгенерировать несколько последующих вызовов сервисов. Давайте рассмотрим пример с MusicCorp, как показано на рис. 10.5. Мы регистрируем клиента в нашем новом стриминговом сервисе. Клиент выбирает желаемый стриминговый пакет и нажимает кнопку подтверждения. При ее нажатии в `Шлюз`, находящийся на периметре нашей системы, приходит сигнал. Это, в свою очередь, активирует вызов микросервиса `Стриминг`, который сообщается с сервисом `Оплата` для принятия первого платежа. Затем сервис `Покупатель` обновляет информацию о том, что стриминг для клиента включен, и с помощью микросервиса `Электронная почта` направляет ему письмо, подтверждающее подписку.

Что будет, если вызов микросервиса `Оплата` приведет к возникновению ошибки? Мы подробно поговорим об обработке сбоев в главе 12, а пока рассмотрим сложность диагностики того, что произошло.

Единственный микросервис, регистрирующий ошибку, — `Оплата`. Если повезет, мы выясним, какой запрос вызвал проблему, и, возможно, даже сможем посмотреть параметры вызова. Но у нас не получится увидеть эту ошибку в более широком контексте, в котором она возникает. В данном примере, даже если мы предположим, что каждое взаимодействие генерирует только одну строку лога, у нас будет пять строк лога с информацией об этом потоке вызовов. Возможность видеть эти строки, сгруппированные вместе, может быть невероятно полезной.

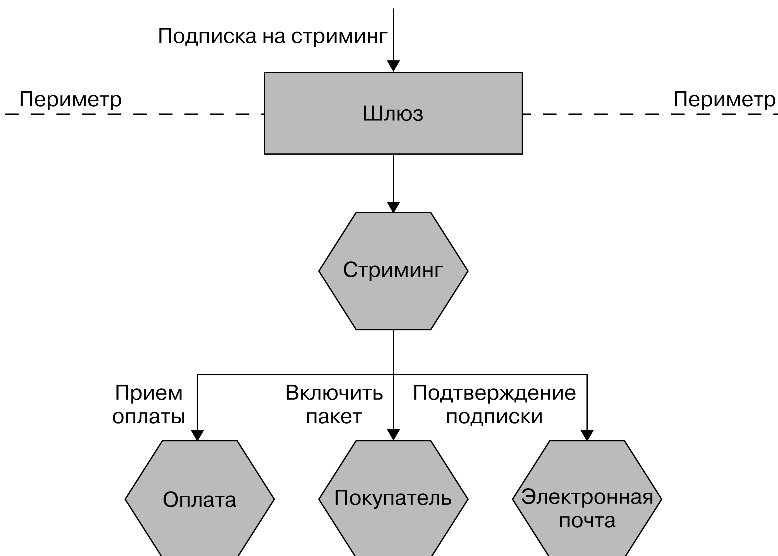


Рис. 10.5. Серия вызовов через несколько микросервисов, связанных с регистрацией покупателя

Один из подходов, который может быть здесь полезен, — это использование идентификаторов корреляции, о чем мы впервые упомянули в главе 6 при обсуждении саги. Когда выполняется первый вызов, вы генерируете уникальный идентификатор, используемый для сопоставления всех последующих вызовов, связанных с запросом. На рис. 10.6 мы генерируем этот идентификатор в сервисе `Шлюз`, и затем он передается в качестве параметра всем последующим вызовам.

Регистрация любой активности микросервиса, вызванной этим входящим вызовом, будет записываться вместе с идентификатором корреляции, размещенным в установленном месте в каждой строке лога, как показано в примере 10.2. В дальнейшем это позволит легко извлечь все логи, связанные с заданным идентификатором корреляции.

Пример 10.2. Использование идентификатора корреляции в фиксированном местоположении в строке лога

```
15-02-2020 16:01:01 Gateway INFO [abc-123] Signup for streaming
15-02-2020 16:01:02 Streaming INFO [abc-123] Cust 773 signs up ...
15-02-2020 16:01:03 Customer INFO [abc-123] Streaming package added ...
15-02-2020 16:01:03 Email INFO [abc-123] Send streaming welcome ...
15-02-2020 16:01:03 Payment ERROR [abc-123] ValidatePayment ...
```

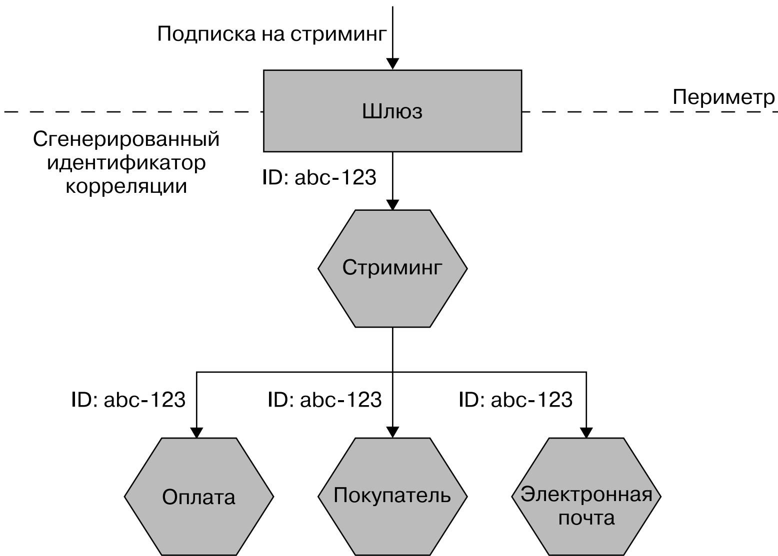
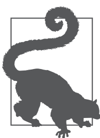


Рис. 10.6. Генерация идентификатора корреляции для набора вызовов

Вам, конечно, потребуется убедиться, что каждый сервис знает, что нужно передать идентификатор корреляции. Именно здесь необходимо применить стандартизацию и проявить больше решительности в обеспечении соблюдения этого правила во всей вашей системе. Но как только это будет сделано, вы сможете создавать инструменты для отслеживания всех видов взаимодействий. Такой инструментарий будет полезен при отслеживании серии сбоев или странных тупиковых ситуаций, или даже при выявлении особенно дорогостоящих транзакций, поскольку вы теперь в состоянии представить себе весь каскад вызовов.

Идентификаторы корреляции в логах поначалу кажутся не такими уж полезными, но, поверьте мне, со временем они могут оказаться *невероятно* полезными. К сожалению, дооснащение системы ими может оказаться непростой задачей. Именно по этой причине я строго рекомендую внедрять идентификатор корреляции в файлы логов как можно раньше. Логи, конечно, могут помочь вам только в этом отношении, ведь некоторые проблемы лучше решаются с помощью

распределенных инструментов отслеживания, которые мы вскоре рассмотрим. Но простой идентификатор корреляции в лог-файле будет невероятно полезен на начальном этапе. Его применение позволит отложить использование специального средства отслеживания до тех пор, пока ваша система не станет достаточно сложной, чтобы оправдать такое решение.



Как только организуете агрегацию логов, создайте идентификаторы корреляции как можно скорее. Простые в создании на начальном этапе и сложные в последующей модернизации, они значительно повысят значимость ваших логов.

Привязка по времени

Глядя на список строк лога, можно легко обмануться, подумав, что мы видим точную хронологию, сообщающую, что произошло и в каком порядке. В конце концов, каждая строка логов содержит дату и время, так почему мы не можем использовать это как способ определить порядок возникновения событий? В последовательности вызовов в примере 10.2 отображается строка из лога `Шлюза`, за которой следуют логи сервисов `Стриминг`, `Покупатель`, `Электронная почта` и `Оплата`. Мы могли бы сделать вывод, что это фактический порядок возникновения вызовов. К сожалению, не всегда можно рассчитывать на это.

Строки лога генерируются на компьютерах, на которых запущены эти экземпляры микросервиса. После локальной записи в какой-то момент данные логи пересылаются. Это означает, что метки дат в строках лога генерируются на машинах, на которых запущены микросервисы. К сожалению, нельзя гарантировать, что часы на этих разных машинах синхронизированы. Например, часы на компьютере, где запущен микросервис `Электронная почта`, могут спешить на несколько секунд относительно часов на машине с запущенным микросервисом `Оплата`. В итоге все будет выглядеть, будто событие произошло в микросервисе `Электронная почта` до того, как что-то произошло в микросервисе `Оплата`, но такая картина складывается просто из-за рассинхронизации часов.

Несоответствие часов вызывает всевозможные проблемы в распределенных системах. Существуют протоколы, помогающие уменьшить перекос часов в системах. Протокол сетевого времени (Network Time Protocol, NTP) стал наиболее широко используемым вариантом. Однако работа NTP не дает гарантий. Он лишь уменьшает расхождение, а не устраняет его. Если есть последовательность вызовов, происходящих достаточно близко друг к другу, вы заметите, что даже секунды перекоса между машинами достаточно, чтобы понимание последовательности вызовов полностью изменилось.

По сути, есть два ограничения, когда речь идет о времени в логах. Мы не можем получить полностью точную информацию о времени для общего потока вызовов и понять причинно-следственную связь.

С точки зрения помощи в решении этой проблемы, чтобы мы могли уловить истинный порядок вещей, Лесли Лэмпорт[1] предложил логическую систему синхронизации, в которой счетчик используется для отслеживания порядка вызовов. Вы могли бы при желании реализовать аналогичную схему. Существует ряд ее вариаций. Лично я, однако, если бы хотел получить более точную информацию о порядке вызовов, а также более точное время, использовал бы инструмент распределенной трассировки, который решит обе проблемы. Мы рассмотрим распределенную трассировку более подробно позже в текущей главе.

Реализации

Не многие области в нашей отрасли остаются такими же спорными, как агрегация логов, где существует большое разнообразие решений.

Популярный набор инструментов с открытым исходным кодом для агрегирования логов заключается в использовании агента пересылки логов, такого как Fluentd (https://www.fluentd.org), для отправки логов в системе Elasticsearch (https://oreil.ly/m0Evo) с помощью программы Kibana (https://oreil.ly/zw8ds) как способа разобрать по крупицам результирующий поток логов. Самой большой проблемой с этим стеком, как правило, становятся накладные расходы на управление самой Elasticsearch, но это меньшая из проблем, если вам нужно запустить Elasticsearch для других целей или если вы используете управляющего инфраструктурой провайдера. Я озвучу два дополнительных предостережения относительно использования этого набора инструментов. Во-первых, в маркетинг Elasticsearch в качестве базы данных было вложено много сил и средств. Лично мне это всегда не нравилось. Взять нечто, что всегда называлось поисковым индексом, и переименовать его в БД может быть весьма проблематично. Мы делаем предположения в неявной форме о том, как действуют базы данных, и относимся к ним соответствующим образом, рассматривая их как источник достоверных жизненно важных сведений. Но по замыслу поисковый индекс не представляет собой источник истины — это его проекция. Система Elasticsearch в прошлом страдала от проблем, которые заставляют меня задуматься[2]. Хотя я уверен, что многие из этих проблем были решены, сегодня я скептически отношусь к использованию Elasticseach в определенных ситуациях и уж тем более в качестве БД. Наличие поискового индекса, способного иногда привести к потере данных, не представляется проблемой, если вы можете провести повторную индексацию. Но обращаться с ним как с базой данных — это совсем другое дело. Если бы

[1] *Lamport L.* Time, Clocks and the Ordering of Events in a Distributed System // Communications of the ACM 21, № 7. Июль 1978: 558.65, https://oreil.ly/qzYmh.

[2] Смотрите анализ, проведенный Кайлом Кингсбери в Elasticsearch 1.1.0 в статье Jepsen: Elasticsearch, https://oreil.ly/uO9wU; и в Elasticsearch 1.5.0 в статье Jepsen: Elasticsearch 1.5.0, https://oreil.ly/8fBCt.

я использовал данный стек и не мог позволить себе потерять информацию лога, я бы хотел убедиться, что смогу повторно проиндексировать исходные логи, если что-то пойдет не так.

Вторая группа проблем связана не столько с техническими аспектами Elasticsearch и Kibana, сколько с поведением Elastic — компании, стоящей за этими проектами. Недавно в Elastic приняли решение изменить лицензию на исходный код как для основной БД Elasticsearch, так и для Kibana с широко используемой и принятой лицензии с открытым исходным кодом (Apache 2.0) на Публичную лицензию на стороне сервера (Server Side Public License, SSPL)[1]. Причиной такого изменения, по-видимому, стала жесткая конкуренция. Такие организации, как AWS, сделали успешные коммерческие предложения, превзошедшие предложения Elastic. Помимо опасений, что SSPL может быть «вирусным» по своей природе (аналогично GNU General Public License — открытому лицензионному соглашению GNU), это решение привело многих в ярость. Более тысячи человек внесли свой вклад в Elasticsearch, предполагая, что они делают вклад в продукт с открытым исходным кодом. Иронично, что сам проект Elasticsearch и, следовательно, большая часть компании Elastic в целом были построены на технологии проекта с открытым исходным кодом Lucene (https://lucene.apache.org). На момент написания книги AWS предсказуемо взяла на себя обязательство создать и поддерживать форк Elasticsearch и Kibana с открытым исходным кодом под ранее использовавшейся лицензией Apache 2.0.

Во многих отношениях Kibana была достойной попыткой создать альтернативу с открытым исходным кодом дорогим коммерческим проектам, таким как Splunk. Какой бы хорошей ни казалась платформа Splunk, каждый ее клиент, с которым я общался, говорил, что эта платформа может быть невероятно дорогой с точки зрения как лицензионных сборов, так и затрат на оборудование. Однако многие из разработчиков действительно видят ценность этой платформы. При этом существует множество коммерческих вариантов. Лично я большой фанат Humio (https://www.humio.com), а кому-то нравится использовать Datadog (https://www.datadoghq.com) для агрегации логов. Существуют также базовые, но вполне работоспособные готовые решения для агрегирования логов у некоторых провайдеров облачных сервисов, таких как CloudWatch для AWS или Application Insights для Azure.

Реальность такова, что в этой сфере есть огромный выбор вариантов, от продуктов с открытым исходным кодом до коммерческих, от размещаемых самостоятельно до уже полностью размещенных на хостингах. Если вы хотите создать микросервисную архитектуру, это не тот вопрос, с которым вам должно быть трудно разобраться.

---

[1] *Losio R.* Elastic Changes Licences for Elasticsearch and Kibana: AWS Forks Both // InfoQ. 25 января 2021 года. https://oreil.ly/VdWzD.

Недостатки

Логи — это фантастический и простой способ быстрого получения информации из ваших работающих систем. Я по-прежнему убежден, что для микросервисной архитектуры на ранней стадии мало что может окупить инвестиции быстрее, чем логи, когда дело доходит до улучшения видимости вашего приложения в эксплуатационной среде. Они станут источником жизненной силы для сбора информации и диагностики. Тем не менее вы должны знать о некоторых потенциально серьезных проблемах с логами.

Как уже упоминалось, из-за рассинхронизации часов на них не всегда можно положиться, чтобы определить порядок, в котором происходили вызовы. Это несоответствие часов между машинами также означает, что точное определение времени последовательности вызовов будет проблематичным, что потенциально ограничивает полезность логов для отслеживания узких мест с задержкой.

Однако основная проблема с логами — генерация *огромного* количества данных по мере увеличения количества микросервисов и вызовов. Нагрузки. Невероятные объемы. Это может привести к росту затрат, поскольку потребуется больше оборудования, а также к увеличению платы, которую вы вносите поставщику услуг (некоторые провайдеры взимают плату за использование). И в зависимости от того, как построена ваша цепочка инструментов агрегации логов, это также может привести к проблемам масштабирования. Некоторые решения для агрегирования логов пытаются создать индекс при получении данных лога, чтобы ускорить запросы. Но поддержание индекса требует больших вычислительных мощностей, и чем больше логов вы получаете с увеличением индекса, тем больше проблем это принесет. Это приводит к необходимости более тщательно настраивать логи, чтобы уменьшить эту проблему, что, в свою очередь, приводит к увеличению объема работы и создает риск отложить регистрацию информации, которая может представлять ценность. Я разговаривал с командой, которая управляла кластером Elasticsearch для инструментов разработчика на базе SaaS. Члены команды обнаружили, что самый большой кластер Elasticsearch, который возможно было успешно запустить, способен обрабатывать только шесть недель ведения лога для одного из своих продуктов, из-за чего команде постоянно приходится перемещать данные, чтобы поддерживать управляемость. Мне нравится система Humio, потому что вместо поддержки индекса она фокусируется на эффективном и масштабируемом приеме данных с помощью некоторых интеллектуальных решений, в стремлении сократить время запросов.

Логи могут в конечном счете содержать очень много ценной и конфиденциальной информации. А это означает, что, найдя решение, способное хранить нужный вам объем логов, придется ограничить к ним доступ (что может еще больше усложнить вам работу по обеспечению коллективного владения вашими микросервисами в пользовательской среде) и они могут стать мишенью для злоумышленников. Поэтому потребуется рассмотреть возможность отказа от

регистрации определенных типов информации (мы рассмотрим этот вопрос в пункте «Будьте экономны» главы 11), чтобы уменьшить влияние доступа неавторизованных сторон. Если вы не храните данные — их нельзя украсть.

Агрегация метрики

Как и в случае с проблемой просмотра логов для разных хостов, нам нужно рассмотреть более эффективные способы сбора и просмотра данных о системах. Понять, что значит «хорошо», когда мы смотрим на метрику для более сложной системы, может быть достаточно затруднительно. Наш веб-сайт видит почти 50 кодов ошибок HTTP 4XX в секунду. Это плохо? Загрузка процессора в сервисе каталогов увеличилась на 20 % с обеда — что-то пошло не так? Секрет того, чтобы знать, когда паниковать, а когда расслабиться, заключается в сборе показателей, как ваша система ведет себя в течение достаточно длительного периода времени, чтобы выявить четкие закономерности.

В более сложной среде мы будем довольно часто предоставлять новые экземпляры микросервисов, поэтому от выбранной системы требуется сильное упрощение процесса сбора метрик с новых хостов. Необходимо получить возможность просматривать агрегированную метрику для всей системы, например среднюю загрузку процессора, а также агрегировать ее для всех экземпляров данного сервиса или даже для одного экземпляра. В итоге нам необходимо получить возможность связывать метаданные с метрикой, чтобы сделать вывод об этой структуре.

Еще одно ключевое преимущество понимания тенденций заключается в планировании производственных мощностей. Достигаем ли мы своего предела? Как скоро нам понадобятся новые хосты? В прошлом, когда мы покупали физические хостинги, это часто становилось ежегодной работой. В современную эпоху вычислений по требованию, предоставляемых поставщиками инфраструктуры как услуги (infrastructure as a service, IaaS), можно увеличивать или уменьшать масштаб за считанные минуты, если не секунды. Если мы понимаем свои модели использования, можно убедиться, что у нас достаточно инфраструктуры для удовлетворения своих потребностей. Чем тщательнее мы отслеживаем свои тенденции и знаем, что с ними делать, тем более экономичными и отзывчивыми могут быть наши системы.

Из-за подобного рода данных нам может потребоваться хранить эти метрики в разных разрешениях и сообщать о них. Например, мне необходима выборка процессора для моих серверов с разрешением по одной выборке каждые 10 секунд в течение последних 30 минут, чтобы лучше реагировать на ситуацию, которая разворачивается в текущий момент. С другой стороны, выборки ЦП с моих серверов за прошлый месяц, скорее всего, необходимы только для общего анализа тенденций, поэтому можно обойтись вычислением средней выборки ЦП на почасовой основе. Это часто делается на стандартных платформах метрик,

чтобы сократить время запросов, а также уменьшить объем хранения данных. Для чего-то столь простого, как частота процессора, это может стать хорошим подходом, но процесс агрегирования старых данных приводит к потере информации. Проблема с необходимостью агрегирования этих данных заключается в том, что вам часто приходится заранее решать, что агрегировать — вы должны заблаговременно определить, какую информацию не жалко потерять.

Стандартные инструменты метрик, безусловно, хороши для понимания тенденций или простых режимов сбоев. Но они часто не помогают сделать наши системы более наблюдаемыми, поскольку ограничивают круг задаваемых вопросов. Все становится интереснее, когда мы переходим от простых фрагментов информации, таких как время отклика, использование ЦП или дискового пространства, к более широкому мышлению о типах информации, которую мы хотим уловить.

Низкая или высокая кардинальность

Многие инструменты, особенно более поздние, были созданы для хранения и извлечения данных с высокой кардинальностью. Существует несколько способов описания кардинальности, но вы можете воспринимать ее как количество полей, которые можно легко запросить в выбранной точке данных. Чем больше потенциальных полей мы захотим запросить из наших данных, тем большую кардинальность потребуется поддерживать. По сути, это становится более проблематичным с базами данных временных рядов по причинам, которые я не буду здесь подробно описывать, но они связаны со способом построения многих из этих систем.

Например, мне могло бы понадобиться постоянно фиксировать и запрашивать имя микросервиса, идентификаторы клиента, запроса, продукта и номер сборки ПО. Затем я решаю собрать информацию о машине на этом этапе: ОС, системная архитектура, облачный провайдер и т. д. Мне может потребоваться собрать всю эту информацию для каждой собираемой точки данных. По мере того как увеличивается количество запрашиваемых параметров, возрастает кардинальность, и тем больше проблем будет у систем, которые не созданы с учетом этого варианта использования. Как объясняет Чарити Мейджорс[1], основатель Honeycomb:

> По сути, все сводится к метрике. Метрика — это точка данных, одно число с именем и некоторыми идентифицирующими тегами. Весь контекст, который вы можете получить, должен быть помещен в эти теги. Но запись всех этих тегов обходится дорого из-за того, как метрики хранятся на диске. Хранение метрики обходится очень дешево, а тега — очень дорого. Поэтому хранение большого количества тегов на одну метрику быстро выведет из строя ваш механизм хранения данных.

[1] *Majors C.* Metrics: Not the Observability Droids You're Looking For // Honeycomb (блог). 24 октября 2017 года. https://oreil.ly/TEETp.

Практически говоря, системы, построенные с учетом низкой кардинальности, будут испытывать большие трудности, если вы попытаетесь поместить в них данные с более высокой кардинальностью. Например, система Prometheus была создана для хранения довольно простых фрагментов информации, таких как частота процессора для выбранной машины. Во многих отношениях можно рассматривать Prometheus и подобные инструменты как отличную реализацию традиционного хранения метрик и запросов. Но отсутствие возможности поддерживать данные с более высокой кардинальностью — ограничивающий фактор. Разработчики Prometheus открыто говорят об этом ограничении (https://oreil.ly/LCoVM).

> Помните, что каждая уникальная комбинация пар «ключ — значение» меток представляет собой новый временной ряд, что значительно увеличивает объем хранимых данных. Не используйте метки для хранения измерений с высокой кардинальностью (много разных значений меток), таких как идентификаторы пользователей, адреса электронной почты или другие неограниченные наборы значений.

Системы, способные обрабатывать высокую кардинальность, дают больше возможностей задавать вашим системам множество различных вопросов, о которых вы не знали заранее. Это может быть трудно для понимания, особенно если вы довольно успешно управляли своей однопроцессной монолитной системой с помощью более «стандартных» инструментов. Даже люди, управляющие большими системами, довольствовались меньшей кардинальностью, часто по причине отсутствия выбора. Но по мере усложнения системы вам потребуется повышать качество выходных данных, предоставляемых системой, чтобы получить возможность улучшить ее наблюдаемость. Это означает сбор большего объема информации и наличие инструментов, позволяющих разбирать данные по крупицам.

Реализации

Со времен выхода первого издания этой книги Prometheus стал популярным инструментом с открытым исходным кодом для сбора и агрегирования метрик. Ранее я рекомендовал использовать Graphite (который я советовал в первом издании), но если вы ищете разумную замену, взгляните на Prometheus. Коммерческие предложения в этой области также значительно расширились: как новые, так и старые поставщики создают или переоснащают существующие решения для целевых пользователей микросервисов.

Однако у меня есть опасения по поводу данных с низкой и высокой кардинальностью. Системы, созданные для обработки данных с низкой кардинальностью, очень сложно модернизировать для поддержки хранения и обработки данных с высокой кардинальностью. Если вы ищете системы, способные хранить данные с высокой кардинальностью и управлять ими для более сложного наблюдения за поведением вашей системы, я бы настоятельно

рекомендовал рассмотреть Honeycomb (https://www.honeycomb.io/visualize) или Lightstep (https://lightstep.com). Хотя эти инструменты часто рассматриваются как решения для распределенной трассировки (которую мы рассмотрим подробнее позже), они обладают высокой способностью хранить, фильтровать и запрашивать данные с высокой кардинальностью.

СИСТЕМЫ МОНИТОРИНГА И НАБЛЮДАЕМОСТИ РАБОТАЮТ В ЭКСПЛУАТАЦИОННОЙ СРЕДЕ

Учитывая растущий набор инструментов, помогающих управлять микросервисной архитектурой, стоит помнить, что они сами по себе относятся к эксплуатируемым системам. Платформы агрегирования логов, средства распределенной трассировки, системы оповещения — все это критически важные приложения. Они так же необходимы, как и наше собственное ПО, если не больше. К поддержке инструментов эксплуатационного мониторинга стоит подходить с той же степенью осмотрительности, что и к ПО, которое мы пишем и поддерживаем.

Также эти инструменты могут стать потенциальными векторами сторонних атак. На момент написания книги правительство США и другие организации по всему миру занимались обнаружением брешей в программном обеспечении для управления сетью от SolarWinds. Хотя точная природа нарушения все еще изучается, считается, что это так называемая атака на цепочку поставок. После установки на сайтах клиентов (а SolarWinds используется 425 компаниями из списка Fortune 500 США), это ПО позволяло злоумышленникам получать внешний доступ к сетям клиентов, включая сеть Казначейства США.

Распределенная трассировка

До этого момента я в основном говорил о сборе информации изолированно. Да, мы собираем ее воедино, но понимание более широкого контекста, в котором она была собрана, может стать ключевым. По сути, микросервисная архитектура — это набор процессов, работающих вместе для выполнения какой-либо задачи. Мы рассмотрели различные способы координации этих действий в главе 6. Таким образом, когда нужно понять, как на самом деле ведет себя система в пользовательской среде, имеет смысл обратить внимание на взаимосвязи между нашими микросервисами. Это поможет лучше понять, как ведет себя система, оценить влияние проблемы или разобраться, что именно работает не так, как ожидалось.

По мере того как системы все более усложняются, становится важным получить способ видеть эти трассировки. Нам нужно иметь возможность извлекать разрозненные данные, чтобы получить единое представление о наборе коррелированных вызовов. Как мы уже видели, сделать что-то простое, например поместить идентификаторы корреляции в файлы логов, — хорошее начало. Но это довольно тривиальное решение, особенно потому, что в итоге придется создавать собственные инструменты для визуализации и обработки этих данных. Вот тут-то и приходит на помощь распределенная трассировка.

Как это работает

Хотя конкретные реализации различаются, в целом все инструменты распределенной трассировки работают одинаково. Локальная активность внутри потока фиксируется в *спане* (span). Эти отдельные спаны коррелируются с использованием некоторого уникального идентификатора. Затем они отправляются в центральный коллектор (сборщик), который способен построить их в единый *трейс* (trace). На рис. 10.7 показана картинка из Honeycomb, которая отображает трассировку через микросервисную архитектуру.

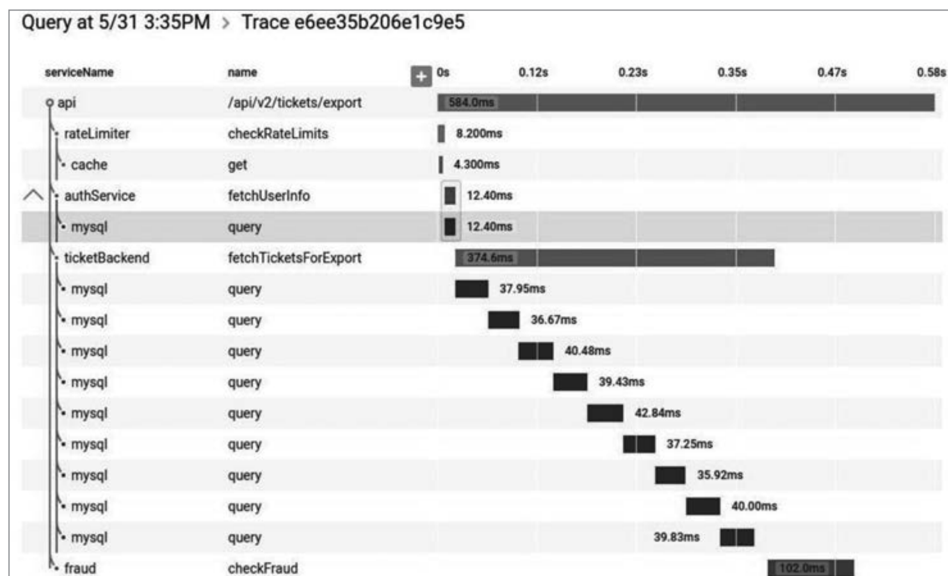


Рис. 10.7. Распределенная трассировка, показанная в Honeycomb, позволяет определить, где тратится время на операции, которые могут охватывать несколько микросервисов

Спаны позволяют собирать огромное количество информации. Какие именно данные вы собираете, будет зависеть от используемого протокола, но в случае с OpenTracing API каждый спан содержит время начала и окончания операции, набор логов, связанных со спаном, и произвольный набор пар «ключ — значение» для последующего запроса (их можно использовать для отправки таких данных, как идентификатор клиента или заказа, имя хоста, номер сборки и т. п.).

Сбор достаточного количества информации, позволяющей отслеживать вызовы в нашей системе, способен оказать прямое влияние на саму систему. Это приводит к необходимости формирования некоторой формы выборки, при которой определенная информация явно исключается из сбора трассировок, чтобы гарантировать дальнейшую работоспособность системы. Сложность заключается

в том, чтобы обеспечить исключение нужной информации и при этом собрать достаточное количество образцов для правильной экстраполяции наблюдений.

Стратегии выборки могут быть очень простыми. Система Dapper от Google, вдохновившая многих последующих разработчиков инструментов распределенной трассировки, выполняла очень агрессивную случайную выборку. Определенный процент вызовов попадал в выборку, и на этом все заканчивалось. Система Jaeger, например, по умолчанию фиксирует только 1 из 1000 вызовов. Задача — собрать достаточно информации для понимания, что делает наша система, а не перегружать ее данными. Такие инструменты, как Honeycomb и Lightstep, обеспечивают более детальную динамическую выборку, чем простой случайный набор данных. Примером динамической выборки может быть ситуация, когда вам требуется больше выборок для определенных типов событий. Например, необходимо сделать выборку всего, что генерирует ошибку, но вам было бы достаточно выборки только одной из ста успешных операций, если все они достаточно похожи.

Реализация распределенной трассировки

Для запуска распределенной трассировки в вашей системе необходимо выполнить несколько действий. Во-первых, захватить информацию в спан внутри ваших микросервисов. Если вы используете стандартный API, такой как OpenTracing или более новый OpenTelemetry API, вы обнаружите, что у некоторых из сторонних библиотек и фреймворков есть встроенная поддержка этих API, и они уже будут отправлять полезную информацию (например, автоматический сбор информации о HTTP-вызовах). Но даже если они это сделают, скорее всего, вам все равно потребуется использовать свой собственный код, предоставляя полезную информацию о том, что делает ваш микросервис в конкретный момент времени.

Во-вторых, вам понадобится какой-то способ отправить эту информацию в спане в коллектор — возможно, вы отправляете эти данные непосредственно из экземпляра микросервиса, но гораздо чаще используется локальный агент пересылки. Так же как и при агрегировании логов, агент запускается локально в своем экземпляре микросервиса и периодически отправляет информацию из спана центральному сборщику. Применение локального агента обычно позволяет использовать некоторые более продвинутые возможности, такие как изменение выборки или добавление дополнительных тегов, а также может более эффективно буферизировать отправляемую информацию.

Конечно, вам нужен сборщик, способный получить эту информацию и разобраться во всем этом.

В области продуктов с открытым исходным кодом Jaeger стал популярным выбором для распределенной трассировки. Что касается коммерческого инструментария, я бы начал с уже упомянутых Lightstep и Honeycomb. Однако я настоятельно рекомендую вам выбрать что-то, что предназначено для поддержки OpenTelemetry API. OpenTelemetry (https://opentelemetry.io) — это от-

крытая спецификация API, значительно облегчающая создание кода, такого как драйверы БД или веб-фреймворки, с поддержкой трассировки, а также упрощающая переносимость между различными поставщиками на стороне сбора данных. Основываясь на работе, проделанной более ранними API OpenTracing и OpenConsensus, данный API получил широкую поддержку в ИТ-индустрии.

Все ли у нас в порядке?

Мы много говорили о том, что вы могли бы делать как оператор системы: какое мышление вам нужно, какую информацию собрать. Но как узнать, делаете ли вы слишком много или же недостаточно? Как узнать, насколько хорошо вы выполняете свою работу, достаточно ли хорошо работает ваша система?

Бинарные концепции работоспособности или неработоспособности системы начинают обретать все меньшее значение по мере ее усложнения. В монолитной системе с одним процессом проще рассматривать состояние системы в черно-белом цвете. Но как насчет распределенной системы? Если один экземпляр микросервиса недоступен, станет ли это проблемой? «Здоров» ли экземпляр, если он доступен для подключения? Как насчет ситуации, в которой микросервис `Возвраты` доступен, но половина предоставляемой им функциональности требует использования нижестоящего микросервиса `Запасы`, который в настоящее время испытывает проблемы? Означает ли это, что мы считаем микросервис `Возвраты` исправным или нет?

По мере усложнения ситуации важно сделать шаг назад и взглянуть на проблему с другой стороны. Представьте пчелиный улей. Мы смотрим на отдельную пчелу и понимаем, что она недовольна. Возможно, она потеряла одно из своих крыльев и поэтому больше не может летать. Это, безусловно, проблема отдельной пчелы, но можете ли вы на основании этого сделать какие-либо выводы о благополучии самого улья? Нет, вам нужно было бы взглянуть на состояние улья более комплексно. Если одна пчела больна, это не значит, что болен весь улей.

Можно попытаться определить, *исправен* ли сервис, задав, например, контрольный показатель уровня загрузки процессора или приемлемого времени отклика. Если наша система мониторинга обнаружит, что фактические значения выходят за пределы этого безопасного диапазона, нам приходит оповещение. Однако во многих отношениях эти значения на один шаг удалены от того, что мы на самом деле хотим отслеживать, а именно: *работает ли система*? Чем сложнее взаимодействие между сервисами, тем дальше мы находимся от реального ответа на этот вопрос, просто рассматривая одну метрику изолированно.

Таким образом, можно собрать много информации, но сама по себе она не дает понимания, работает ли система должным образом. Для этого нам нужно начать думать немного шире в плане определения приемлемого поведения. В области *обеспечения надежности систем* (site reliability engineering, SRE) была проделана внушительная работа, в центре внимания которой находится обеспечение

гарантий надежности ваших систем во время изменений. И тут есть несколько полезных концепций для изучения.

Пристегнитесь — мы въезжаем в город аббревиатур.

Соглашение об уровне услуг

Соглашение об уровне услуг (service-level agreement, SLA) — это соглашение, достигнутое между создающими и использующими систему людьми. В нем описываются не только ожидания пользователей, но и последствия, если система не достигнет приемлемого уровня поведения. SLA, как правило, очень близки к «абсолютному минимуму», часто до такой степени, что, если бы система уже достигла своих целей, конечный пользователь все равно был бы недоволен. В качестве примера, у AWS есть SLA для своего вычислительного сервиса. В нем четко указано, что не существует гарантии бесперебойной работы для одного экземпляра EC2 (управляемой виртуальной машины[1]). AWS заявляет, что прилагает все усилия, чтобы обеспечить 90 % безотказной работы для отдельного экземпляра, но если это не достигается, то он просто не взимает с вас плату за определенный час, в течение которого экземпляр был недоступен. Если ваши экземпляры EC2 постоянно находятся ниже 90 % уровня доступности в течение определенного часа, что приводит к значительной нестабильности системы, вы можете это время не оплачивать. Но приятного в этом мало. По моему опыту, AWS на практике достигает гораздо больших целей, чем указано в SLA, как это часто бывает в принципе.

Цель уровня услуг

Предоставить выполнение SLA одной команде достаточно проблематично, особенно если у SLA несколько широкий и сквозной характер. На уровне команды мы вместо этого говорим о *целях уровня услуг* (SLO, service-level objectives). Цели SLO определяют, что команда обязуется предоставить. Достижение SLO каждой командой по всей организации удовлетворит (и, вероятно, значительно превысит) требования SLA. Пример SLO может включать такие вещи, как ожидаемое время безотказной работы или приемлемое время отклика для данной операции.

Считать SLO лишь способом достижения соглашений SLA — слишком упрощенное представление. Да, если вся организация достигает всех своих целей SLO, мы бы предположили, что все SLA также были достигнуты, но SLO могут описывать другие цели, не изложенные в SLA. Или они могут быть амбициозными и обращенными на саму систему (попытка осуществить некоторые внутренние изменения). SLO часто отражают, чего хочет достичь сама команда, и это не всегда имеет отношение к соглашению SLA.

[1] В основном AWS теперь предоставляет экземпляры «на голом железе», что вносит определенную путаницу в мои мысли.

Индикаторы уровня услуг

Чтобы определить, соответствует ли наша работа целям SLO, необходимо собрать реальные данные, то есть *индикаторы уровня услуг* (service-level indicators, SLI). SLI — это показатель того, что делает наше ПО. Это может быть, например, время отклика от процесса, регистрация клиента, сообщение об ошибке пользователю или размещение заказа. Необходимо собрать и вывести индикаторы SLI, чтобы убедиться, что мы выполняем свои цели SLO.

Бюджеты ошибок

Когда мы пробуем что-то новое, мы привносим больше потенциальной нестабильности в свои системы. Таким образом, желание сохранить (или улучшить) стабильность системы может привести к нежеланию модифицировать ее. *Бюджеты ошибок* — это попытка избежать такого рода проблем, четко определив, какой объем ошибок допустим в системе.

Если вы уже определились со SLO, то разработка вашего бюджета ошибок должна быть довольно четкой. Например, можно сказать, что микросервис должен быть доступен 99,9 % времени в квартал в режиме 24/7. Это означает, что на самом деле сервис может отключаться на 2 часа 11 минут в квартал. С точки зрения данного SLO это ваш бюджет ошибок.

Бюджеты ошибок дают четкое представление, насколько хорошо вы достигаете (или нет) SLO, позволяя принимать более взвешенные решения по рискам. Если ваш бюджет ошибок на квартал имеет значительный запас, то, возможно, вы согласитесь на развертывание микросервиса, написанного на новом языке программирования. Если же вы уже превысили его, возможно, стоит отложить внедрение и вместо этого выделить больше времени команде на повышение надежности системы.

Такой запас прочности позволяет вашим командам попробовать что-то новое.

Оповещение

Иногда (надеюсь, редко, но, вероятно, чаще, чем хотелось бы) в наших системах происходит что-то, что потребует уведомления оператора-человека для принятия мер. Микросервис неожиданно стал недоступным, отобразилось большее количество ошибок, чем ожидалось, или, возможно, вся система стала недоступной для пользователей. В таких ситуациях нужно, чтобы люди были в курсе, что происходит, чтобы попытаться исправить положение.

Проблема в том, что при микросервисной архитектуре, учитывая большее количество вызовов и процессов, а также более сложную базовую инфраструктуру, часто что-то идет не так. Задача в микросервисной среде — точно определить, о каких типах проблем должен быть оповещен человек и как именно.

Одни проблемы хуже других

Когда что-то идет не так, мы хотим знать об этом. Или не хотим? Все ли проблемы одинаковы? По мере увеличения источников проблем важно научиться расставлять приоритеты, чтобы решить, следует ли привлекать оператора и каким образом. Часто самый важный вопрос, который я задаю себе, когда дело доходит до оповещения, звучит так: «Эта проблема стоит того, чтобы разбудить кого-то среди ночи?»

Я видел один пример такого образа мышления много лет назад, когда был в кампусе Google. В приемной одного из зданий в Маунтин-Вью стояла старая стойка с машинами, служившая своего рода экспонатом. Я кое-что заметил: эти серверы не были в корпусах (это были просто голые материнские платы, вставленные в стойку), а жесткие диски были прикреплены с помощью липучек. Я спросил одного из сотрудников, почему именно так. «О, — сказал он, — жесткие диски так часто выходят из строя, что никому не хочется их прикручивать. Мы просто отрываем их, выбрасываем в мусорное ведро и вставляем новые».

Созданные Google системы предполагали, что жесткие диски выйдут из строя. Они оптимизировали конструкцию этих серверов, чтобы обеспечить максимально простую замену жесткого диска. Поскольку система была сконструирована так, чтобы выдерживать отказ HDD. И хотя было важно, чтобы диск в итоге был заменен, скорее всего, сбой отдельного винчестера не вызовет каких-либо существенных проблем, заметных пользователю. С тысячами серверов в дата-центре Google было бы чьей-то ежедневной задачей просто ходить вдоль ряда стоек и менять жесткие диски по мере их выхода из строя. Конечно, отказ HDD был проблемой, но с ней можно было легко справиться. Выход из строя жесткого диска считался обычным делом и не приходилось вызывать сотрудников в нерабочее время. Это стало такой неполадкой, о которой сообщалось в течение обычного рабочего дня.

По мере увеличения источников потенциальных проблем вам нужно будет расставлять приоритеты в отношении того, какие события вызывают те или иные типы предупреждений. В противном случае вполне может оказаться, что вам трудно отделить тривиальное от неотложного.

Усталость от оповещений

Часто слишком большое количество предупреждений может вызвать серьезные проблемы. В 1979 году на атомной электростанции Три-Майл-Айленд в США произошло частичное расплавление реактора. Расследование инцидента выявило, что операторы объекта были настолько перегружены предупреждениями, что не могли определить, какие действия необходимо предпринять. Поступило предупреждение, указывающее на основную проблему, которую необходимо решить, но это не было чем-то очевидным для операторов, поскольку одновременно срабатывало множество других предупреждений. Во время публичных слушаний по инциденту один из операторов, Крейг Фауст, вспоминал: «Я бы с удовольствием выбросил панель сигнализации. Она не дала нам никакой полез-

ной информации». В отчете об инциденте был сделан вывод, что диспетчерская «была совершенно не приспособлена для управления аварией»[1].

Совсем недавно мы столкнулись с проблемой слишком большого количества предупреждений в связи с серией инцидентов с самолетом 737 Max. Это были две отдельные авиакатастрофы, в результате которых в общей сложности погибло 346 человек. В первоначальном отчете[2] Национального совета по безопасности на транспорте (National Transportation Safety Board, NTSB), посвященном этим проблемам, было обращено внимание на сбивающие с толку оповещения, которые срабатывали в реальных условиях и считались факторами, способствующими авариям. Из отчета:

> Исследование человеческого фактора показало, что в нештатных условиях, например при отказе системы с многочисленными оповещениями, когда может потребоваться несколько действий летного экипажа, крайне важно дать пилотам понять, какие действия должны быть приоритетными. Это особенно актуально в случае функций, реализованных в нескольких системах самолета, поскольку сбой в одной системе в рамках высокоинтегрированных системных архитектур может представлять множество предупреждений и указаний для летного экипажа, так как каждая система взаимодействия регистрирует сбой... Таким образом, важно, чтобы взаимодействие в системе и интерфейс приборной панели были спроектированы так, чтобы указать пилотам на действия с наивысшим приоритетом.

Итак, здесь мы говорим об эксплуатации ядерного реактора и управлении самолетом. Я подозреваю, что многие из вас прямо сейчас задаются вопросом, какое это имеет отношение к создаваемой вами системе. Возможно (хотя и маловероятно), что вы не создаете подобные критически важные для безопасности системы, но мы можем многому научиться на этих примерах. И то и другое связано с очень сложными взаимосвязанными системами, в которых проблема в одной области может вызвать проблему в другой. А когда мы генерируем слишком много оповещений или не даем операторам возможности расставить приоритеты, на чем следует сосредоточить внимание, может произойти катастрофа. Перегрузка оператора оповещениями может вызвать реальные проблемы. Снова из отчета:

> Кроме того, исследование реакции пилотов на множественные/одновременные аномальные ситуации, наряду с данными об авариях, показывает, что множественные конкурирующие предупреждения могут превышать доступные умственные ресурсы и сужать фокус внимания, что приводит к запоздалым или неадекватно расставленным по приоритетам ответам.

---

[1] United States President's Commission on the Accident at Three Mile Island, The Need for Change, the Legacy of TMI: Report of the President's Commission on the Accident at Three Mile Island. The Commission, 1979.

[2] National Transportation Safety Board, Safety Recommendation Report: Assumptions Used in the Safety Assessment Process and the Effects of Multiple Alerts and Indications on Pilot Performance. NTSB, 2019.

Так что дважды подумайте, прежде чем просто отправлять больше оповещений оператору, — вы можете не получить желаемого результата.

ТРЕВОГА ИЛИ ОПОВЕЩЕНИЕ

При более широком рассмотрении темы оповещения я обнаружил множество невероятно полезных исследований и практик из различных контекстов. Во многих из них конкретно не говорилось об оповещении в ИТ-системах. Термин «тревога» (alarm) часто встречается при изучении данной темы в инженерном деле и за его пределами, в то время как мы склонны чаще использовать термин «оповещение» (alert) в ИТ. Я побеседовал с несколькими людьми, понимающими различие между этими двумя словами. Но, как ни странно, различия, которые они проводили, не казались последовательными. Основываясь на том, что большинство людей считают *оповещение* и *тревогу* практически синонимами, я решил использовать слова «оповещение», «предупреждение» для этой книги.

На пути к лучшему оповещению

Поэтому оповещения должны быть полезными, а также необходимо избежать слишком большого их количества. На какие рекомендации стоит обратить внимание, чтобы получить помощь в создании более качественных оповещений?

Стивен Шоррок подробно останавливается на этой теме в своей статье Alarm Design: From Nuclear Power to WebOps[1]. Для тех, кому интересна эта тема, данная статья станет хорошей отправной точкой для дальнейшего изучения. Из статьи:

> Цель [оповещений] — направить внимание пользователя на важные аспекты текущих операций или работы оборудования, которые требуют своевременного участия.

Есть полезный набор правил от Ассоциации пользователей инженерного оборудования и материалов (Engineering Equipment and Materials Users Association, EEMUA). В ней разработали, на мой взгляд, самое исчерпывающее описание того, что делает оповещение хорошим.

Актуальность

Убедитесь, что оповещение представляет ценность.

Уникальность

Убедитесь, что оповещение не дублирует другое.

Своевременность

Необходимо получить оповещение достаточно быстро, чтобы вовремя отреагировать на него.

---

[1] *Shorrock S.* Alarm Design: From Nuclear Power to WebOps // Humanistic Systems (блог). 16 октября 2015 года. https://oreil.ly/RCHDL.

Приоритетность

Предоставьте оператору достаточно информации, чтобы он мог решить, в каком порядке следует обрабатывать предупреждения.

Понятность

Информация в предупреждении должна быть ясной и читаемой.

Диагностичность

Должно быть ясно, что не так.

Консультативность

Помогите оператору понять, какие действия необходимо предпринять.

Сфокусированность

Обратите внимание на наиболее важные вопросы.

Оглядываясь на тот этап карьеры, когда мне довелось работать в службе поддержки, я с грустью вспоминаю, как редко оповещения, с которыми мне приходилось иметь дело, следовали какому-либо из этих правил.

К сожалению, слишком часто люди, предоставляющие информацию и получающие оповещения, — это разные люди. Снова из Шоррока:

> Понимание природы обработки сигналов тревоги и связанных с этим процессом вопросов проектирования может помочь вам — эксперту в своей работе — стать более информированным пользователем, и способствует созданию лучших систем оповещения для поддержки вашей работы.

Одна из техник, которая поможет уменьшить количество предупреждений, привлекающих наше внимание, заключается в изменении представления о том, на какие проблемы указать оператору в первую очередь. Давайте рассмотрим эту тему далее.

Семантический мониторинг

С помощью семантического мониторинга определяется модель приемлемой семантики нашей системы. Какими свойствами должна обладать система, чтобы мы думали, что она работает в пределах допустимых значений? В значительной степени семантический мониторинг требует изменения нашего поведения. Вместо того чтобы искать наличие ошибок, необходимо постоянно задавать один вопрос: ведет ли система себя так, как мы ожидаем? Если да, то это лучше помогает понять, как расставить приоритеты в работе с появляющимися ошибками.

Следующее, что нужно решить, — как определить модель для правильно работающей системы. При таком подходе вы можете стать сверхформальными (буквально, поскольку некоторые организации используют для этого формальные

методы), но создание нескольких простых заявленных значений поможет вам улучшить свою систему. Например, в случае с MusicCorp: что должно быть истиной, чтобы мы были уверены в корректности работы системы? Возможно, мы скажем, что:

- новые покупатели могут зарегистрироваться;
- в пиковое время мы продаем товаров на сумму не менее 20 000 долларов в час;
- мы отправляем заказы с нормальной скоростью.

Если эти три утверждения выполняются, то в целом можно считать, что система работает достаточно хорошо. Возвращаясь к SLA и SLO, можно предположить, что наша модель семантической корректности значительно превышает наши обязательства по SLA, и ожидается, что у нас будут конкретные цели SLO, позволяющие отслеживать соответствие этой модели. Другими словами, эти заявления о том, какие действия мы ожидаем от ПО, будут иметь большое значение для определения целей SLO.

Одна из самых больших проблем заключается в достижении согласия относительно того, что представляет собой эта модель. Как вы могли заметить, мы не говорим о таких низкоуровневых вещах, как «использование диска не должно превышать 95 %», мы делаем заявления более высокого уровня, о системе. Как оператор системы или человек, написавший и протестировавший микросервис, вы, возможно, не в состоянии решить, какими должны быть эти заявленные значения. В организации доставки, ориентированной на продукт, этим должен заниматься его владелец, но ваша работа как оператора может заключаться в том, чтобы убедиться, что обсуждение с владельцем продукта действительно происходит.

Как только вы определитесь с моделью, необходимо выяснить, соответствует ли текущее поведение системы данной модели. Грубо говоря, у нас есть два ключевых способа сделать это: мониторинг реального пользователя и синтетические транзакции. К последнему мы вернемся чуть позже, а пока давайте рассмотрим мониторинг реальных пользователей.

Мониторинг реальных пользователей

С помощью мониторинга реальных пользователей можно рассмотреть, что на самом деле происходит в эксплуатируемой системе, и сравнить это со своей семантической моделью. В MusicCorp стоило бы рассмотреть, сколько клиентов зарегистрировалось, сколько заказов мы отправили и т. д.

Сложность мониторинга реальных пользователей состоит в том, что зачастую необходимая информация не может быть получена своевременно. Учтите, MusicCorp ожидает продавать продукцию на сумму не менее 20 000 долларов в час. Если эта информация хранится где-то в БД, мы можем не успеть собрать ее и принять соответствующие меры. Вот почему требуется улучшить предо-

ставление доступа к информации, которая ранее считалась бизнес-метриками для вашего инструментария. Если у вас есть возможность передавать частоту процессора в свое хранилище метрик и его можно использовать для оповещения о состоянии ЦП, то почему вы не можете также записать продажу и стоимость в долларах в это же хранилище?

Одним из главных недостатков мониторинга реальных пользователей стало то, что он в принципе «шумный». Вы получаете настолько много информации, что просеять ее может быть непросто. Также стоит понимать, что мониторинг реального пользователя сообщает об уже произошедших событиях, и в результате вы можете не заметить проблему до тех пор, пока она не проявится. Если покупатель не смог зарегистрироваться — это недовольный клиент. Синтетические транзакции (еще одна форма эксплуатационного тестирования, которую мы рассмотрим чуть позже) дают возможность не только снизить уровень информационного шума, но и выявить проблемы до того, как пользователи узнают о них.

Тестирование в эксплуатации

> Не проводить тестирование в эксплуатации — это все равно что не репетировать с оркестром, потому что ваше соло прекрасно звучало дома[1].
>
> *Чарити Мейджорс*

Как мы неоднократно отмечали на протяжении всей книги, начиная с обсуждения канареечного релиза в подразделе «Канареечный релиз» главы 8 и заканчивая законом о балансировке в отношении предварительного и пострелизного тестирования, проведение той или иной формы тестирования в эксплуатации может быть невероятно полезным и безопасным занятием. В данной книге мы рассмотрели несколько различных типов тестирования в эксплуатации, и, кроме них, существует еще несколько форм такого тестирования, поэтому я подумал, что было бы полезно обобщить некоторые из них, а также поделиться другими часто используемыми примерами. Меня удивляет большое количество людей, которых пугает концепция тестирования в эксплуатации, хотя они уже фактически делают это, сами того не осознавая.

Все модели подобного тестирования, возможно, относятся к некой форме мониторинговой деятельности. Мы привносим эти формы тестирования в эксплуатацию, чтобы убедиться, что наша система работает так, как ожидалось, и многие виды эксплуатационного тестирования могут быть невероятно эффективными в выявлении проблем еще до того, как пользователи с ними столкнутся.

---

[1] Charity Majors (@mipsytipsy), Twitter, 7 июля 2019 года, 9:48 утра. https://oreil.ly/4VUAX.

Синтетические транзакции

При помощи синтетических транзакций внедряется ложное поведение пользователей в эксплуатируемую систему. Это поведение получает известные входные данные и ожидаемые выходные данные. Например, для MusicCorp мы могли бы искусственно создать нового покупателя, а затем проверить, что покупатель был успешно создан. Эти транзакции будут выполняться на регулярной основе, что даст возможность как можно быстрее устранять проблемы.

Впервые я реализовал такой подход еще в 2005 году. Я состоял в небольшой команде Thoughtworks, создававшей систему для инвестиционного банка. В течение всего торгового дня происходило множество событий, отражавших изменения на рынке. Нашей задачей было отреагировать на эти изменения и оценить их влияние на портфель банка. Мы работали в довольно сжатые сроки, и требовалось завершить все наши расчеты менее чем через 10 секунд после поступления события. Сама система состояла примерно из пяти отдельных сервисов, по крайней мере один из них работал в вычислительной сети, которая, помимо прочего, очищала неиспользуемые циклы процессора примерно на 250 настольных компьютерах в центре аварийного восстановления банка.

Количество подвижных элементов в системе означало, что многие собираемые нами показатели более низкого уровня создавали много информационного шума. У нас также не было возможности постепенного масштабирования или запуска системы в течение нескольких месяцев, чтобы понять, что есть «хорошо» с точки зрения низкоуровневых показателей, таких как частота процессора или время отклика. Наш подход заключался в создании ложных событий, чтобы оценить часть портфеля, которая не была забронирована в нижестоящих системах. Примерно каждую минуту мы использовали инструмент под названием Nagios для запуска задания командной строки, которое вставляло поддельное событие в одну из наших очередей. Система подхватывала его и выполняла все различные вычисления точно так же, как и с любым другим заданием, за исключением того, что результаты появлялись в «мусорной» книге, используемой только для тестирования. Если изменение цен не было замечено в течение определенного времени, Nagios сообщал об этом как о проблеме.

На практике я обнаружил, что использование синтетических транзакций для выполнения семантического мониторинга, подобного этому, представляет собой гораздо лучший индикатор проблем в системах, чем оповещение о метриках более низкого уровня. Однако они не заменяют необходимость в детализации более низкого уровня — нам все равно понадобится эта информация, когда потребуется выяснить, почему синтетическая транзакция завершилась неудачей.

Реализация синтетических транзакций. В прошлом реализация синтетических транзакций была довольно сложной задачей. Но все в мире развивается и средства для их реализации теперь есть у нас под рукой! Вы проводите тесты для своих систем, верно? Если нет, прочитайте главу 9 и возвращайтесь. Готово? Хорошо!

Если посмотреть на имеющиеся тесты, проверяющие определенный сервис от начала до конца или даже всю систему, у нас есть многое из того, что необходимо для реализации семантического мониторинга. Наша система уже предоставляет хуки, необходимые для запуска теста и проверки результата. Так почему бы просто не запускать подмножество этих тестов на постоянной основе как способ мониторинга системы?

Конечно, кое-что нам нужно сделать. Необходимо внимательно отнестись к требованиям к информации для наших тестов. Возможно, потребуется найти способ адаптации своих тестов к различным оперативным данным, если они меняются с течением времени, или же установить другой источник данных. Например, у нас может быть набор фейковых пользователей, используемых в работающем приложении с известным набором данных.

Точно так же мы должны быть уверены, что случайно не спровоцируем непредвиденных побочных эффектов. Мой друг рассказал историю о компании электронной коммерции, которая случайно провела тесты своих работающих систем заказов. Компания не осознавала своей ошибки до тех пор, пока в головной офис не прибыло большое количество стиральных машин.

A/B-тестирование

С помощью A/B-теста вы развертываете две разные версии одной и той же функции, при этом пользователи видят либо функциональность A, либо B. Затем вы оцениваете, какая версия работает лучше всего. Такой подход обычно используется при попытке выбрать между двумя различными моделями реализации функциональности, например, можно попробовать две разные формы регистрации клиентов, чтобы увидеть, какая из них более эффективна для увеличения количества зарегистрированных пользователей.

Канареечный релиз

Небольшая часть ваших пользователей получает возможность попользоваться новой версией функциональности. Если функция работает хорошо, вы можете увеличить количество пользователей, которым она будет доступна. Так продолжается до тех пор, пока новая версия не станет доступной для всех. С другой стороны, если выпущенный вариант продукта работает не так, как предполагалось, вы можете либо отменить изменение, либо попытаться устранить недочет, в любом случае проблема затрагивает лишь определенную часть клиентов.

Параллельное выполнение

При параллельном выполнении вы запускаете две разные реализации одной и той же функциональности одновременно. Любой запрос пользователя направляется в обе версии, и их результаты можно сравнить. Таким образом, вместо того, чтобы направлять трафик пользователя либо на старую, либо на

новую версию, как при канареечном релизе, мы выполняем обе версии, но пользователь видит только одну. Это позволяет провести полное сравнение между двумя различными вариантами, что чрезвычайно полезно, когда необходимо лучше понять такие аспекты, как характеристики нагрузки новой реализации некоторых ключевых функций.

Дымовые тесты

Используются после развертывания ПО в рабочей среде, но перед его релизом. Дымовые тесты выполняются, чтобы убедиться, что ПО работает должным образом. Данные тесты обычно полностью автоматизированы и могут варьироваться от очень простых действий, таких как проверка работоспособности определенного микросервиса, до полномасштабных синтетических транзакций.

Синтетические транзакции

В систему внедряется полномасштабное фейковое взаимодействие с пользователем. Часто это очень похоже на сквозной тест, который вы могли бы написать.

Хаос-инжиниринг

Эту тему мы подробнее обсудим в главе 12. Хаос-инжиниринг может включать в себя внедрение ошибок в активно работающую систему, чтобы гарантировать ее способность справиться с этими ожидаемыми проблемами. Наиболее известным примером этого метода, вероятно, стал инструмент Chaos Monkey от Netflix. Он способен отключать виртуальные машины в среде эксплуатации, ожидая, что система достаточно надежна, чтобы эти отключения не сказывались на конечном пользователе.

Стандартизация

Как мы уже говорили ранее, одно из постоянных балансирующих действий, которые вам придется совершать, заключается в определении того, где необходимо принимать решения только для одного микросервиса, а где нужно стандартизировать всю вашу систему. На мой взгляд, мониторинг и наблюдаемость — это одна из областей, в которой стандартизация может быть невероятно важна. Поскольку микросервисы взаимодействуют множеством различных способов для предоставления функциональных возможностей пользователям, использующим несколько интерфейсов, необходимо рассматривать систему целостно.

Вы должны стараться записывать логи в стандартном формате. Определенно надо хранить все метрики в одном месте, и, возможно, вам также захочется создать список стандартных имен для метрик. Было бы очень досадно, если бы

у одного сервиса была метрика с именем `ResponseTime`, а у другого — с именем `RspTimeSecs`, хотя они означают одно и то же.

Как всегда в случае стандартизации, здесь могут помочь специализированные инструменты. Как я уже говорил ранее, главное — упростить выполнение правильных действий, поэтому наличие платформы со многими базовыми строительными блоками, такими как агрегация логов, имеет большой смысл. Все чаще весомая часть этого ложится на плечи команды платформы, роль которой мы более подробно рассмотрим в главе 15.

Выбор инструментов

Как мы уже говорили, потенциально существует множество различных инструментов, которые вам могут понадобиться для улучшения наблюдаемости системы. Но это быстро развивающаяся область, и весьма вероятно, что в будущем мы будем использовать инструменты, сильно отличающиеся от тех, что у нас есть сейчас. Учитывая, что такие платформы, как Honeycomb и Lightstep, лидируют в плане того, как выглядит инструментарий наблюдаемости для микросервисов, а остальные инструменты, представленные на рынке, в какой-то степени играют в догонялки, я ни на секунду не сомневаюсь, что в будущем в этой сфере произойдет множество изменений.

Так что вполне возможно, что вам понадобятся инструменты, отличные от тех, что у вас есть на данный момент, если вы только осваиваете микросервисы, и, вполне возможно, вам понадобятся другие инструменты и в будущем, поскольку прогресс не стоит на месте. Имея это в виду, я хочу поделиться несколькими мыслями об очень важных, на мой взгляд, критериях для любого инструмента в этой области.

Демократичность

Если у вас есть инструменты, с которыми могут работать только опытные операторы, то вы ограничиваете количество людей, способных участвовать в производственной деятельности. Аналогично если вы выберете настолько дорогие инструменты, что их использование запрещается в любых ситуациях, кроме критических, то разработчики не смогут изучить их, пока не станет слишком поздно.

Выбирайте инструменты, учитывающие потребности всех людей, которые будут ими пользоваться. Если вы действительно хотите перейти к модели коллективного владения программным обеспечением, то ПО должно быть доступно всем членам команды. Убедитесь, что выбранный вами инструментарий будет также использоваться в средах разработки и тестирования, — это поможет сделать эту цель реальностью.

Простота интеграции

Получение необходимой информации из архитектуры вашего приложения и систем, в которых вы работаете, жизненно важно, и, как мы уже говорили, вам может потребоваться извлекать больше сведений, чем раньше, и в разных форматах. Крайне важно максимально упростить этот процесс. Такие начинания, как OpenTracing (https://opentracing.io), помогли нам в плане предоставления стандартных API, способных поддерживать клиентские библиотеки и платформы, что упрощает интеграцию и переносимость между цепочками инструментов. Особый интерес, как я уже говорил, представляет новый проект OpenTelemetry, продвигаемый большим числом участников.

Выбор инструментов, поддерживающих эти открытые стандарты, снизит трудозатраты по интеграции, а также, возможно, упростит последующую смену поставщиков.

Обеспечение контекста

Когда я просматриваю фрагмент информации, мне нужен инструмент, предоставляющий как можно больше контекста, чтобы помочь понять, что может произойти дальше. Мне очень нравится следующая система категоризации для различных типов контекста, которую я нашел в блоге Lightstep[1].

Временной контекст

Как данные выглядят по сравнению с тем, что было минуту, час, день или месяц назад?

Относительный контекст

Как данные изменились по отношению к другим элементам в системе?

Реляционный контекст

От этих данных что-то зависит? Зависят ли эти данные от чего-то еще?

Пропорциональный контекст

Насколько это плохо? Каков масштаб проблемы? На что это влияет?

Своевременность

Вы не можете ждать этой информации целую вечность. Она нужна вам сейчас. Ваше определение «сейчас», конечно, может несколько отличаться, но в контексте систем сведения нужны достаточно быстро, чтобы получить шанс обнаружить проблему раньше, чем это сделает пользователь, или по крайней мере иметь информацию под рукой, когда кто-то пожалуется. На практике речь идет о секундах, а не о минутах или часах.

[1] Observability: A Complete Overview for 2021 // Lightstep, доступен с 16 июня 2021 года. https://oreil.ly/a1ERu.

Подходит для вашего масштаба

Большая часть работ в области наблюдаемости распределенных систем была вдохновлена работой, проделанной в крупномасштабных распределенных системах. К сожалению, это может привести к попытке воссоздать решения для систем гораздо большего масштаба, чем наши собственные, но без понимания компромиссов.

Крупномасштабным системам часто приходится идти на определенные уступки, ограничивая свою функциональность, чтобы справиться с масштабом, в котором они работают. Например, Dapper пришлось использовать очень агрессивную случайную выборку данных (фактически отбрасывая много информации), чтобы справиться с масштабом Google. Как выразился Бен Сигелман, основатель LightStep и создатель Dapper[1]:

> Микросервисы Google генерируют около 5 миллиардов RPC-вызовов в секунду. Таким образом, создание инструментов наблюдения, способных масштабироваться до 5 миллиардов RPC в секунду, сводится к созданию инструментов наблюдения, получающих крайне низкую функциональность. Если ваша организация делает больше 5 миллионов RPC-запросов в секунду, это все равно впечатляет, но вам почти наверняка не следует использовать то, что использует Google: при масштабе 1/1000 вы можете позволить себе гораздо более мощные функциональные возможности.

В идеале вам также необходим инструмент, *способный* масштабироваться по мере масштабирования системы. Опять же здесь играет роль экономическая эффективность. Даже если выбранный вами инструмент технически способен масштабироваться для поддержки ожидаемого роста вашей системы, можете ли вы позволить себе продолжать платить за него?

Эксперт в машине

В данной главе я много говорил об инструментах, возможно, больше, чем в любой другой. Отчасти это связано с фундаментальным переходом от взгляда на мир исключительно с точки зрения мониторинга к размышлениям о том, как сделать наши системы более наблюдаемыми. Такое изменение в поведении требует инструментов для его поддержки. Однако было бы ошибкой считать, что такое смещение связано исключительно с новыми инструментами. Тем не менее, учитывая большое количество различных поставщиков, претендующих на наше внимание, мы также должны быть осторожны.

Уже более десяти лет я наблюдаю, как некоторые поставщики заявляют об «умных» системах, способных волшебным образом обнаруживать проблемы и точно указывать нам, какие действия стоит предпринять. Кажется, появление

---

[1] *Sigelman B.* Three Pillars with Zero Answers — Towards a New Scorecard for Observability // Lightstep (пост в блоге). 5 декабря 2018 года. https://oreil.ly/R3LwC.

таких заявлений происходит волнами, но поскольку ажиотаж вокруг машинного обучения (МО) и искусственного интеллекта (ИИ) только усиливается, я вижу все больше предложений автоматического обнаружения отклонений. Я сомневаюсь в достаточной эффективности полностью автоматизированной версии такого ПО, и даже в этом случае было бы проблематично предположить, что все необходимые знания и опыт можно автоматизировать.

Большая часть усилий по созданию ИИ всегда была направлена на то, чтобы попытаться закодировать знания экспертов в автоматизированную систему. Идея, что мы можем автоматизировать экспертизу, может показаться кому-то привлекательной, но это также потенциально опасно, по крайней мере с нашим нынешним уровнем понимания. Зачем автоматизировать экспертизу? Чтобы потом не содержать кучу операторов-экспертов. Я не пытаюсь здесь что-то сказать о замещении рабочей силы, вызванном технологическим прогрессом. Скорее, люди продают эту идею, а компании покупают ее в надежде, что это полностью разрешимые (и автоматизируемые) проблемы. Реальность такова, что на данный момент это далеко от истины.

Недавно я работал с европейским стартапом, занимающимся наукой о данных. Стартап сотрудничал с компанией, поставлявшей оборудование для мониторинга медицинских коек, способное собирать различные данные о пациенте. Специалисты по обработке данных смогли помочь увидеть закономерности в собранных данных, показав необычные группы пациентов, определяемые путем сопоставления различных аспектов данных. Специалисты по обработке данных могли бы сказать: «Кажется, эти пациенты взаимосвязаны», но не понимали, в чем смысл этой связи. Потребовался врач, чтобы объяснить, что некоторые из этих групп относились к пациентам, которые в целом более больны, чем другие. Для выявления группы пациентов требовались одни знания, а для его расшифровки — совсем другие. Возвращаясь к нашим инструментам мониторинга и наблюдаемости, я мог бы представить подобный инструмент, предупреждающий кого-то, что «что-то выглядит странно», но понимание, что делать с этой информацией, по-прежнему требует определенного опыта.

Хотя я уверен, что такие функции, как автоматическое обнаружение отклонений, вполне могут продолжать совершенствоваться, необходимо признать, что прямо сейчас экспертом в системе является и еще какое-то время будет оставаться человек. Мы можем создавать инструменты, способные более качественно информировать оператора, что необходимо сделать, и автоматизировать их, чтобы помочь оператору более эффективно выполнять свои задачи. Но принципиально разнообразная и сложная среда распределенной системы означает, что без квалифицированных операторов-людей не обойтись. Наши эксперты должны использовать свой опыт, задавая правильные вопросы и принимая наилучшие из доступных решения. Не стоит просить их устранять недостатки плохих инструментов. Мы также не должны поддаваться иллюзиям, что какой-то модный новый инструмент решит все наши проблемы.

Приступая к работе

Как я уже говорил, здесь есть над чем подумать. Но я хочу предоставить базовую отправную точку для простой микросервисной архитектуры в плане того, что и как вы должны фиксировать.

Для начала вам нужна возможность собирать основную информацию о хостах, на которых работают ваши микросервисы: скорость процессора, I/O и т. д. Убедитесь, что вы можете сопоставить экземпляр микросервиса с хостом, на котором он запущен. Для каждого экземпляра микросервиса требуется фиксировать время отклика его сервисных интерфейсов и записывать все последующие вызовы в логи. Сразу внесите идентификаторы корреляции в свои логи. Регистрируйте другие важные этапы ваших бизнес-процессов. Это потребует от вас наличия хотя бы базовых метрик и цепочки инструментов для агрегации логов.

Я бы не решился сказать, что вам нужно начать со специального распределенного инструмента трассировки. Если вам придется запускать и размещать инструмент самостоятельно, это значительно усложнит задачу. С другой стороны, если вы можете легко воспользоваться полностью управляемым сервисом, оснащение микросервисов инструментами с самого начала может быть разумным вариантом.

Для ключевых операций я настоятельно рекомендую рассмотреть возможность создания синтетических транзакций, чтобы лучше понять, правильно ли работают жизненно важные аспекты вашей системы. Стройте свою систему с учетом данной возможности.

Все это лишь базовый сбор информации. Что еще более важно, необходимо убедиться, что вы можете просеять эту информацию, чтобы задать вопросы работающей системе. Можете ли вы с уверенностью сказать, что система должным образом работает для ваших пользователей? Со временем потребуется собирать очень много информации и совершенствовать свой инструментарий (и то, как вы его используете) для улучшения наблюдаемости вашей платформы.

Резюме

Распределенные системы сложны для понимания, и чем сильнее они распределены, тем труднее устранить неполадки в эксплуатации. Когда давление растет, всюду мелькают сигналы тревоги, а клиенты кричат на вас, важно иметь в распоряжении необходимую информацию, позволяющую понять, что происходит и как это исправить.

По мере усложнения архитектуры микросервиса становится нелегко предугадать, какие неполадки могут возникнуть. Вы столкнетесь с такими проблемами, которые сильно вас удивят. Поэтому важно переключить свое мышление с преимущественно пассивной деятельности по мониторингу на активное

формирование наблюдаемости системы. Это предполагает не только потенциальное изменение набора инструментов, но и переход от статических информационных панелей мониторинга к более динамичным действиям по детальному разбору.

Применение простой системы даст вам основы, которые помогут пройти долгий путь. Сперва создайте агрегацию логов, а также включите идентификаторы корреляции в строки логов. Распределенную трассировку можно добавить позже, но будьте начеку, когда придет время ввести ее в действие.

Сместите свое понимание работоспособности системы или микросервиса от бинарного состояния: «исправна» или «неисправна». Осознайте, что истина — всегда более тонкое понятие. Переключитесь с того, что каждая маленькая проблема вызывает тревогу, на более целостное мышление о том, что приемлемо. Настоятельно рекомендую рассмотреть возможность внедрения SLO и оповещения на основе этих принципов, чтобы снизить уровень усталости от оповещений и правильно сфокусировать внимание.

Прежде всего речь идет о признании, что нельзя знать все детали реализации до того, как вы выпустите продукт в свет. Научитесь справляться с неизвестностью.

Мы уже многое рассмотрели, но здесь есть в чем покопаться. Если вы хотите более подробно изучить концепции наблюдаемости, то я рекомендую книгу «Инженерия наблюдаемости» Чарити Мейджорс, Лиз Фонг-Джонс и Джорджа Миранды[1]. Я также рекомендую оба варианта «Site Reliability Engineering. Надежность и безотказность как в Google»[2] и «Site Reliability Workbook. Практическое применение»[3] в качестве хорошей отправной точки для более широкого изучения SLO, SLI и т. п. Стоит отметить, что эти две книги написаны по большей части с точки зрения того, как все делается (или было сделано) в Google, и это означает, что данные концепции не всегда будут применимы. Вероятно, вы не Google и у вас нет проблем размером с Google. Тем не менее в этих книгах все же есть стоящие рекомендации.

В следующей главе мы взглянем на наши системы с другой стороны и рассмотрим некоторые уникальные преимущества и проблемы, которые детализированная архитектура может создать в области безопасности.

---

[1] *Majors C., Fong-Jones L., Miranda G.* Observability Engineering. — O'Reilly, 2022.

[2] *Бейер Б., Джоунс К., Петофф Дж., Мерфи Н. Р.* Site Reliability Engineering. Надежность и безотказность как в Google. — Питер, 2021.

[3] *Бейер Б., Рензин Д., Мерфи Н. Р.* Site Reliability Workbook. Практическое применение. — Питер, 2021.

ГЛАВА 11

Безопасность

В предисловии к этой главе я хочу сказать, что не считаю себя экспертом в области безопасности приложений. Я стремлюсь просто быть *сознательно некомпетентным*. Другими словами, я хочу понять, чего я не знаю, и оценить свой текущий уровень подготовки. И каждый раз, когда я познаю что-то новое, я понимаю, сколько еще неизведанного впереди. Это не значит, что изучать подобные темы бессмысленно. Я чувствую, что все изученное в данной области за последние десять лет сделало меня более эффективным разработчиком и архитектором.

В текущей главе я выделяю те аспекты безопасности, которые, по моему мнению, стоит понять рядовому разработчику, архитектору или специалисту по эксплуатации, работающему над микросервисной архитектурой. Конечно, необходимость привлекать экспертов в области безопасности приложений никуда не делась, и даже если у вас есть такие люди, вам все равно необходимо немного разбираться в этой теме. Точно так же, как разработчики узнают больше о тестировании или управлении данными — темах, ранее доступных только определенным специалистам, общая осведомленность о вопросах безопасности может представлять жизненно важное значение для вашего ПО.

При сравнении микросервисов с менее распределенными архитектурами выясняется, что мы сталкиваемся с интересной дихотомией. С одной стороны, теперь у нас больше передаваемых по сетям данных, которые раньше оставались бы на одной машине, и инфраструктура, управляющая нашей архитектурой, стала сложнее — площадь поверхности атаки значительно повысилась. С другой стороны, микросервисы дают широкие возможности для глубокой обороны и ограничения объема доступа, потенциально увеличивая проекцию системы, а также снижая воздействие атаки, если она произойдет. Этот очевидный парадокс, что микросервисы могут сделать наши системы как менее, так и более безопасными, на самом деле представляет собой тонкий баланс. Я надеюсь, что к концу главы вы найдете золотую середину.

Чтобы помочь найти баланс, когда речь заходит о безопасности вашей микросервисной архитектуры, мы рассмотрим следующие темы.

Основные принципы

Фундаментальные концепции, которые полезно использовать при создании более безопасного ПО.

Пять функций кибербезопасности

Идентификация, защита, обнаружение, реагирование и восстановление — обзор пяти ключевых функциональных областей безопасности приложений.

Основы безопасности приложений

Некоторые особые фундаментальные концепции безопасности приложений и их применение к микросервисам, включая учетные данные и секреты, исправления, резервные копии и повторную сборку.

Безусловное доверие в сравнении с нулевым доверием

Различные подходы к обеспечению доверия в микросервисной среде, и как это влияет на деятельность, связанную с безопасностью.

Защита данных

Как мы защищаем данные при их перемещении по сетям и хранении на диске.

Аутентификация и авторизация

Как работает технология единого входа (single sign-on, SSO) в микросервисной архитектуре, централизованные и децентрализованные модели авторизации и роль JWT-токенов как составляющей этого процесса.

Основные принципы

Часто, когда речь заходит о безопасности микросервисов, люди пытаются начать разговор о достаточно сложных технологических вопросах, таких как использование JWT-токенов или необходимость внедрения двухстороннего протокола TLS (эти темы мы рассмотрим чуть позже). Однако проблема с безопасностью заключается в том, что вы защищены настолько, насколько сильна защита вашего наиболее слабого звена. Используем аналогию. При желании обезопасить свой дом было бы ошибкой сосредоточить все свои усилия на приобретении устойчивой к взлому входной двери с подсветкой и камерами для отпугивания злоумышленников, но при этом оставив черный ход открытым.

Таким образом, существуют фундаментальные аспекты безопасности приложений, на которые стоит обратить внимание хотя бы поверхностно, чтобы выделить множество важных проблем. Мы рассмотрим, как эти основные проблемы становятся более (или менее) сложными в контексте микросервисов, но они также должны быть применимы к разработке ПО в целом. Для тех из вас, кто хочет сразу перейти ко всем этим «хорошим вещам», пожалуйста, просто убедитесь, что вы не слишком зацикливаетесь на защите парадного входа, оставляя дверь черного хода открытой.

Принцип наименьших привилегий

Предоставляя доступ к приложениям отдельным лицам, внешним или внутренним системам или даже нашим собственным микросервисам, необходимо ясно понимать, что это за доступ. Принцип наименьших привилегий описывает идею о том, что стоит предоставлять минимальный доступ, необходимый стороне для выполнения требуемой функциональности, и только на конкретный период времени. Основное преимущество такого подхода в том, что, если учетные данные будут скомпрометированы взломщиком, они дадут злоумышленнику максимально ограниченный доступ.

Если у микросервиса есть доступ к БД только для чтения, то злоумышленник, обошедший защиту, получит доступ только для чтения и только к определенной БД. Если срок действия учетных данных для базы данных истекает до того, как они будут скомпрометированы, то украденные учетные данные становятся бесполезными. Эту концепцию можно расширить, ограничив перечень микросервисов для взаимодействия с конкретными сторонами.

Чуть позже мы увидим, что доступ может предоставляться и на определенный период времени, что еще больше ограничивает любые неблагоприятные последствия, которые могут возникнуть в случае компрометации.

Глубокая оборона

В Великобритании, где я живу, очень много замков. Эти сооружения являются памятником истории нашей страны (не в последнюю очередь напоминанием о времени до того, как Соединенное Королевство было, ну, в какой-то степени, объединено). Они напоминают нам о том времени, когда люди чувствовали необходимость защищать свою собственность от врагов. Иногда предполагаемые враги различались — многие замки недалеко от того места в Кенте, где я живу, были спроектированы для защиты от прибрежных вторжений со стороны Франции[1]. Какова бы ни была причина, замки могут быть отличным примером принципа глубокой обороны.

Наличие только одного механизма защиты станет проблемой, если злоумышленник найдет способ взломать эту защиту или если механизм защищает только от определенных типов злоумышленников. Представьте себе форт береговой обороны, единственная стена которого обращена к морю, что делает его полностью беззащитным перед нападением с суши. Если посмотреть на Дуврский замок, то можно увидеть множество защитных сооружений. Во-первых, он находится на большом холме, что затрудняет подход к замку по суше. У него не одна стена, а две — брешь в первой все равно требует, чтобы нападающий справился со второй. И после преодоления последней стены придется иметь дело с большой и внушительной крепостью (башней).

[1] Пожалуйста, давайте не будем все сводить к Brexit.

Тот же принцип должен применяться, когда мы встраиваем защиту в свои приложения. Наличие множества средств для защиты от злоумышленников имеет жизненно важное значение. Благодаря микросервисным архитектурам появилось гораздо больше мест, где можно укрепить свои системы. Разбивая функциональность на различные микросервисы и ограничивая объем их задач, мы уже применяем глубокую оборону. Также можно запускать микросервисы в разных сегментах сети, применяя сетевые средства защиты в большем количестве мест, и даже использовать различные технологии для создания и запуска этих микросервисов таким образом, чтобы одна уязвимость нулевого дня не повлияла на все, что у нас есть.

Микросервисы обеспечивают более глубокую оборону, чем аналогичные однопроцессные монолиты, и в результате они способствуют созданию более безопасных систем в организациях.

СРЕДСТВА КОНТРОЛЯ БЕЗОПАСНОСТИ

Рассматривая типы средств контроля безопасности, которые реально внедрить для защиты системы, можно классифицировать их следующим образом[1].

- *Превентивный*

  Предотвращение нападения. Перечень таких средств включает в себя безопасное хранение секретов, шифрование данных в состоянии покоя и при передаче, а также внедрение надлежащих механизмов аутентификации и авторизации.

- *Детективный*

  Оповещает вас о том, что атака происходит/уже произошла. Хорошими примерами могут служить брандмауэры приложений и службы обнаружения вторжений.

- *Реагирующий*

  Помогает вам реагировать во время/после атаки. Наличие автоматизированного механизма повторной сборки вашей системы, рабочих резервных копий для восстановления данных и надлежащего плана связи на случай инцидента может иметь жизненно важное значение.

Для надлежащей защиты системы потребуется комбинация всех трех типов контроля, причем мест применения каждого из них может быть несколько. Возвращаясь к примеру с замком, у нас может быть несколько стен, представляющих собой превентивные средства контроля. Мы могли бы установить сторожевые башни и систему маяков, чтобы получить возможность видеть, происходит ли атака. Наконец, можно держать наготове несколько плотников и каменщиков на случай, если понадобится укрепить двери или стены после нападения. Очевидно, что вы вряд ли будете зарабатывать на жизнь строительством замков, поэтому далее мы рассмотрим примеры этих элементов управления в контексте архитектуры микросервисов.

[1] Как я ни старался, не смог найти первоисточник для этой схемы категоризации.

Автоматизация

Постоянно повторяющаяся тема книги — автоматизация. Поскольку в микросервисных архитектурах гораздо больше подвижных элементов, автоматизация становится ключевым фактором, помогающим управлять растущей сложностью системы. В то же время мы стремимся увеличить скорость доставки, и здесь автоматизация играет важную роль. Компьютеры намного быстрее и эффективнее (и с меньшей вариативностью) выполняют рутинные задачи, чем люди. Они также способствуют уменьшению количества человеческих ошибок и упрощают реализацию принципа наименьших привилегий. Например, можно назначить определенные привилегии конкретным скриптам.

Как мы увидим в этой главе, автоматизация способна помочь восстановиться после инцидента. Мы можем использовать ее для отзыва и ротации ключей безопасности, а также применять инструменты, помогающие легче обнаруживать потенциальные проблемы безопасности. Как и в случае с другими аспектами архитектуры микросервисов, внедрение культуры автоматизации существенно поможет в обеспечении безопасности.

Встраивание безопасности в процесс доставки

Как и многие другие аспекты доставки программного обеспечения, безопасность слишком часто рассматривается как нечто второстепенное. Исторически сложилось, что с обеспечением безопасности системы разбираются уже после написания кода, что приводит к новым доработкам. Безопасность часто рассматривалась как нечто вроде препятствия для выпуска ПО на рынок.

За последние 20 лет мы наблюдали аналогичные проблемы с тестированием, удобством использования (юзабилити) и эксплуатацией. Эти аспекты доставки программного обеспечения часто выполнялись изолированно и после того, как основная часть кода была завершена. Мой бывший коллега Джонни Шнайдер сравнивает подход к юзабилити ПО с возникновением запоздалой мысли, которую добавляют поверх «основного блюда»[1].

Реальность, конечно, такова, что непригодное для использования, небезопасное, не способное должным образом работать в эксплуатации, изобилующее ошибками ПО не может быть «основным блюдом» — это в лучшем случае ущербное предложение. Мы стали интенсивнее прибегать к тестированию в основном процессе доставки, как это было сделано с операционными аспектами (DevOps о чем-нибудь говорит?) и удобством использования — ситуация с безопасностью не должна отличаться. Нам нужно обеспечить разработчиков

---

[1] Рекомендую книгу Джонни Understanding Design Thinking, Lean, and Agile (O'Reilly) для получения дополнительной информации.

более общим представлением о проблемах, связанных с безопасностью, чтобы специалисты находили способ встраиваться в команды доставки, когда это необходимо, и чтобы инструментарий улучшался, позволяя внедрять в наше ПО мышление, связанное с безопасностью.

Это создаст проблемы в организациях, внедряющих команды, ориентированные на потоки, с повышенной степенью автономии в отношении владения своими микросервисами. Какова роль экспертов по безопасности? В разделе «Команды поддержки» главы 15 мы рассмотрим, как эксперты по безопасности могут поддерживать потоковые команды и помогать владельцам микросервисов внедрять больше идей в области безопасности в свое ПО, а также убедиться, что у вас в нужный момент есть под рукой все необходимое.

Существуют автоматизированные инструменты, способные проверять системы на наличие уязвимостей, например, путем поиска межсайтовых скриптовых атак. Хорошим примером будет сканер Zed Attack Proxy (ZAP). Основываясь на работе OWASP, приложение ZAP пытается воссоздать вредоносные атаки на вашем веб-сайте. Есть и другие инструменты, которые используют статический анализ для поиска распространенных ошибок в программировании и которые могут открыть бреши в безопасности, например Brakeman (https://brakemanscanner.org) для Ruby. Такой инструмент, как Snyk (https://snyk.io), может обнаруживать зависимости от сторонних библиотек, имеющих известные уязвимости. Если эти инструменты можно легко интегрировать в обычные сборки CI, то интеграция их в ваши стандартные проверки — отличное начало. Конечно, стоит отметить, что многие из этих типов инструментов могут решать только локальные проблемы, например уязвимость в определенном фрагменте кода. Они не заменяют необходимость понимания безопасности вашей системы на более широком, системном уровне.

Пять функций кибербезопасности

Имея в виду эти основные принципы, давайте теперь рассмотрим широкий спектр мероприятий, связанных с безопасностью, которые нам необходимо выполнить. Затем мы сможем понять, как эти действия меняются в контексте микросервисной архитектуры. Модель, которую я предпочитаю для описания безопасности приложений, взята из Национального института стандартов и технологий США (National Institute of Standards and Technology, NIST), в котором описана полезная модель из пяти частей (https://oreil.ly/MSAuU) для различных видов деятельности, связанных с кибербезопасностью.

- *Выявите* потенциальных злоумышленников, каких целей они пытаются достичь и где вы наиболее уязвимы.
- *Защитите* свои ключевые активы от потенциальных хакеров.

- *Определите*, произошла ли атака, несмотря на все ваши усилия.
- *Реагируйте*, когда узнаете, что произошло что-то плохое.
- *Восстанавливайте* работоспособность после инцидента.

Я нахожу эту модель особенно полезной из-за ее целостного характера. Очень легко направить все усилия на защиту своего приложения, не задумываясь о том, с какими угрозами вы можете столкнуться на самом деле, не говоря уже о том, чтобы решить, что вы можете сделать, если умный злоумышленник проскользнет мимо вашей защиты.

Давайте подробнее рассмотрим каждую из этих функций и посмотрим, как микросервисная архитектура может изменить подход к этим идеям по сравнению с более традиционной монолитной архитектурой.

Выявление

Прежде чем мы сможем решить, что следует защищать, необходимо выяснить, кто может охотиться за нашими материалами и что именно они будут искать. Часто бывает трудно представить себя в роли злоумышленника, но это именно то, что следует сделать, чтобы сосредоточить свои усилия в нужном месте. Моделирование угроз — это первое, на что стоит обратить внимание при рассмотрении данного аспекта безопасности приложений.

Будучи людьми, мы довольно плохо понимаем риски. Мы часто зацикливаемся на неправильных вещах, игнорируя более серьезные проблемы, которые могут быть просто вне поля зрения. Это, конечно, распространяется и на сферу безопасности. Наше понимание, каким рискам безопасности мы можем подвергнуться, часто в значительной степени обусловлено нашим ограниченным представлением о системе, нашими навыками и опытом.

Когда я общаюсь с разработчиками о рисках безопасности в контексте микросервисной архитектуры, они сразу же начинают говорить о JWT-токенах и двухстороннем протоколе TLS. Они ищут технические решения технических проблем, о которых имеют некоторое представление. Я не хочу указывать пальцем только на разработчиков — у всех нас ограниченный взгляд на мир. Возвращаясь к аналогии, которую мы использовали ранее, вот как можно получить невероятно защищенный парадный вход и распахнутую настежь дверь черного хода.

Я работал в одной организации, где неоднократно обсуждалась необходимость установки камер видеонаблюдения в приемных компании по всему миру. Это произошло из-за инцидента, в ходе которого неуполномоченное лицо получило доступ в офисную зону, а затем и в корпоративную сеть. Считалось, что система камер видеонаблюдения не только удержит других от подобных попыток, но и поможет выявить причастных постфактум.

Призрак корпоративной слежки вызвал в фирме волну беспокойства по поводу проблем типа «большого брата». Все свелось к тому, что голосующие

разделились на два лагеря: если вы за камеры, значит, поддерживаете слежку за сотрудниками, если против — вы на стороне злоумышленников и хотите зла компании. Оставляя в стороне проблему групповой поляризации[1], один сотрудник высказался в довольно застенчивой манере, предположив, что, возможно, дискуссия была немного ошибочной, поскольку мы упустили некоторые более важные вопросы. Он указал на тот факт, что никто не обеспокоился, что у входной двери одного из главных офисов был неисправный замок и что в течение многих лет люди приходили утром и обнаруживали, что дверь незаперта.

Эта экстремальная (но правдивая) история — отличный пример общей проблемы, с которой мы сталкиваемся, пытаясь обезопасить свои системы. Не имея времени принять во внимание все факторы и понять, в чем заключаются самые большие риски, вы вполне можете пропустить места, которым стоило бы уделить больше времени. Цель моделирования угроз — помочь вам понять, чего злоумышленники могут хотеть от вашей системы. Чего они добиваются? Им всем нужно одно и то же или каждый преследует свою цель? Моделирование угроз, если все сделано правильно, в значительной степени сводится к тому, чтобы поставить себя на место злоумышленника, думать как он. Такой взгляд со стороны важен, и это одна из причин, по которой помощь внешней стороны в проведении моделирования угроз может быть очень полезной.

Основная идея моделирования угроз не сильно меняется, когда мы смотрим на микросервисные архитектуры, за исключением того, что любая анализируемая архитектура теперь может быть более сложной. Меняется то, как мы воспринимаем результаты модели угроз и воплощаем их в жизнь. Одним из результатов моделирования угроз может стать список рекомендаций, какие средства контроля безопасности необходимо внедрить. Эти средства контроля могут включать такие вещи, как изменение процесса, технологии или вообще архитектуры системы. Некоторые из этих преобразований могут быть сквозными и способны повлиять на несколько групп и связанные с ними микросервисы, другие же — привести к более целенаправленной работе. Однако, по сути, при моделировании угроз необходимо смотреть целостно: сосредоточение этого анализа на слишком малом подмножестве вашей системы, таком как один или два микросервиса, приведет к ложному ощущению безопасности. Вероятно, в итоге вы потратите время на создание фантастически безопасной входной двери только для того, чтобы оставить окно открытым.

Для более глубокого погружения в эту тему я могу порекомендовать книгу «Моделирование угроз: Проектирование для обеспечения безопасности»[2] Адама Шостака.

---

[1] Это был скорее пассивно-агрессивный спор, часто без «пассивной» части.

[2] *Shostack A.* Threat Modeling: Designing for Security. — Wiley, 2014.

Защита

Как только мы определим самые ценные и наиболее уязвимые активы, необходимо убедиться, что они должным образом защищены. Как я уже отмечал, микросервисные архитектуры, возможно, предоставляют гораздо более широкую область для хакерской атаки, и поэтому у нас возрастает и количество элементов, требующих защиты. Но микросервисы также дают множество возможностей для углубленной обороны. Значительную часть данной главы мы посвятим различным аспектам защиты, в первую очередь потому, что именно в этой области микросервисные архитектуры создают больше всего проблем.

Определение

При использовании микросервисной архитектуры заподозрить неладное может быть гораздо сложнее. Здесь у нас больше сетей для мониторинга и больше машин, за которыми необходимо следить. Значительно увеличивается количество источников информации, что еще сильнее затрудняет выявление проблем. Многие методы, рассмотренные в главе 10, такие как агрегирование логов, помогут собрать информацию, способствующую обнаружению потенциальных опасностей. В дополнение к ним существуют специальные инструменты, такие как системы обнаружения проникновений. Их можно запустить, чтобы вскрыть вредоносное поведение. Программное обеспечение, позволяющее справляться с растущей сложностью систем, совершенствуется, особенно в области контейнерных рабочих нагрузок с развитием таких инструментов, как Aqua (https://oreil.ly/OQn0O).

Реакция

Если случилось худшее и вы узнали об этом, что вам следует сделать? Выработка эффективного подхода к реагированию на инциденты жизненно важна для ограничения ущерба, причиняемого нарушением безопасности. Обычно формирование такого подхода начинается с понимания масштабов нарушения и того, какие данные были раскрыты. Если раскрытые данные включают личную информацию (personally identifiable information, PII), то вам необходимо следовать как процедурам реагирования на инциденты в области безопасности и конфиденциальности, так и процессам уведомления. Это означает, что вам придется общаться с различными подразделениями организации и в некоторых ситуациях вы можете быть юридически обязаны информировать назначенного сотрудника, отвечающего за защиту данных, о возникновении определенных типов нарушений.

Многие организации усугубили последствия нарушения из-за неправильного обращения с последствиями, что часто приводило к увеличению финансовых штрафов, помимо ущерба, нанесенного их бренду и отношениям с клиентами. Поэтому важно понимать, что вы должны делать по юридическим или

нормативным причинам и что сделать с точки зрения заботы о пользователях вашего ПО. Например, GDPR требует, чтобы о нарушениях персональных данных сообщалось соответствующим органам в течение 72 часов — срока, который не кажется чрезмерно обременительным. Это не означает, что не стоит стремиться к тому, чтобы люди раньше узнавали об утечке данных.

Помимо внешних коммуникационных аспектов реагирования, решающее значение имеет то, как вы справляетесь с внутренними проблемами. Организациям, в которых существует культура взаимных обвинений и поиска виновных, скорее всего, придется плохо после крупного инцидента. Уроки не будут заучены, и способствующие проблеме факторы не будут выявлены. С другой стороны, организация, уделяющая особое внимание открытости и безопасности, получает наилучшие возможности для усвоения уроков, гарантирующих, что подобные инциденты будут происходить с меньшей вероятностью. Мы вернемся к этому в разделе «Поиск виновных» главы 12.

Восстановление

Восстановление относится к способности снова запустить систему после атаки, а также применить полученный опыт, чтобы снизить вероятность повторения проблем. При использовании микросервисной архитектуры у нас гораздо больше подвижных элементов, что может усложнить восстановление, если инцидент вызвал масштабные последствия. Поэтому вскоре мы рассмотрим, как автоматизация и резервное копирование помогут вам выполнить повторную сборку микросервисной системы по требованию и вернуть ее в рабочее состояние как можно быстрее.

Основы безопасности приложений

Итак, теперь мы усвоили несколько основных принципов и получили некоторое представление о широком мире, который может охватывать деятельность в области безопасности. Поговорим об основополагающих темах безопасности в контексте микросервисной архитектуры: учетных данных, исправлениях, резервном копировании и повторной сборке.

Учетные данные

Условно говоря, учетные данные предоставляют человеку (или компьютеру) доступ к какому-либо ограниченному ресурсу. Это может быть база данных, компьютер, учетная запись пользователя или что-то еще. При использовании микросервисной архитектуры, в сравнении с эквивалентной монолитной архитектурой, у нас, вероятно, задействовано такое же количество людей, но гораздо больше учетных данных, представляющих различные микросервисы,

(виртуальные) машины, базы данных и т. п. Это может привести к некоторой путанице в отношении того, как ограничить (или не ограничивать) доступ, и во многих случаях спровоцирует «ленивый» подход, при котором используется небольшое количество учетных данных с широкими привилегиями в попытке упростить ситуацию. Это, в свою очередь, приводит к еще большим проблемам, если учетные данные будут скомпрометированы.

Мы можем разбить тему учетных данных на две ключевые области. Первая — учетные данные пользователей (и операторов) нашей системы. Они часто становятся самым слабым местом системы и часто используются злоумышленниками в качестве вектора атаки. Вторая область — секреты. Это фрагменты информации, имеющие решающее значение для запуска микросервисов. Для обоих наборов учетных данных мы должны рассмотреть вопросы ротации, аннулирования и ограничения сферы действия.

Учетные данные пользователя

Учетные данные пользователей, такие как электронная почта и пароли, по-прежнему важны для работы с нашим ПО, но они также представляют собой потенциальное слабое место, когда речь идет о доступе злоумышленников к системам. Отчет по результатам расследований утечки данных за 2020 год (https://oreil.ly/hqXfM) компании Verizon показал, что в 80 % случаев хакерских атак использовалась та или иная форма кражи учетных данных. Сюда входят ситуации, когда учетные данные были украдены с помощью таких механизмов, как фишинговые атаки или перебор паролей.

Существует несколько отличных советов (которые, несмотря на свою простоту и понятность, все еще недостаточно широко применяются), как правильно обращаться с паролями, например. У Троя Ханта есть отличный обзор последних рекомендаций как от NIST, так и от Национального центра кибербезопасности Великобритании[1]. Эти советы включают рекомендации использовать менеджеры паролей и длинные пароли, избегать использования сложных правил паролей и — что несколько удивительно — избегать обязательной регулярной смены пароля. Полную публикацию Троя стоит прочитать подробно.

В нынешнюю эпоху систем, управляемых API, учетные данные также распространяются на управление API-ключами для сторонних систем, например учетные записи облачного провайдера. Если злоумышленник получит доступ к вашей корневой учетной записи AWS, он сможет уничтожить все, что работает под этой учетной записью. В одном экстремальном примере такая атака привела к тому, что компания Code Spaces[2] прекратила свою деятельность, так как

[1] *Hunt T.* Passwords Evolved: Authentication Guidance for the Modern Era. 26 июля 2017 года. https://oreil.ly/T7PYM.

[2] *McAllister N.* Code Spaces Goes Titsup FOREVER After Attacker NUKES Its Amazon-Hosted Data // The Register. 18 июня 2014 года. https://oreil.ly/mw7PC.

все их ресурсы, резервные копии и т. д. работали под одной учетной записью. Иронично, что Code Spaces предлагали «надежный, безопасный и доступный Svn-хостинг, Git-хостинг и управление проектами».

Даже если кто-то завладеет вашими API-ключами для облачного провайдера и решит не уничтожить все, что вы создали, он может запустить несколько дорогостоящих виртуальных машин для майнинга биткоинов в надежде, что вы этого не заметите. Это случилось с одним из моих клиентов, который обнаружил, что кто-то потратил более 10 тысяч долларов, делая именно так, прежде чем учетная запись была отключена.

Оказывается, злоумышленники тоже умеют автоматизировать. Существуют боты, которые просто сканируют учетные данные и пытаются использовать их для запуска машин для майнинга криптовалют.

Секреты

В широком смысле секреты — это критически важные фрагменты информации, необходимые микросервису для работы, а также достаточно конфиденциальные, чтобы требовалось защищать их от злоумышленников. Примеры секретов, которые могут понадобиться микросервису, включают:

- сертификаты для TLS;
- SSH-ключи;
- пары открытых/закрытых API-ключей;
- учетные данные для доступа к базам данных.

При рассмотрении жизненного цикла секрета можно выделить различные аспекты управления секретами, для которых могут понадобиться определенные требования безопасности.

Создание

Как изначально создается секрет?

Распределение

Как убедиться, что только что созданный секрет попадет в нужное место (и только в нужное место)?

Хранение

Хранится ли секрет так, чтобы только авторизованные стороны могли получить к нему доступ?

Мониторинг

Знаем ли мы, как используется этот секрет?

Ротация

Можем ли мы изменить секрет, не вызывая проблем?

Если есть несколько микросервисов, каждому из которых могут потребоваться разные наборы секретов, нам понадобится использовать инструменты управления секретами.

Kubernetes предоставляет встроенное решение для работы с секретами. Оно несколько ограничено с точки зрения функциональности, но доступно «из коробки», поэтому его может быть достаточно для многих вариантов использования[1].

Если вы искали более сложный инструмент, стоит рассмотреть Vault (https://www.vaultproject.io) от Hashicorp. ПО с открытым исходным кодом и доступными коммерческими опциями — это настоящий швейцарский армейский нож для управления секретами, который обрабатывает все, начиная с основных аспектов распространения секретов и заканчивая созданием ограниченных по времени учетных данных для облачных платформ и БД. У Vault есть дополнительное преимущество — инструмент consul-template (https://oreil.ly/qNmAZ), способный динамически обновлять секреты в обычном файле конфигурации. Это означает, что части вашей системы, желающие считывать секреты из локальной файловой системы, не нужно изменять для поддержки инструмента управления секретами. Когда секрет изменяется в Vault, инструмент consul-template обновит соответствующую запись в файле конфигурации, позволяя вашим микросервисам динамически изменять используемые ими секреты. Это потрясающе для масштабного управления учетными данными.

Некоторые поставщики публичных облачных сервисов также предлагают решения в данной области. На ум приходят Secrets Manager от AWS (https://oreil.ly/cuwRX) или Key Vault (https://oreil.ly/rV3Sb) от Azure. Однако некоторым людям в принципе не нравится идея хранения критически важной секретной информации в общедоступном облачном сервисе. Все зависит от вашей модели угроз. Если это вызывает серьезную озабоченность, ничто не мешает вам запустить Vault на выбранном вами облачном провайдере и самостоятельно управлять системой. Даже если данные в состоянии покоя хранятся у провайдера, с помощью соответствующего серверного хранилища вы сможете обеспечить шифрование данных таким образом, что, даже если сторонний пользователь завладеет ими, он не сможет ничего с ними сделать.

Ротация

В идеале нужно часто менять учетные данные, чтобы ограничить вероятный ущерб в случае получения несанкционированного доступа к этим данным. Если злоумышленник получает доступ к вашей паре открытых/закрытых AWS

[1] Некоторые люди обеспокоены, что секреты хранятся в виде обычного текста. Проблема это для вас или нет, во многом зависит от вашей модели угроз. Чтобы секреты были прочитаны, злоумышленник должен получить прямой доступ к основным системам, на которых работает ваш кластер, и в этот момент можно утверждать, что кластер уже безнадежно скомпрометирован.

API-ключей, но эти учетные данные меняются раз в неделю, у него есть только одна неделя, чтобы использовать учетные данные. Конечно, за неделю можно нанести внушительный ущерб, но вы поняли идею. Некоторые злоумышленники, получая доступ к системам, стараются оставаться незамеченными, чтобы со временем собирать более ценные данные и находить пути проникновения в другие части вашей системы. Если украденные учетные данные предполагается использовать для получения доступа, то ограничение срока действия этих учетных данных поможет защитить вашу систему, так как хакеры могут просто не успеть воспользоваться ими.

Отличным примером ротации учетных данных оператора может быть генерация ограниченных по времени API-ключей для использования AWS. Многие организации в настоящее время генерируют API-ключи на лету для своих сотрудников, причем открытая и закрытая пары ключей действительны только в течение короткого периода времени — обычно менее часа. Это позволяет формировать API-ключи, необходимые для выполнения любой операции, и вы будете чувствовать себя в безопасности, зная, что, даже если злоумышленник впоследствии получит доступ к данным ключам, он не сможет ими воспользоваться. Даже если вы случайно засветили эту пару ключей в общедоступном GitHub, от нее не будет толку, как только срок ее действия истечет.

Использование учетных данных, ограниченных по времени, может быть полезно и для систем. Инструмент Vault от Hashicorp способен генерировать ограниченные по времени учетные данные для баз данных. Вместо того чтобы ваш экземпляр микросервиса считывал сведения о подключении к БД из хранилища конфигурации или текстового файла, эти сведения могут быть сформированы на лету для конкретного экземпляра вашего микросервиса.

Переход к процессу частой ротации учетных данных, таких как ключи, может быть болезненным. Я общался с представителями компаний, которые сталкивались с инцидентами в результате ротации ключей, когда системы переставали работать при смене ключей. Часто это происходит из-за того, что может быть неясно, для чего используются те или иные учетные данные. Если область применения учетных данных ограничена, потенциальное влияние ротации на ваш рабочий процесс значительно снижается. Но если учетные данные широко используются в системе, определить влияние изменений становится непросто. Я рассказываю это не для того, чтобы отговорить вас от ротации, а для того, чтобы вы знали о потенциальных рисках, и я по-прежнему убежден, что это правильно. Наиболее разумным шагом вперед, вероятно, было бы внедрение инструментов, помогающих автоматизировать данный процесс, одновременно ограничивая область действия каждого набора учетных данных.

Аннулирование

Наличие политики, гарантирующей регулярную смену ключевых учетных данных, может быть разумным способом ограничить последствия их утечки, но

что произойдет, если вы *знаете*, что определенные данные попали не в те руки? Нужно ли вам ждать, пока начнется запланированная ротация? Это может быть непрактично или неразумно. Вместо этого в идеале необходимо иметь возможность автоматически отзывать и, возможно, восстанавливать учетные данные, когда происходит нечто подобное.

Здесь поможет использование инструментов, позволяющих централизованно управлять секретами, но для этого потребуется повторное считывание микросервисами вновь созданных значений. Если микросервис напрямую считывает секреты из чего-то вроде хранилища секретов Kubernetes или Vault, он может получать уведомления об изменении этих значений, что позволяет ему использовать измененные значения. В качестве альтернативы, если микросервис считывает эти секреты только при запуске, вам понадобится выполнить перезапуск вашей системы на ходу, чтобы перезагрузить эти учетные данные. Если вы регулярно меняете учетные данные, то, скорее всего, уже сталкивались с проблемой повторного считывания этих данных микросервисами. Если вас устраивает регулярная ротация учетных данных, вероятно, ваша система уже настроена на обработку экстренного отзыва.

СКАНИРОВАНИЕ В ПОИСКАХ КЛЮЧЕЙ

Случайная проверка закрытых ключей в репозиториях исходного кода — это распространенный вариант утечки учетных данных. Такое случается на удивление часто. GitHub автоматически сканирует репозитории на наличие некоторых типов секретов, но вы можете запустить и собственное сканирование. Было бы здорово, если бы была возможность получить секреты перед регистрацией, что и позволяет сделать инструмент git-secrets (https://oreil.ly/Ra9ii). Он может сканировать существующие коммиты на предмет потенциальных секретов. Но при настройке его в качестве хука коммитов он способен остановить даже их создание. Существует также аналогичный инструмент gitleaks (https://oreil.ly/z8xrf), который обладает несколькими дополнительными функциями, превращающими его в потенциально более полезный общий инструмент для сканирования локальных файлов.

Ограничение области действия

Ограничение области действия учетных данных лежит в основе идеи принятия принципа наименьших привилегий. Это может применяться ко всем формам учетных данных, но ограничение области того, к чему данный набор учетных данных дает вам доступ, может быть невероятно полезным. Например, на рис. 11.1 каждому экземпляру микросервиса `Запасы` присваиваются одно и то же имя пользователя и пароль для поддерживающей БД. Мы также предоставляем доступ только для чтения вспомогательному процессу Debezium (https://debezium.io), который используется для считывания данных и отправки их через Kafka как часть существующего процесса ETL. Если имя пользователя и пароль для микросервисов будут скомпрометированы, сторонний пользо-

ватель теоретически может получить доступ на чтение и запись БД. Однако, если бы злоумышленники получили доступ к ресурсам сервиса Debezium, у них был бы доступ только для чтения.

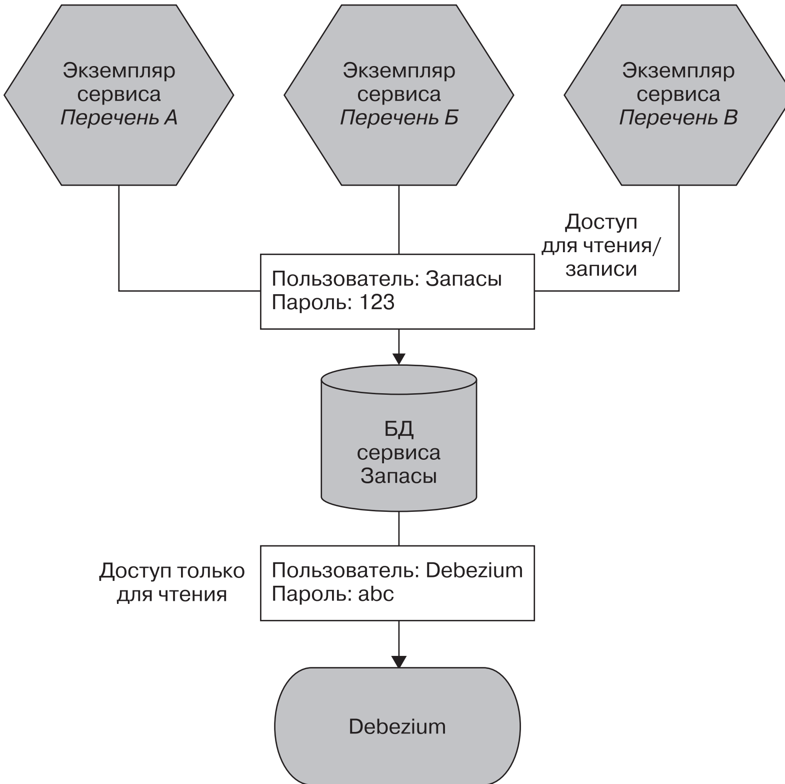


Рис. 11.1. Ограничение области действия учетных данных для минимизации последствий неправильного использования

Ограничение области действия может применяться в отношении как того, к чему может получить доступ набор учетных данных, так и того, кто имеет доступ к этому набору учетных данных. На рис. 11.2 настройки изменены таким образом, чтобы каждый экземпляр сервиса `Запасы` получал свой набор учетных данных. Это означает, что мы могли бы чередовать все учетные данные независимо или просто отозвать их для одного из экземпляров, если они оказались скомпрометированными. Кроме того, с более конкретными учетными данными может быть легче выяснить, откуда и как они были получены. Очевидно, что наличие здесь уникального идентифицируемого имени пользователя для экземпляра микросервиса дает и другие преимущества. Например, может быть проще отследить, какой экземпляр вызвал дорогостоящий запрос.

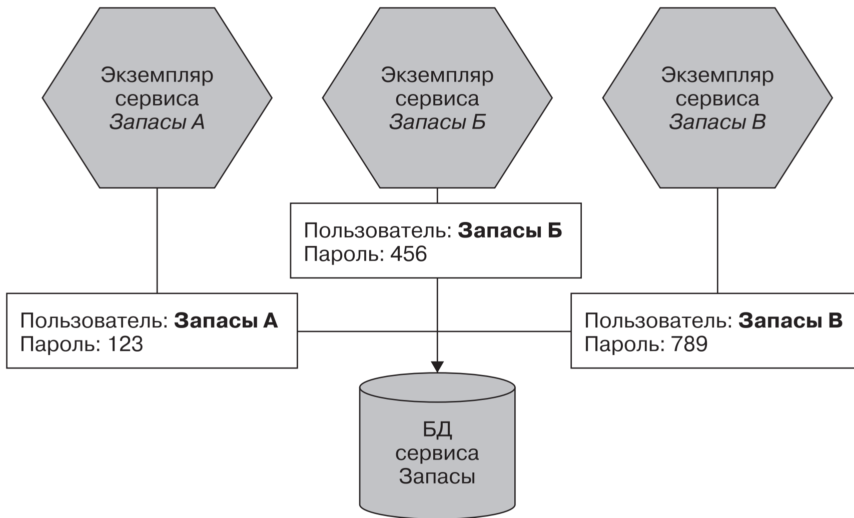


Рис. 11.2. У каждого экземпляра сервиса Запасы свои собственные учетные данные для доступа к БД, что еще больше ограничивает доступ

Как мы уже говорили, управление большим количеством более мелких учетных данных в крупномасштабной системе может быть сложным, и если вы действительно хотите применить подобный подход, необходима некоторая форма автоматизации — такие хранилища секретов, как Vault, приходят на ум как идеальные способы реализации подобных схем.

Исправление

Утечка данных Equifax в 2017 году — отличный пример важности исправления. Известная уязвимость в Apache Struts была использована для получения несанкционированного доступа к данным, хранящимся в Equifax. Поскольку Equifax — это кредитное бюро, та информация была особенно конфиденциальной. В конце концов установили, что в результате взлома были скомпрометированы данные более 160 миллионов человек. В итоге Equifax пришлось выплатить компенсацию в размере 700 миллионов долларов.

За несколько месяцев до взлома была обнаружена уязвимость в Apache Struts, и разработчики выпустили патч, устраняющий проблему. К сожалению, в Equifax не обновились до новой версии ПО, несмотря на то что оно было доступно за несколько месяцев до атаки. Если бы в Equifax своевременно обновили это программное обеспечение, вероятно, взлом был бы невозможен.

Постоянные обновления становятся непростой задачей по мере внедрения все более сложных систем. Нам следует стать изощреннее в обращении с этой довольно простой концепцией.

На рис. 11.3 показан пример уровней инфраструктуры и ПО, которые существуют под типичным кластером Kubernetes. Если вы сами управляете всей этой инфраструктурой, вы отвечаете за управление и исправление всех уровней. Насколько вы уверены в актуальности своих исправлений? Очевидно, что, если вы можете переложить часть данной работы на поставщик общедоступных облачных вычислений, вы также сможете снять с себя часть этого бремени.

Рис. 11.3. Различные уровни современной инфраструктуры, которые требуют технического обслуживания и исправления

Например, если бы вы использовали управляемый кластер Kubernetes на одном из основных провайдеров общедоступных облачных сервисов, вы бы резко сократили объем владения, как показано на рис. 11.4.

Контейнеры подбрасывают нам интересный сюрприз. Мы рассматриваем данный экземпляр контейнера как неизменяемый. Но контейнер содержит не только наше ПО, но и ОС. И знаете ли вы, откуда взялся этот контейнер? Контейнеры основаны на образе, который, в свою очередь, может быть расширением других образов. Вы уверены, что в используемых вами базовых образах нет лазеек? Если вы не меняли экземпляр контейнера в течение шести месяцев, это означает, что исправления операционной системы не применялись в течение этого же времени.

Уследить за всем непросто, поэтому компании вроде Aqua (https://www.aquasec.com) предоставляют инструменты для анализа запущенных в эксплуатацию контейнеров, чтобы вы могли понять, какие проблемы необходимо решить.

Рис. 11.4. Перенос ответственности за некоторые уровни этого стека может снизить сложность

На самом верху этого набора уровней, конечно же, находится код нашего приложения. Он актуален? Дело не только в коде, написанном нами. Как насчет стороннего кода? Ошибка в библиотеке внештатного разработчика может сделать приложение уязвимым для атак. В случае взлома Equifax неисправленная уязвимость на самом деле была в Struts — веб-фреймворке Java.

При серьезном масштабе системы невероятно сложно вычислить, какие микросервисы связываются с библиотеками с известными уязвимостями. Это область, в которой я настоятельно рекомендую использовать такие инструменты, как Snyk или анализатор кода GitHub, способные автоматически сканировать ваши сторонние зависимости и предупреждать, если у вас есть ссылки на подобные библиотеки. Если инструмент найдет таковые, он отправит вам запрос на извлечение, чтобы помочь обновить систему до последних исправленных версий. Вы даже можете встроить это в свой процесс CI и вызвать сбой сборки микросервиса, если он ссылается на библиотеки с проблемами.

Резервное копирование (бэкап)

Я иногда думаю, что создание резервных копий похоже на чистку зубов зубной нитью, поскольку большинство людей говорят, что делают это, хотя на самом деле — нет. Я не чувствую необходимости приводить аргументы в пользу резервных копий, кроме как сказать: вы должны делать бэкапы, потому что данные ценны и нельзя их потерять.

Данные стали более ценными, чем когда-либо, и все же я иногда задаюсь вопросом: не привели ли усовершенствования технологий к тому, что мы перестали уделять пристальное внимание резервному копированию? Диски стали надежнее, чем раньше. Базы данных, скорее всего, поддерживают встроенную репликацию, чтобы избежать потери данных. С применением таких систем можно убедить себя, что резервные копии вовсе не нужны. Но что, если произойдет катастрофическая ошибка и весь ваш кластер Cassandra будет уничтожен? Или если из-за ошибки в коде ваше приложение удалит ценные данные? Резервные копии так же важны, как и прежде. Поэтому, пожалуйста, создавайте бэкап важных данных.

Поскольку развертывание микросервисов автоматизировано, нет необходимости создавать полные запасные копии компьютеров, так как можно пересобрать инфраструктуру из исходного кода. Поэтому мы не пытаемся копировать состояние машин целиком. Вместо этого мы нацеливаем свой процесс резервного копирования на наиболее значимое состояние. Это означает, что наше внимание при создании бэкапа ограничивается такими вещами, как данные в наших БД или, возможно, логи приложений. При правильной технологии файловой системы можно практически мгновенно создавать клоны данных из БД на уровне блоков без заметного прерывания обслуживания клиентов.

Избегайте резервной копии Шрёдингера

При создании бэкапа надо избегать так называемой резервной копии Шрёдингера[1]. Это копия, которая на самом деле может быть резервной копией, а может и не быть. Пока вы фактически не попытаетесь произвести восстановление из нее, вы не знаете, действительно ли это резервная копия[2] или просто набор нулей и единиц, записанных на диск. Лучший способ избежать данной проблемы — убедиться, что дубликат данных является реальным, фактически восстановив его. Найдите способы встроить регулярное восстановление резервных копий в процесс разработки ПО, например используя бэкапы эксплуатируемой системы для создания данных тестирования производительности.

---

[1] Это выдуманный мной термин, но, вероятнее всего, я не единственный, кто использует его.

[2] Точно так же, как Нильс Бор утверждал, что кот Шрёдингера одновременно и жив и мертв, пока вы не откроете коробку, чтобы проверить.

«Старое» руководство по резервному копированию гласит, что бэкапы следует хранить отдельно. Идея в том, чтобы возможный инцидент в ваших офисах или дата-центре не повлиял на ваши копии, если они находятся в другом месте. Однако что означает «отдельно», если ваше приложение развернуто в общедоступном облаке? Важно, чтобы бэкапы хранились максимально изолированно от основной системы, чтобы компрометация не подвергала риску и их. У компании Code Spaces, о которой мы упоминали ранее, были резервные копии, но они хранились в AWS в той же учетной записи, которая была взломана. Если ваше приложение работает на AWS, вы все равно можете хранить бэкапы там же, но необходимо делать это в отдельной учетной записи на отдельных облачных ресурсах. Возможно, вам стоит поместить их в другой облачный регион, чтобы снизить риск возникновения проблем в масштабах всего региона, или же хранить их у другого провайдера.

Поэтому убедитесь, что вы создаете резервные копии важных данных, храните эти резервные копии в системе, отдельной от вашей основной рабочей среды, и удостоверьтесь, что резервные копии действительно работают, регулярно восстанавливая их.

Повторная сборка (ребилд)

Мы можем сделать все возможное, чтобы злоумышленник не получил доступ к нашим системам, но что произойдет, если он все же обойдет защиту? Что ж, часто самое важное, что вы можете сделать на начальном этапе, — снова запустить систему, но предварительно необходимо удалить доступ неавторизованной стороны. Однако это не всегда просто. Я помню, как много лет назад одна из наших машин была взломана руткитом. Руткит — это набор ПО, предназначенный для скрытия действий неавторизованной стороны, и этот метод обычно используется злоумышленниками, которые хотят остаться незамеченными, что дает им время для изучения системы. В нашем случае мы обнаружили, что руткит изменил основные системные команды, такие как `ls` (listing files — «перечисление файлов») или `ps` (to show process listings — «для отображения списков процессов»), чтобы скрыть следы постороннего присутствия. Мы заметили это только тогда, когда сверили хеши программ, запущенных на компьютере, с официальными пакетами. В конце концов нам фактически пришлось переустановить весь сервер с нуля.

Возможность просто стереть сервер с лица земли и полностью пересобрать его может быть невероятно эффективной не только в случае известной атаки, но и для уменьшения воздействия настойчивых злоумышленников. Даже не зная о присутствии вредоносного ПО в вашей системе, вы можете значительно ограничить влияние атак, регулярно проводя ребилд серверов и замену учетных данных.

Ваша способность повторно собрать определенный микросервис или даже целую систему зависит от качества вашей автоматизации и резервного копирова-

ния. Если у вас есть возможность развернуть и настроить каждый микросервис с нуля на основе информации, хранящейся в системе управления версиями, то это уже хорошее начало. Конечно, вам нужно совместить это с надежным процессом восстановления данных из резервной копии. Как и в случае с бэкапом, лучший способ убедиться, что автоматическое развертывание и настройка микросервисов работают, — делать это постоянно, а самый простой способ добиться этого — просто использовать тот же процесс для восстановления микросервиса, что и для каждого развертывания. Конечно, именно так работает большинство процессов развертывания на основе контейнеров. Вы развертываете новый набор контейнеров, работающих под управлением новой версии микросервиса, и завершаете работу старого набора. Выполнение данной операционной процедуры делает ребилд почти незаметным.

Однако есть одно «но», особенно если развертывание происходит на контейнерной платформе, такой как Kubernetes. Возможно, вы часто удаляете и повторно развертываете экземпляры контейнеров, но как насчет самой контейнерной платформы? Есть ли у вас возможность восстановить ее с нуля? Если вы используете полностью управляемый поставщик Kubernetes, развертывание нового кластера может оказаться не слишком сложным, но если вы установили кластер самостоятельно и управляете им, то это окажется нетривиальной задачей.

Возможность выполнить повторную сборку микросервиса и воссоздать его данные в автоматическом режиме помогает восстановить функциональность после атаки. А также предоставляет преимущество в упрощении развертывания по всем направлениям, что положительно сказывается на разработке, тестировании и эксплуатации.

Безусловное или нулевое доверие

Микросервисная архитектура состоит из множества взаимодействующих объектов. Пользователи взаимодействуют с системой через UI. Эти UI, в свою очередь, вызывают микросервисы, а те — еще больше микросервисов. Когда дело доходит до безопасности приложений, нам необходимо рассмотреть вопрос о доверии между всеми этими точками соприкосновения. Как установить приемлемый уровень доверия?

Вскоре мы изучим эту тему с точки зрения аутентификации и авторизации как пользователей, так и микросервисов, но перед этим следует познакомиться с некоторыми фундаментальными моделями, связанными с доверием.

Доверяем ли мы всему, что работает в нашей сети? Или относимся ко всему с подозрением? Здесь можно рассмотреть два типа мышления: безусловное доверие и нулевое доверие.

Безусловное доверие

Первый вариант может заключаться в предположении, что любые вызовы сервиса, сделанные внутри периметра, относятся к безусловно доверенным.

В зависимости от степени конфиденциальности данных это может быть оптимальным вариантом. Некоторые организации пытаются обеспечить безопасность по периметру своих сетей и поэтому предполагают, что им не нужно делать ничего дополнительного, когда два сервиса взаимодействуют друг с другом. Однако, если злоумышленник проникнет в вашу сеть, может начаться настоящий кошмар. Если хакер решит перехватить и прочитать отправляемые данные, изменить их без вашего ведома или даже выдать себя за другого, вы можете об этом не узнать.

Это, безусловно, самая распространенная форма доверия внутри периметра, которую я встречал в организациях. Я не говорю, что это хорошо! Я переживаю, что для большинства организаций, использующих модель безусловного доверия, она не стала сознательным решением. Скорее люди вообще не знают о возможных рисках.

Нулевое доверие

Джилл, мы отследили звонок — он идет из дома!

Из к/ф «Когда звонит незнакомец»

В среде с нулевым доверием стоит *воспринимать* окружение как уже скомпрометированную среду: входящие соединения, компьютеры, с которыми вы взаимодействуете, данные, которые вы записываете, — все взломано и прочитано. Параноидально? Да! Добро пожаловать в зону нулевого доверия.

Нулевое доверие, по сути, — это образ мышления. Это не то, что можно волшебным образом реализовать с помощью продукта или инструмента. Это идея о том, что, если вы работаете с мыслью, что кругом враждебная среда, в которой уже могут присутствовать недобросовестные акторы, тогда вам необходимо принять меры предосторожности, чтобы убедиться, что вы все еще можете работать безопасно. По сути, концепция «периметра» бессмысленна при нулевом доверии (по этой причине нулевое доверие часто также называют «вычислениями без периметра»).

Поскольку вы предполагаете, что система была скомпрометирована, все входящие вызовы от других микросервисов должны быть соответствующим образом проанализированы. *«Действительно ли это клиент, которому стоит доверять?»*

Необходимо также надежно и безопасно хранить все данные и ключи шифрования, и, поскольку мы предполагаем, что кто-то прослушивает нас, все конфиденциальные данные, передаваемые в системе, должны быть зашифрованы.

Интересно, что если вы правильно внедрили образ мышления нулевого доверия, можно начать принимать решения, которые кажутся довольно странными.

> [С нулевым доверием] вы действительно можете принимать определенные нелогичные решения о доступе и, например, разрешать подключения к внутренним сервисам из Интернета, потому что вы относитесь к своей «внутренней» сети с той же степенью доверия, что и к Интернету (то есть никакой).
>
> *Ян Шауманн*[1]

Аргумент Яна здесь заключается в том, что, если вы предполагаете, что ничему внутри вашей сети нельзя доверять и что доверие должно быть восстановлено, можно быть гораздо более гибкими в отношении среды, в которой работает микросервис: вы не ожидаете, что она будет безопасной. Но помните: нулевое доверие нельзя включить по щелчку пальцами. Это основополагающий принцип того, как вы решаете что-то делать. Он должен определять ваши решения, как строить и развивать систему — это то, во что вам придется постоянно инвестировать, чтобы получать отдачу.

Доверие — это спектр возможных вариантов

Я не хочу сказать, что вы должны выбрать или безусловное доверие, или нулевое. Степень, в которой вы доверяете (или не доверяете) другим сторонам в вашей системе, может измениться в зависимости от конфиденциальности информации, к которой осуществляется доступ. Например, вы можете принять концепцию нулевого доверия для любых микросервисов, обрабатывающих PII, но быть не столь бдительными в других областях. Опять же стоимость любой реализации безопасности должна быть оправданна (и определяться) вашей моделью угроз. Решая, стоит ли вам внедрять политику нулевого доверия, ориентируйтесь на свое понимание угроз и их последствий.

В качестве примера посмотрим на MedicalCo, компанию, с которой я работал. Она управляла конфиденциальными медицинскими данными, относящимися к отдельным лицам. Вся информация, которой располагала компания, была классифицирована на основе довольно разумного и прямолинейного подхода.

Общедоступная

Данные, которыми можно было свободно делиться с любой третьей стороной. Эта информация фактически становится общедоступной.

Частная

Информация, которая должна быть доступна только вошедшим в систему пользователям. Доступ к этой информации может быть дополнительно урезан ограничениями авторизации. Например, страховой план клиента.

[1] Ян Шауманн (@jschauma), Twitter. November 5, 2020, 4:22 p. m. https://oreil.ly/QaCm2.

Секретная

Невероятно конфиденциальная информация о физических лицах, которая может быть доступна другим людям, кроме рассматриваемого физического лица, только в крайне специфических ситуациях. Сюда относится информация о здоровье человека.

Затем микросервисы были распределены по категориям на основе наиболее конфиденциальных данных, которые они использовали, и должны были выполняться в соответствующей среде (зоне) с соответствующими элементами управления, как показано на рис. 11.5.

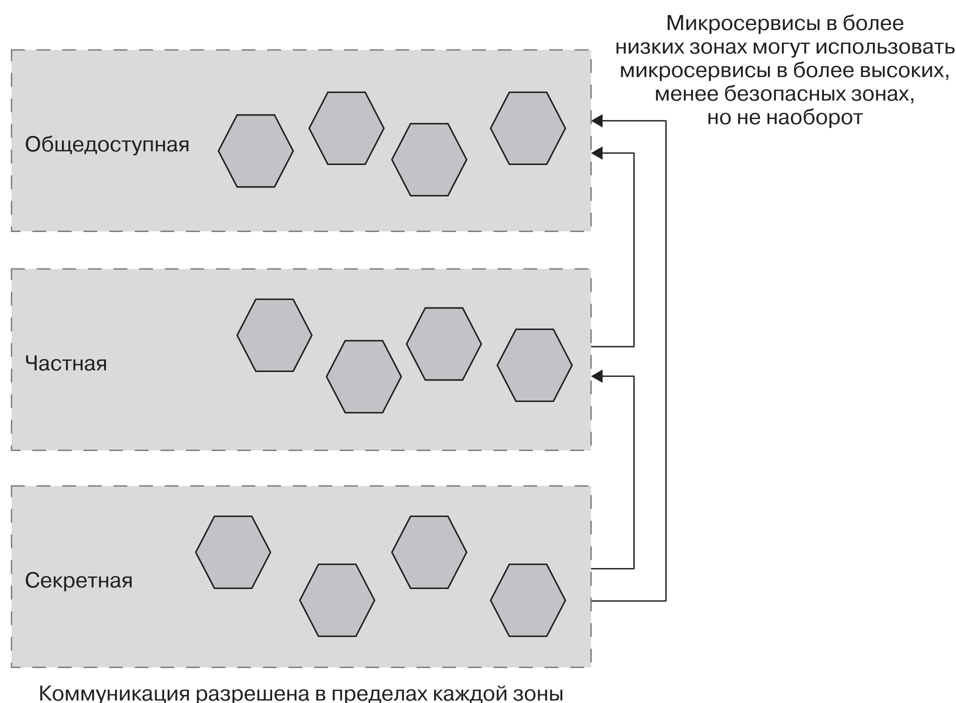


Рис. 11.5. Развертывание микросервисов в разных зонах в зависимости от конфиденциальности обрабатываемых ими данных

Микросервис должен был бы выполняться в зоне, соответствующей наиболее конфиденциальным данным, которые он использовал. Например, микросервис, работающий в общедоступной зоне, может использовать только общедоступные данные. С другой стороны, сервис, использовавший общедоступные и приватные данные, должен был работать в частной зоне, а сервис, имеющий доступ к секретной информации, всегда должен был выполняться в секретной зоне.

Микросервисы внутри каждой зоны взаимодействуют друг с другом, но не могут напрямую получить доступ к данным или функциям в нижних, более безопасных зонах. Однако сервисы в более защищенных зонах могут обращаться к функциональности из менее защищенных.

Компания MedicalCo обеспечила себе возможность гибко варьировать свой подход в каждой зоне. Менее защищенная общедоступная зона может работать в условиях, более близких к безусловному доверию, в то время как секретная зона предполагает применение модели нулевого доверия. Возможно, если бы MedicalCo приняла подход с нулевым доверием во всей своей системе, развертывание микросервисов в отдельных зонах не понадобилось бы, поскольку все вызовы между микросервисами потребовали бы дополнительной аутентификации и авторизации. Тем не менее, еще раз подумав об углубленной защите, я не могу отделаться от мысли, что все же рассмотрел бы такой зонированный подход, учитывая конфиденциальность данных!

Защита данных

Поскольку мы разбиваем монолитное ПО на микросервисы, наши данные перемещаются между системами чаще, чем раньше. Они не просто передаются по сетям, но хранятся на дисках. Если мы не будем осторожны и не позаботимся о защите нашего приложения, то ценные данные, разбросанные по разным местам, могут стать настоящим кошмаром. Давайте более подробно рассмотрим, как защитить свои данные при их перемещении по сетям и в состоянии покоя.

Данные в процессе передачи

Характер имеющихся у вас средств защиты будет во многом зависеть от выбранных протоколов связи. Например, если вы используете HTTP, было бы естественно рассмотреть возможность использования HTTP с защитой транспортного уровня (Transport Layer Security, TLS). Эту тему мы подробнее рассмотрим в следующем разделе. Но если вы используете альтернативные протоколы, такие как связь через брокер сообщений, вам, возможно, придется поискать конкретные технологии поддержки для защиты данных при передаче. Вместо того чтобы копаться в широком спектре технологий в этой области, я думаю, важно рассмотреть четыре основные области, представляющие интерес в сфере защиты данных при передаче, и посмотреть, как эти проблемы могут быть решены с помощью HTTP в качестве примера. Надеюсь, вам не составит большого труда сопоставить эти идеи с любым выбранным протоколом связи.

На рис. 11.6 показаны четыре ключевые проблемы, связанные с передачей данных.

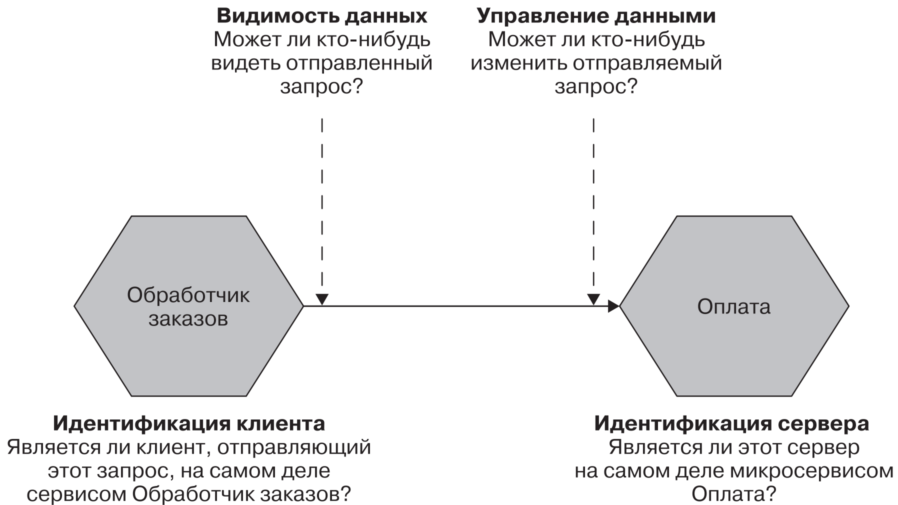


Рис. 11.6. Четыре основные проблемы, возникающие при передаче данных

Рассмотрим каждую проблему немного подробнее.

Идентификация сервера

Одна из самых простых и важных задач — идентификация сервера. Важно понимать, что сервер, с которым вы взаимодействуете, подлинный, потому что злоумышленник теоретически может выдать себя за конечную точку и удалить все полезные данные, которые вы отправляете. Проверка подлинности сервера уже давно представляет собой проблему в Интернете. Это привело к более широкому использованию протокола HTTPS — и в какой-то степени мы можем извлечь выгоду из проделанной работы по обеспечению безопасности Глобальной сети, когда дело доходит до управления внутренними конечными точками HTTP.

Когда люди говорят о HTTPS, они обычно имеют в виду использование HTTP с TLS[1]. Поскольку большая часть взаимодействий осуществляется через публичный Интернет из-за различных потенциальных векторов атак (незащищенный Wi-Fi, «загрязнение» DNS и т. п.), жизненно важно знать, что веб-сайт, на который мы заходим, тот самый. Используя HTTPS, браузер может просмотреть сертификат для этого веб-сайта и убедиться, что он действителен.

[1] Буква S в HTTPS раньше относилась к более старому протоколу Secure Socket Layer (SSL), который по ряду причин был заменен на TLS. Как ни странно, термин SSL все еще употребляется, даже когда на самом деле подразумевается TLS. Библиотека OpenSSL, например, широко применяется для реализации TLS, и когда вам выдают сертификат SSL, это фактически TLS. Мы не облегчаем себе жизнь, правда?

Это очень разумный механизм безопасности. Фраза «HTTPS повсюду» стала лозунгом общественного Интернета, и не без оснований.

Стоит отметить, что некоторые протоколы связи, применяющие HTTP, могут использовать преимущества HTTPS. Так что можно легко запускать SOAP или gRPC по HTTPS. Протокол HTTPS также дает дополнительную защиту, помимо простой идентификации сервера. Мы скоро к этому вернемся.

Идентификация клиента

При упоминании идентификации клиента в этом контексте имеется в виду вышестоящий микросервис, выполняющий вызов, и мы пытаемся подтвердить и аутентифицировать этот микросервис. О том, как происходит аутентификация людей (пользователей!), поговорим чуть позже.

Провести идентификацию клиента можно несколькими способами. Самый простой — попросить клиента прислать нам в запросе информацию, кто он такой. Примером может быть использование какого-либо общего секрета или сертификата на стороне клиента для подписи запроса. Когда сервер должен проверить личность клиента, нужно, чтобы это было максимально эффективно, — я видел некоторые решения (в том числе те, которые предлагаются поставщиками API-шлюзов), которые предполагают, что серверу приходится обращаться в центральные сервисы для проведения идентификации клиента, что довольно глупо, если учесть последствия задержки.

Я с трудом представляю ситуацию, в которой возможна проверка идентификации клиента без проверки идентификации сервера. Чтобы проверить и то и другое, обычно в итоге реализуется некоторая форма взаимной аутентификации. При взаимной аутентификации обе стороны проверяют подлинность друг друга. Так, на рис. 11.6 сервис `Обработчик заказов` аутентифицирует микросервис `Оплата`, а микросервис `Оплата` аутентифицирует сервис `Обработчик заказов`.

Это можно реализовать с помощью *протокола взаимной идентификации mutual TLS* (MTLS), и в этом случае как клиент, так и сервер используют сертификаты. В Интернете проверка подлинности клиентского устройства обычно менее важна, чем проверка личности человека, использующего это устройство. Как таковой, протокол MTLS используется редко. Однако в нашей микросервисной архитектуре, особенно там, где можно работать в среде с нулевым доверием, такая схема встречается гораздо чаще.

Проблема с внедрением таких протоколов, как MTLS, исторически заключалась в инструментарии. В настоящее время это проявляется уже не так явно. Такие инструменты, как Vault, могут способствовать распространению сертификатов, и желание упростить использование MTLS является одной из основных причин, по которым люди внедряют сервисные сети, рассмотренные в разделе «Сервисные сети и API-шлюзы» главы 5.

Видимость данных

Когда мы отправляем данные из одного микросервиса в другой, может ли кто-нибудь подсмотреть их? За некоторые данные, например цены на альбомы Питера Андре, мы можем не беспокоиться, поскольку эта информация уже является доступной для всех. С другой стороны, некоторые данные могут включать в себя PII, и нам необходимо убедиться, что они защищены.

Когда вы используете обычный HTTPS или MTLS, данные не будут видны промежуточным сторонам, так как протокол TLS шифрует отправляемые данные. Это может быть проблематично, если вы точно хотите, чтобы данные отправлялись в открытом виде. Например, обратные прокси, такие как Squid или Varnish, могут кэшировать HTTP-ответы, но это невозможно с HTTPS.

Управление данными

Мы могли бы представить себе ряд ситуаций, в которых управление отправляемыми данными может быть опасным. Например, изменение суммы отправляемых денег. Поэтому, как показано на рис. 11.6, нам нужно убедиться, что злоумышленник не в состоянии изменить запрос, направляющийся от сервиса `Обработчик заказов` к сервису `Оплата`.

Как правило, типы защиты, которые делают данные невидимыми, также гарантируют, что данными нельзя управлять (например, это делает HTTPS). Однако мы можем решить отправлять данные в открытом виде, но при этом хотим убедиться, что ими нельзя манипулировать. Для HTTP одним из таких подходов будет использование *кода аутентификации сообщений на основе хеш-функций* (hash-based message authentication code, HMAC) для подписи отправляемых данных. С помощью HMAC вместе с данными генерируется и отправляется хеш, и получатель может сверить хеш с данными, чтобы убедиться, что они не были изменены.

Данные в состоянии покоя

Данные на хранении — это ответственность, особенно если они конфиденциальны. Надеюсь, мы сделали все возможное, чтобы злоумышленники не смогли проникнуть в нашу сеть и взломать наши приложения или операционные системы для доступа к основным данным. Однако мы должны быть к такому готовы.

Многие из громких случаев нарушения безопасности, о которых мы слышим, связаны с тем, что данные, находящиеся в состоянии покоя, попадают в руки злоумышленника и он может их прочитать. Это происходит либо потому, что данные хранились в незашифрованном виде, либо потому, что механизм, используемый для защиты данных, имел фундаментальный недостаток.

Механизмов, с помощью которых могут быть защищены данные в состоянии покоя, много, и они разнообразны, но есть некоторые общие моменты, о которых следует помнить.

Используйте хорошо знакомые подходы

В некоторых случаях можно переложить задачу шифрования данных на существующее ПО, например используя встроенную поддержку шифрования в БД. Однако, если вам потребуется шифровать и дешифровывать данные в вашей собственной системе, убедитесь, что используете хорошо известные и проверенные методы. Самый простой способ испортить шифрование данных — это попытаться реализовать свои собственные алгоритмы шифрования или даже внедрить чужие. Какой бы язык программирования вы ни использовали, у вас будет доступ к проверенным, регулярно исправляемым версиям хорошо зарекомендовавших себя алгоритмов шифрования. Используйте их! И подпишитесь на почтовые рассылки/списки рекомендаций по выбранной вами технологии, чтобы знать об уязвимостях по мере их обнаружения, тогда вы будете иметь возможность вовремя исправлять хранилища.

Для защиты паролей обязательно используйте метод, называемый *«соленым» хешированием паролей* (https://oreil.ly/kXUbY). Это гарантирует, что пароли никогда не будут храниться в виде обычного текста и что, даже если злоумышленник взломает один такой пароль, он не сможет автоматически прочитать другие[1].

Плохо реализованное шифрование может быть хуже, чем его отсутствие, поскольку ложное чувство безопасности, вероятно, заставит вас потерять бдительность.

Выберите свои цели

Предположение, что все должно быть зашифровано, может несколько упростить ситуацию. Нет никаких догадок, что должно быть защищено, а что — нет. Однако вам все равно необходимо подумать, какие данные стоит поместить в файлы логов, чтобы помочь в выявлении проблем, так как вычислительные затраты на шифрование абсолютно всего довольно обременительны и в результате потребуют более мощного оборудования. Это еще сложнее, когда вы применяете миграцию БД как часть рефакторинга схем. В зависимости от вносимых изменений данные, возможно, потребуется расшифровать, перенести и повторно зашифровать.

Разделив свою систему на более детализированные сервисы, можно обнаружить целое хранилище данных, которое можно зашифровать оптом, но это маловероятно. Разумным подходом будет ограничение этого шифрования известным набором таблиц.

Будьте экономны

По мере удешевления дискового пространства и расширения возможностей баз данных стремительно возрастает простота сбора и хранения больших объемов информации. Эти данные ценны не только для самих компаний,

[1] Мы не шифруем пароли в состоянии покоя, поскольку шифрование означает, что любой, у кого есть правильный ключ, может перечитать пароль.

рассматривающих их как ценный актив, но и для пользователей, которым важна собственная конфиденциальность. С данными, которые могут быть использованы для получения информации о ком-либо, следует обращаться наиболее осторожно.

Однако что, если немного облегчить себе жизнь? Почему бы не вычистить как можно больше информации, позволяющей установить личность, и сделать это как можно скорее? При регистрации запроса от пользователя нужно ли нам постоянно хранить IP-адрес целиком или можно заменить последние несколько цифр на «x»? Нужно ли сохранять чье-либо имя, возраст, пол и дату рождения, чтобы предоставлять им предложения продуктов, или их возрастной диапазон и почтовый индекс будут достаточной информацией?

Преимущества экономии при сборе данных многочисленны. Если вы не храните данные, никто не сможет их украсть и никто (например, правительственное учреждение) не сможет их запросить!

Немецкое слово *Datensparsamkeit* представляет эту концепцию. Основанная на немецком законодательстве о конфиденциальности, она включает в себя концепцию хранения только *минимально необходимого* для выполнения деловых операций или соблюдения местных законов объема информации.

Очевидно, что это прямо противоречит движению к хранению все большего количества информации, но осознание того, что это противоречие вообще существует, — хорошее начало!

Все дело в ключах

Большинство форм шифрования предполагает использование некоторого ключа в сочетании с подходящим алгоритмом для создания зашифрованных данных. Для расшифровки данных, чтобы их можно было прочитать, авторизованным сторонам потребуется доступ к ключу — либо к тому же, либо к другому (в случае шифрования с открытым ключом). Итак, где хранятся ваши ключи? Если я шифрую свои данные, потому что беспокоюсь, что кто-то украдет всю мою БД, и храню используемый ключ в той же БД, то это нельзя назвать умным ходом! Следовательно, нам нужно хранить ключи где-то в другом месте. Но где?

Одним из решений может стать применение отдельного устройства безопасности для шифрования и дешифрования данных. Или использование отдельного хранилища ключей, к которому ваш сервис получает доступ, когда ему необходим ключ. Управление жизненным циклом ключей (и доступом к их изменению) может быть жизненно важной операцией, и эти системы могут справиться с этим за вас. Именно здесь может пригодиться хранилище HashiCorp от Vault.

Некоторые БД даже включают встроенную поддержку шифрования, например прозрачное шифрование данных SQL Server, целью которого является прозрачная обработка данных. Даже если выбранная вами база данных включает такую

поддержку, изучите, как обрабатываются ключи и действительно ли устраняется угроза, от которой вы защищаетесь.

Это довольно сложный вопрос. Главное, избегайте внедрения собственного шифрования и проведите несколько хороших исследований!

Шифруйте данные сразу. Расшифровывайте только по требованию и убедитесь, что незашифрованные данные никогда и нигде не хранятся.

Шифрование резервных копий

Резервные копии — это хорошо. Необходимо создавать бэкапы наших важных данных. И это может показаться очевидным, но если данные настолько конфиденциальны, что мы хотим зашифровать их в нашей работающей системе, то стоит убедиться, что любые резервные копии этих данных также зашифрованы!

Аутентификация и авторизация

Аутентификация и авторизация представляют собой ключевые понятия, когда речь идет о людях и процессах, которые взаимодействуют с нашей системой. В контексте безопасности *аутентификация* — это процесс подтверждения личности. Обычно пользователь-человек аутентифицируется путем ввода своих логина и пароля. Предполагается, что только фактический пользователь имеет доступ к этой информации, и поэтому лицо, вводящее ее, должно быть им. Конечно, существуют и другие, более сложные системы. Наши телефоны теперь позволяют подтверждать свою личность по отпечатку пальца или лицу. Как правило, когда мы абстрактно говорим, кто или что проходит проверку подлинности, мы называем эту сторону *принципалом* (principal).

Авторизация — это механизм, с помощью которого сопоставляются манипуляции участника с разрешенными ему действиями. Часто, когда принципал проходит аутентификацию, о нем предоставляется информация, помогающая принять решение, что ему позволяется делать. Например, нам могут сообщить, в каком отделе или офисе пользователь работает — информация, которую наша система использует, чтобы решить, что принципалу разрешено делать, а что — нет.

Очень важно, чтобы пользователям было легко получить доступ к нашей системе. Мы не хотим, чтобы для доступа к различным микросервисам им приходилось использовать свои логин и пароль для каждого сервиса отдельно. Поэтому стоит реализовать технологию единого входа (SSO) в среде микросервисов.

Аутентификация между сервисами

Ранее мы обсуждали двухсторонний протокол TLS, который, помимо защиты передаваемых данных, также позволяет реализовать форму аутентификации. Когда клиент взаимодействует с сервером, используя MTLS, сервер может аутентифицировать клиента, а клиент может аутентифицировать сервер — это форма аутентификации между сервисами. Помимо MTLS, для реализации этого процесса есть и другие схемы. Например, использование API-ключей, когда клиенту необходимо использовать ключ для хеширования запроса таким образом, чтобы сервер мог проверить, что клиент указан действительный ключ.

Аутентификация человека

Мы привыкли к тому, что люди аутентифицируют себя с помощью логина и пароля. Однако все чаще это используется лишь как часть подхода к многофакторной аутентификации, когда пользователю может потребоваться более одного элемента знаний *(фактора)* для аутентификации. Нередко это принимает форму многофакторной аутентификации (multifactor authentication, MFA)[1], где требуется более одного параметра. MFA чаще всего предполагает использование обычной комбинации логина и пароля в дополнение к предоставлению по крайней мере одного дополнительного условия.

В последние годы увеличилось количество различных типов факторов аутентификации — от кодов, отправляемых по SMS, и «волшебных ссылок» (magic links), отправляемых по электронной почте, до специализированных мобильных приложений, таких как Authy (https://authy.com), и аппаратных устройств USB и NFC, например YubiKey (https://www.yubico.com). Биометрические данные также часто используются в настоящее время, поскольку пользователям стало доступно больше оборудования, поддерживающего такие функции, как распознавание лиц или отпечатков пальцев. В то время как MFA показала себя гораздо более безопасной в качестве общего подхода, и многие государственные службы поддерживают ее, она не прижилась на массовом рынке, хотя, я думаю, это изменится. Для управления аутентификацией ключевых сервисов, жизненно важных для работы вашего ПО или позволяющих получить доступ к особо конфиденциальной информации (например, доступ к исходному коду), я бы счел использование MFA обязательным.

[1] Раньше мы бы говорили о двухфакторной аутентификации (2FA). MFA — это та же концепция, но позволяющая пользователям предоставлять дополнительный фактор с различных устройств, таких как защищенный токен, мобильное приложение для аутентификации или, возможно, биометрические данные. Вы можете рассматривать 2FA как подгруппу MFA.

Общие реализации единого входа

Общий подход к аутентификации заключается в использовании какого-либо решения для SSO, гарантирующего, что пользователь должен аутентифицироваться только один раз за сеанс, даже если во время этого сеанса он планирует взаимодействовать с несколькими нижестоящими сервисами или приложениями. Например, когда вы входите в систему со своей учетной записью Google, вы входите в Google Calendar, Gmail и Google Docs, хотя это отдельные системы.

Когда принципал пытается получить доступ к ресурсу (например, к веб-интерфейсу), ему направляется запрос на аутентификацию с помощью *провайдера идентификационных данных* (identity provider, IdP). Провайдер может попросить их предоставить логин и пароль или потребовать использования чего-то более продвинутого, например MFA. Как только IdP удостоверяется, что клиент прошел аутентификацию, он предоставляет информацию *поставщику услуг*, позволяя ему определять, предоставить ли пользователю доступ к ресурсу.

IdP может быть внешней системой или чем-то внутри вашей собственной организации. Google, например, предоставляет программу OpenID Connect. Однако для предприятий обычно требуется собственный провайдер идентификационных данных, который может быть связан со *службой каталогов* вашей компании. Служба каталогов может быть чем-то вроде облегченного протокола доступа к каталогам (Lightweight Directory Access Protocol, LDAP) или Active Directory, AD. Эти системы позволяют хранить информацию о принципалах, например о том, какие роли они играют в организации. Часто служба каталогов и IdP являются одним и тем же, в то время как в некоторых случаях они разделены, но связаны. Например, Okta — это удаленный поставщик удостоверений SAML, обрабатывающий такие задачи, как двухфакторная аутентификация, но может ссылаться на службы каталогов вашей компании в качестве источника достоверности.

Таким образом, IdP предоставляет системе информацию, кем является принципал, но решение, что этому принципалу разрешено делать, принимает система.

SAML — это стандарт на основе протокола SOAP, и он известен тем, что с ним довольно сложно работать, несмотря на библиотеки и инструментарий, доступные для его поддержки. С момента выхода первого издания он быстро потерял популярность[1]. OpenID Connect — это стандарт, который появился как конкретная реализация OAuth 2.0, основанная на том, как Google и другие компании работают с SSO. В OAuth используются более простые вызовы REST, и отчасти благодаря своей относительной простоте и широкой поддержке данный протокол стал доминирующим механизмом SSO для конечных пользователей и получил значительное распространение на предприятиях.

[1] Я не виноват!

SSO-шлюз

Мы могли бы принять решение обрабатывать перенаправление на провайдер идентификационных данных и обмен данными с ним в каждом микросервисе, чтобы любой неаутентифицированный запрос от внешней стороны обрабатывался должным образом. Очевидно, что при таком подходе дублируется множество функций в наших микросервисах. Общая библиотека могла бы помочь, но мы должны быть осторожны, чтобы избежать связи, которая рискует возникнуть из общего кода (подробнее см. в разделе«DRY и опасности повторного использования кода в мире микросервисов» главы 5). Общая библиотека также не помогла бы, если бы у нас были микросервисы, написанные в разных технологических стеках.

Вместо того чтобы заставлять каждый сервис управлять обменом данными с нашим провайдером идентификационных данных, более распространенным подходом стало использование шлюза в качестве прокси-сервера, расположенного между вашими сервисами и внешним миром (как показано на рис. 11.7). Идея в том, что мы можем централизовать поведение для перенаправления пользователя и установить связь только в одном месте.

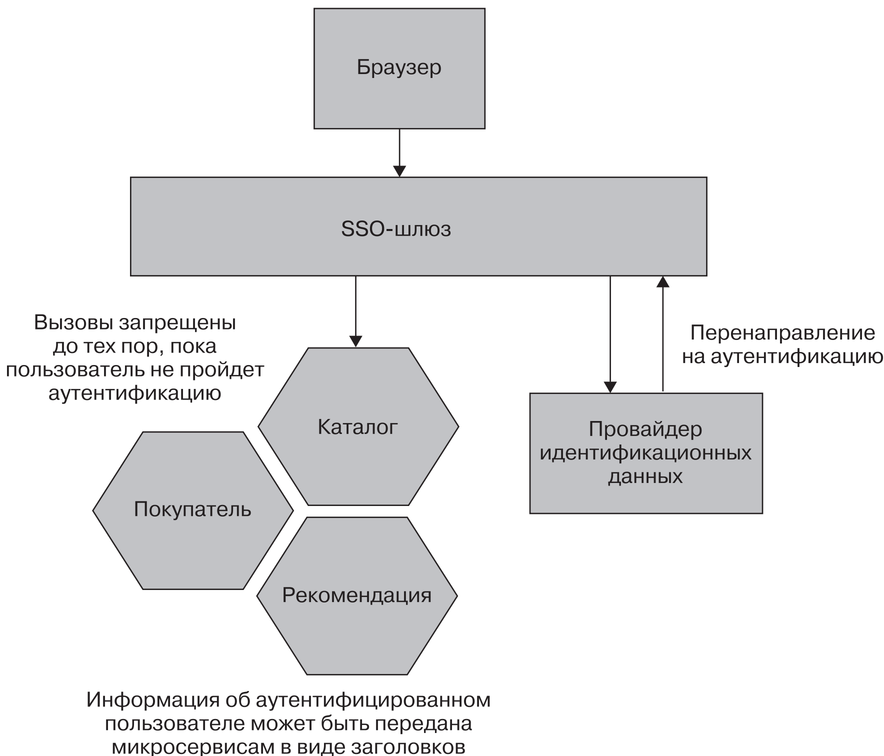


Рис. 11.7. Использование шлюза для обработки SSO

Однако нам все еще нужно решить проблему того, как нижестоящий сервис получает информацию о пользователях, например их логин или какие роли исполняют пользователи. При использовании HTTP можно настроить свой шлюз на заполнение заголовков этой информацией. Shibboleth — это один из инструментов, способный сделать данную работу за вас, и я видел, как он использовался с веб-сервером Apache для обработки интеграции с IdP на основе SAML, что дало превосходный результат. Альтернатива, которую мы рассмотрим более подробно в ближайшее время, — создание веб-токена JSON (JWT, JSON Web Token), содержащего всю информацию о принципале. Это даст ряд преимуществ, в том числе возможность легче передавать данные от микросервиса к микросервису.

Еще один момент, связанный с использованием SSO-шлюза, заключается в том, что, если переложить ответственность за аутентификацию на шлюз, может стать сложнее предугадать поведение микросервиса, если рассматривать его изолированно. Помните, в главе 9 мы изучали некоторые проблемы, связанные с воспроизведением условий, приближенных к эксплуатационным? Если вы решите использовать шлюз, убедитесь, что у разработчиков есть возможность запускать свои сервисы за ним без особых усилий.

Последняя проблема такого подхода заключается в том, что он может дать вам ложное чувство безопасности. Опять же я хотел бы вернуться к идее углубленной обороны — от периметра сети до подсети, брандмауэра, компьютера, ОС и аппаратного обеспечения. Существует возможность реализовать меры безопасности во всех этих точках. Я видел, как некоторые люди кладут все яйца в одну корзину, надеясь, что шлюз сделает за них каждый шаг. И мы все знаем, что происходит, когда в системе появляется единая точка отказа...

Очевидно, что вы могли бы использовать этот шлюз для других целей. Например, вы также можете принять решение о прекращении использования HTTPS на этом уровне, запустить обнаружение вторжений и т. д. Однако будьте осторожны: уровни шлюза, как правило, имеют тенденцию к росту функциональности, что само по себе может в конечном счете стать гигантской точкой связанности. И чем больше функциональности у какого-то элемента, тем больше поверхность атаки.

Детализированная авторизация

Шлюз вполне способен обеспечить довольно эффективную грубую аутентификацию. Например, он может отказать в доступе к приложению службы поддержки всем не вошедшим в систему пользователям. Предполагая, что шлюз извлекает атрибуты участника в результате аутентификации, он может принимать более тонкие решения. Например, обычно людей объединяют в группы или назначают им роли. Допускается использовать эту информацию для понимания, что им разрешено делать. Таким образом, приложению службы поддержки можно было бы разрешить доступ только принципалам с определенной ролью (например, `ПЕРСОНАЛ`). Однако, помимо разрешения (или запрета) доступа к определенным ресурсам или конечным точкам, требуется оставить все остальное на

усмотрение самого микросервиса — ему необходимо будет принять дальнейшие решения о том, какие операции разрешить.

Вернемся к нашему приложению службы поддержки: разрешаем ли мы любому сотруднику видеть все детали? Скорее всего, на работе у каждого сотрудника своя роль. Например, пользователю из группы `КОЛЛ_ЦЕНТР` может быть разрешено просматривать любую информацию о клиенте, кроме его платежных реквизитов. Этот принципал также может осуществлять возврат средств, но сумма может быть ограничена. Однако получившему роль `ЛИДЕР_ГРУППЫ_КОЛЛ_ЦЕНТРА` разрешается осуществлять возвраты в более крупных размерах.

Такие решения должны быть локальными для конкретного микросервиса. Я видел, как люди используют различные атрибуты, предоставляемые провайдерами идентификационных данных, ужасным образом, прибегая к чрезмерно мелким ролям, таким как `КОЛЛ_ЦЕНТР_ВОЗВРАТ_СРЕДСТВ_50_ДОЛЛАРОВ`, где они в конечном счете помещают информацию, специфичную для одной части функциональности микросервиса, в свои службы каталогов. Это кошмар для обслуживания и такой подход дает сервисам очень мало возможностей иметь собственный независимый жизненный цикл, поскольку неожиданно фрагмент информации о том, как ведет себя сервис, находится, например, в системе, управляемой другой частью организации.

Тема обеспечения микросервиса информацией, необходимой для оценки более детализированных запросов на авторизацию, заслуживает отдельного обсуждения. Мы вернемся к этому вопросу, когда будем рассматривать JWT-токены.

И все же я советую отдавать предпочтение более общим ролям, смоделированным в соответствии с тем, как работает ваша организация. Возвращаясь к первым главам, помните, что мы создаем ПО, опираясь на методы работы нашей организации. Так что используйте роли и таким образом.

Проблема иерархии полномочий

Аутентификация пользователя в системе с помощью чего-то вроде SSO-шлюза достаточно проста, чего может быть достаточно для управления доступом к микросервису. Но что произойдет, если данному микросервису позже потребуется совершить дополнительные вызовы для завершения операции? Это может оставить нашу систему открытой для уязвимости, известной как «проблема запутанного заместителя» (confused deputy problem). Смысл данного выражения в том, что некий микросервис с меньшими привилегиями или правами на выполнение определенного действия может заставить другой вышестоящий сервис выполнить это действие. Рассмотрим конкретный пример на рис. 11.8, который иллюстрирует сайт онлайн-покупок MusicCorp. Наш браузерный UI на JavaScript взаимодействует с микросервисом `Интернет-магазин` на стороне сервера, который представляет собой разновидность бэкенда для фронтенда. Мы рассмотрим это более подробно в разделе «Шаблон: бэкенд для фронтенда (BFF)» главы 14, но на данный момент думайте об этом как о серверном компоненте, выполняющем

агрегацию и фильтрацию для определенного внешнего интерфейса (в нашем случае — браузерный JavaScript-UI). Вызовы, выполняемые между браузером и сервисом `Интернет-магазин`, могут быть аутентифицированы с помощью OpenID Connect. Пока все идет хорошо.

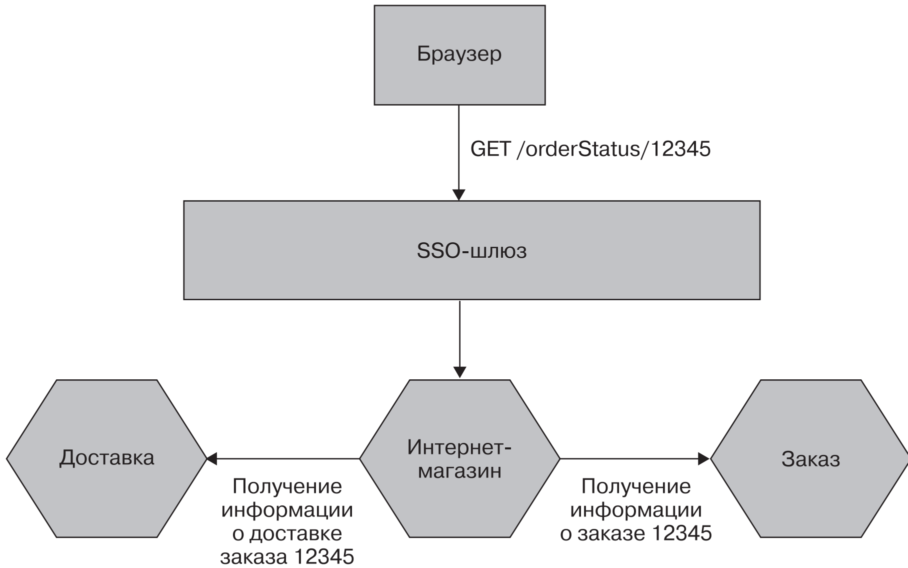


Рис. 11.8. Пример, когда в игру может вступить «запутанный заместитель»

Когда пользователь входит в систему, он может нажать на ссылку, чтобы просмотреть детали заказа. Чтобы отобразить информацию, необходимо извлечь исходный заказ из сервиса `Заказ`, но мы также хотим просмотреть информацию о доставке заказа. Поэтому, когда зарегистрированный клиент щелкает на ссылке `/orderStatus/12345`, этот запрос направляется сервису `Интернет-магазин`, который затем должен вызвать стоящие ниже по потоку микросервисы `Заказ` и `Доставка` для получения информации о заказе `12345`.

Но должны ли эти нижестоящие сервисы принимать вызовы от сервиса `Интернет-магазин`? Можно было бы занять позицию безусловного доверия: поскольку вызов поступил в пределах периметра — все в порядке. Мы могли бы даже использовать сертификаты или API-ключи, чтобы подтвердить, что действительно это сервис `Интернет-магазин` запрашивает информацию. Но достаточно ли этого? Например, клиент, вошедший в систему онлайн-покупок, может увидеть данные своей личной учетной записи. Что, если клиент мог бы обмануть пользовательский интерфейс интернет-магазина, заставив его запросить *чужие* данные, возможно выполнив вызов вместе со своими собственными учетными данными?

Что помешает клиенту в данном примере запрашивать не принадлежащие ему заказы? Войдя в систему, он может начать отправлять запросы на другие

заказы, которые ему не принадлежат, чтобы узнать, сможет ли он получить полезную информацию. Он мог бы начать угадывать идентификаторы заказов, чтобы посмотреть, можно ли извлечь информацию других людей. По сути, здесь произошло то, что, хотя пользователь из примера прошел *аутентификацию*, мы не обеспечили достаточной *авторизации*. Нужно, чтобы какая-то часть системы могла определить, что запрос на просмотр сведений о пользователе А может быть удовлетворен только в том случае, если его осуществляет именно пользователь А. Но где же располагается эта логика?

Авторизация в вышестоящих элементах, централизованная

Один из вариантов избежать вышеописанной проблемы — выполнить всю необходимую авторизацию, как только запрос будет получен в нашей системе. На рис. 11.8 это означало бы, что мы будем стремиться авторизовать запрос либо в самом шлюзе единого входа, либо в сервисе `Интернет-магазин`. Идея в том, что к тому времени, когда вызовы отправляются в микросервис `Заказ` или `Доставка`, предполагается, что запросы разрешены.

Эта форма авторизации в вышестоящих элементах фактически подразумевает, что мы принимаем некоторую форму безусловного доверия (в отличие от нулевого доверия) — микросервисы `Доставка` и `Заказ` должны считать, что им отправляются только разрешенные к выполнению запросы. Другая проблема заключается в том, что вышестоящий элемент, например шлюз или что-то подобное, должен знать, какие функциональные возможности предоставляют нижестоящие микросервисы и как ограничить доступ к этой функциональности.

Однако в идеале нужно, чтобы микросервисы были как можно более автономными, чтобы максимально упростить внесение изменений и развертывание новой функциональности. Мы хотим, чтобы наши релизы проходили как можно проще, нужна возможность независимого развертывания. Если процесс развертывания теперь включает в себя как развертывание нового микросервиса, так и применение некоторой связанной с авторизацией конфигурации на вышестоящем шлюзе, то мне это не кажется слишком «независимым».

Поэтому хотелось бы, чтобы решение о том, должен ли вызов быть авторизован, принималось в том же микросервисе, где находится запрашиваемая функциональность. Это делает микросервис более автономным, а также дает возможность при желании реализовать нулевое доверие.

Децентрализованная авторизация

Учитывая сложности централизованной авторизации в среде микросервисов, стоит перенести эту логику на нижестоящий микросервис. В микросервисе `Заказ` находится функциональность для доступа к информации о заказе, поэтому логично, что именно этот сервис будет решать, является ли вызов

действительным. Однако в данном конкретном случае микросервису `Заказ` требуется информация, какой человек делает запрос. Итак, как нам передать эти сведения в микросервис `Заказ`?

На самом простом уровне мы можем потребовать, чтобы идентификатор человека, отправляющего запрос, был отправлен в микросервис `Заказ`. Например, при использовании HTTP можно было бы просто вставить имя пользователя в заголовок. Но что в таком случае может помешать злоумышленнику вставить просто любое старое имя в запрос для получения необходимой ему информации? В идеале нужен способ убедиться, что запрос действительно делается от имени пользователя, прошедшего аутентификацию, и что мы можем передать дополнительную информацию об этом пользователе.

Так сложилось, что существует множество различных способов решения данной проблемы (включая такие методы, как вложенные утверждения SAML, столь же болезненные, как их звучание), но в последнее время наиболее распространенным приемом стало использование веб-токенов JSON.

Веб-токены JSON

JWT-токены позволяют хранить несколько заявок о человеке в строке, которую можно передавать. Этот токен имеет подпись, гарантирующую, что структура токена не изменялась, и дополнительное шифрование. Хотя JWT-токены могут использоваться для общего обмена информацией, когда важно убедиться, что данные не были подделаны, они чаще всего используются для передачи информации с целью авторизации.

После подписания JWT-токены можно легко передать по различным протоколам, а также дополнительно настроить период действия. Они широко поддерживаются рядом IdP, поддерживающих генерацию JWT-токенов, и большим количеством библиотек для использования JWT внутри вашего собственного кода.

Формат

Основная полезная нагрузка JWT — это структура JSON, способная содержать все что угодно. Вариант токена показан в примере 11.1. Стандарт JWT описывает (https://oreil.ly/2Yo5X) некоторые специально названные поля («публичные заявки»), которые вы должны использовать, если они к вам относятся. Например, `exp` определяет дату истечения срока действия токена. Если вы правильно применяете эти поля, есть большая вероятность, что используемые вами библиотеки смогут действовать надлежащим образом, например отклонять токен, если в поле `exp` указано, что срок его действия уже истек[1]. Даже если вы не будете использо-

[1] Сайт JWT (https://jwt.io) содержит отличный обзор того, какие библиотеки поддерживают те или иные публичные заявления, это вообще отличный ресурс по всем вопросам JWT.

вать все эти поля, стоит знать, что они собой представляют, чтобы убедиться, что вы не используете их для своих собственных приложений, поскольку это может привести к странному поведению поддерживающих библиотек.

Пример 11.1. Пример полезной нагрузки JSON в JWT

```
{
  "sub": "123",
  "name": "Sam Newman",
  "exp": 1606741736,
  "groups": "admin, beta"
}
```

В примере 11.2 показан закодированный токен из примера 11.1. Этот токен на самом деле представляет собой всего лишь одну строку, но он разбит на три части, разделенные знаком «.», — заголовок, полезную нагрузку и подпись.

Пример 11.2. Результат кодирования полезной нагрузки JWT

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. ❶
eyJzdWIiOiIxMjMiLCJuYW1lIjoiU2FtIE5ld21hbiIsImV4cCI6MTYwNjc0MTczNiwiZ3J... . ❷
Z9HMH0DGs60I0P5bVVSFixeDxJjGovQEtlNUi__iE_0 ❸
```

❶ Заголовок.

❷ Полезная нагрузка (укороченное отображение).

❸ Подпись.

Для удобства просмотра данного примера я разделил строку на части, но на самом деле это одна строка без разрывов. Заголовок содержит информацию об используемом алгоритме подписи. Это позволяет программе, декодирующей токен, поддерживать различные схемы подписи. Полезная нагрузка — место, где хранится информация о заявках, которые делает токен, — это всего лишь результат кодирования структуры JSON в примере 11.1. Подпись используется для гарантии того, что полезная нагрузка не подвергалась манипуляциям, а токен был сгенерирован тем, кем вы полагали (при условии, что токен подписан закрытым ключом).

В виде простой строки токен может быть легко передан по различным протоколам связи, например, как заголовок в HTTP (в заголовке `Authorization`) или, возможно, как часть метаданных в сообщении. Эта закодированная строка, конечно, может быть отправлена по зашифрованному транспортному протоколу, например TLS через HTTP. В этом случае токен не будет виден людям, наблюдающим за обменом данными.

Использование токенов

Рассмотрим популярный способ использования JWT-токенов в микросервисной архитектуре. На рис. 11.9 покупатель входит в систему как обычно, и после аутентификации генерируется какой-то токен для представления его сеанса

входа в систему (вероятно, Oauth-токен), хранящийся на клиентском устройстве. Последующие запросы с данного клиентского устройства попадают на наш шлюз, генерирующий JWT-токен, который будет актуален в течение всего срока действия запроса. Именно этот JWT-токен затем передается нижестоящим микросервисам. Они могут проверять его и извлекать заявки из полезной нагрузки, чтобы определить, какой тип авторизации будет подходящим.

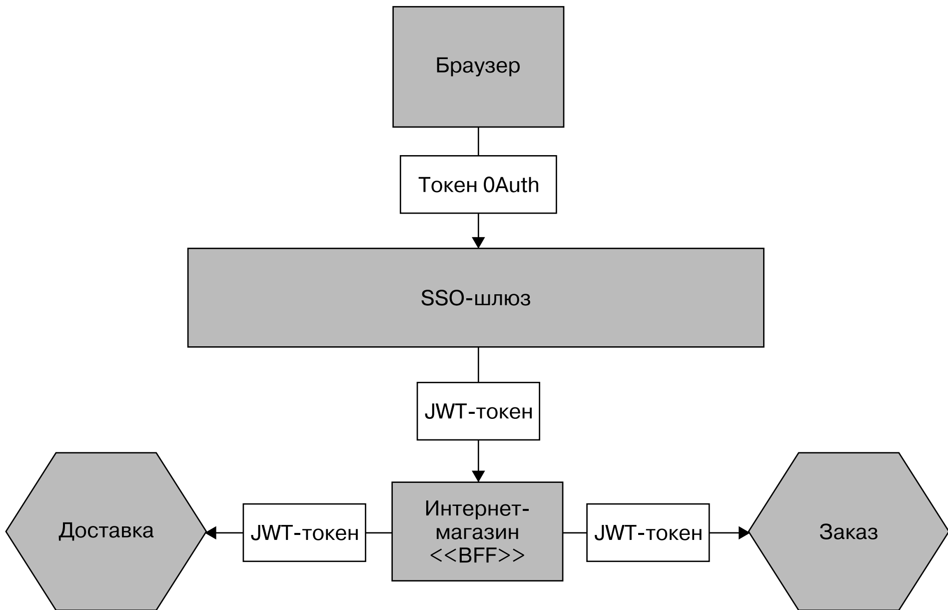


Рис. 11.9. JWT-токен генерируется для конкретного запроса и передается нижестоящим микросервисам

Разновидностью этого подхода будет генерация JWT-токена при первоначальной аутентификации пользователя в системе и дальнейшее сохранение этого JWT-токена на клиентском устройстве. Однако стоит учитывать, что такой токен должен быть действителен в течение всего сеанса входа в систему. Ранее обсуждалось, что стоило бы ограничить срок действия учетных данных, сгенерированных системой, чтобы снизить вероятность их неправильного использования. Также необходимо уменьшить последствия для системы, если потребуется изменить ключи, используемые для формирования закодированного токена. Генерация JWT-токена на основе каждого запроса, по-видимому, стала наиболее распространенным решением данной проблемы, как показано на рис. 11.9. Обмен токенами в шлюзе также может значительно упростить использование JWT-токенов без необходимости изменять какую-либо часть процесса аутентификации, включающую в себя связь с клиентским устройством, даже

если у вас уже есть работающее решение единого входа, скрывающее факт использования JWT-токена, в том числе из основного потока аутентификации пользователя.

Таким образом, при подходящей генерации JWT-токенов нижестоящие микросервисы могут получить всю необходимую для подтверждения личности пользователя информацию, а также дополнительные сведения, такие как группы или роли, в которых находится пользователь. Подлинность этого токена также может быть проверена простой проверкой подписи JWT-токена. По сравнению с предыдущими решениями в данной области (такими как вложенные утверждения SAML), JWT-токены значительно упростили процесс децентрализации авторизации в микросервисной архитектуре.

Проблемы

При использовании JWT-токенов есть несколько проблем, которые стоит иметь в виду. Первая — ключи. В случае подписанных JWT-токенов для проверки подписи получателю JWT-токена потребуется некоторая информация, которая должна быть передана отдельно, — обычно это открытый ключ. В этом случае могут возникнуть все проблемы управления ключами. Как микросервис получает открытый ключ и что произойдет, если потребуется изменить его? Vault — пример инструмента, который может быть использован микросервисом для извлечения (и обработки ротации) открытых ключей, и он предназначен для работы в сильно распределенной среде. Конечно, вы могли бы просто жестко закодировать открытый ключ в файле конфигурации для принимающего микросервиса, но тогда у вас возникла бы проблема с изменением открытого ключа.

Вторая проблема — определение срока действия токена. Это может быть непростой задачей, если речь идет о длительном времени обработки. Рассмотрим ситуацию с размещением заказа покупателем. Это действие запускает набор асинхронных процессов. Их выполнение может занять часы, если не дни, без какого-либо последующего участия клиента (получение оплаты, отправка уведомлений по электронной почте, упаковка и отправка товара и т. д.). В таком случае вам тоже потребуется сгенерировать токен с соответствующим сроком действия? Вопрос здесь в том, в какой момент наличие токена с длительным сроком службы становится более проблематичным, чем отсутствие токена. Я общался с несколькими командами, занимавшимися данной проблемой. Некоторые создали специальный токен, предназначенный для работы только в этом конкретном контексте. Другие просто перестали использовать его в определенной точке потока. Я еще не рассмотрел достаточно примеров этой проблемы, чтобы определить правильное решение в такой ситуации, но это вопрос, о котором следует знать.

Наконец, в некоторых ситуациях вам может понадобиться так много информации в JWT-токене, что сам размер токена становится проблемой. Такие ситуации редки, но встречаются. Несколько лет назад я беседовал с одной командой

об использовании токена для управления авторизацией определенного аспекта ее системы, которая занималась управлением правами на музыку. Логика вокруг этого была невероятно сложной: мой клиент выяснил, что для любого указанного трека потенциально может потребоваться до 10 000 записей в токене, чтобы обработать различные сценарии. Однако мы поняли, что по крайней мере в данной области только один конкретный вариант использования требовал такого большого объема информации, в то время как основная часть системы могла бы обойтись простым токеном с меньшим количеством полей. В той ситуации имело смысл по-другому работать с более сложным процессом авторизации управления правами: использовать JWT-токен для начальной «простой» авторизации, а затем выполнить последующий поиск в хранилище данных для получения дополнительных полей по мере необходимости. Это означало, что основная часть системы могла просто работать с токенами.

Резюме

Надеюсь, у меня получилось объяснить, что создание защищенной системы — это не просто выполнение какого-то одного сценария. Безопасность требует целостного видения вашей системы с использованием моделирования угроз, чтобы понять, какие средства контроля безопасности необходимо внедрить.

Когда речь заходит об этих средствах контроля, для построения безопасной системы необходимо их сочетание. Глубокая оборона означает не только наличие множества средств защиты, но и многогранный подход к созданию более безопасной системы.

Мы снова возвращаемся к основной теме книги: декомпозиция системы на более детализированные сервисы дает гораздо больше возможностей для решения проблем. Наличие микросервисов не только может потенциально снизить влияние любого конкретного нарушения безопасности, но и позволяет рассмотреть компромиссы между издержками при использовании более сложных и безопасных подходов, когда данные конфиденциальны, и более легким подходом, когда риски ниже.

Для более глубокого погружения в тему безопасности приложений я рекомендую книгу «Безопасность разработки в Agile-проектах»[1].

Далее мы рассмотрим, как можно сделать системы более надежными, поскольку мы переходим к теме отказоустойчивости.

---

[1] *Белл Л., Брантон-Сполл М. и др.* Безопасность разработки в Agile-проектах.

ГЛАВА 12

Отказоустойчивость

Поскольку ПО становится все более важной частью нашей жизни, необходимо постоянно улучшать его качество. Сбой программного обеспечения может оказать значительное влияние на людей, даже если оно не подпадает под категорию «критически важных для безопасности», как в случае с системами управления самолетами. Во время пандемии COVID-19 такие услуги, как онлайн-покупки продуктов, превратились из удобства в необходимость для многих людей, не имевших возможности покинуть свои дома.

На этом фоне перед нами часто ставится задача создания все более *надежного* ПО. Ожидания пользователей от программного обеспечения изменились. Все реже приходится поддерживать ПО только в рабочее время, а также снижается терпимость к простоям из-за технического обслуживания.

Как уже говорилось в начале книги, существует множество причин, по которым организации по всему миру выбирают микросервисные архитектуры. Но для многих основной причиной является перспектива повышения *отказоустойчивости* предлагаемых ими услуг.

Прежде чем перейти к деталям отказоустойчивости, важно сделать шаг назад и рассмотреть, что же это такое на самом деле. Оказывается, что, когда дело доходит до повышения отказоустойчивости нашего ПО, внедрение микросервисной архитектуры — это только часть головоломки.

Что такое отказоустойчивость

Мы используем термин «отказоустойчивость» по-разному, в зависимости от контекста. Это может привести к путанице, а также к тому, что мы будем слишком узко мыслить. За пределами ИТ существует более широкая сфера инженерии надежности, которая рассматривает концепцию отказоустойчивости применительно к множеству систем — от пожаротушения до управления воздушным движением, биологических систем и операционных залов. Опираясь на опыт в этой области, Дэвид Д. Вудс попытался классифицировать различные

аспекты отказоустойчивости, чтобы помочь нам более широко взглянуть на это понятие[1]. Перед вами четыре разработанные им концепции.

Надежность

Способность поглощать ожидаемые возмущения.

Восстановление

Способность восстанавливаться после травмирующего события.

Стабильная расширяемость

Насколько хорошо мы справляемся с неожиданной ситуацией.

Непрерывная адаптивность

Способность постоянно адаптироваться к меняющимся условиям, заинтересованным сторонам и требованиям.

Рассмотрим каждую из этих концепций по очереди и то, как они могут (или не могут) воплотиться в наш мир построения микросервисных архитектур.

Надежность

Надежность — это концепция, согласно которой мы встраиваем механизмы для решения ожидаемых проблем в свое ПО и процессы. У нас есть понимание видов возможных возмущений, и мы принимаем меры, чтобы при возникновении этих проблем наша система могла с ними справиться. В контексте микросервисной архитектуры есть целый ряд возмущений, которых стоит ожидать: хост вышел из строя, время ожидания сетевого подключения истекло, микросервис недоступен. Повысить надежность архитектуры можно несколькими способами, например автоматически развернуть запасной хост, выполнить повторную попытку подключения или обработать сбой определенного микросервиса стабильным образом.

Однако надежность выходит за рамки ПО. Она может применяться и к людям. Если у вас есть один человек, отвечающий за ПО, что произойдет, если он заболеет или будет недоступен во время инцидента? Это довольно легко предусмотреть, и решением может быть наличие резервного сотрудника.

Надежность по определению требует предварительных знаний — мы принимаем меры для борьбы с известными проблемами. Эти знания могут быть основаны на предвидении: мы могли бы использовать свое понимание создаваемой компьютерной системы, ее вспомогательных служб и наших сотрудников, чтобы предугадать, что может пойти не так. Но надежность также может быть результатом ретроспективного анализа — допустимо повышать ее после того, как произойдет что-то неожиданное. Возможно, мы никогда не рассматривали

[1] *Woods D. D.* Four Concepts for Resilience and the Implications for the Future of Resilience Engineering // Reliability Engineering & System Safety 141 (сентябрь 2015): 5–9, doi.org/10.1016/j.ress.2015.03.018.

тот факт, что наша глобальная файловая система может стать недоступной, или недооценили влияние того, что представители службы поддержки клиентов недоступны в нерабочее время.

Одна из проблем, связанных с повышением надежности системы, заключается в усложнении системы по мере повышения надежности приложения, и мы рискуем получить новые источники проблем. Допустим, вы переносите свою микросервисную архитектуру на Kubernetes, потому что хотите, чтобы она управляла желаемым состоянием рабочих нагрузок микросервиса. В результате вы, возможно, улучшили некоторые аспекты надежности своего приложения, но также ввели новые потенциальные болевые точки. Таким образом, любая попытка повысить надежность приложения должна рассматриваться с точки зрения не только простого анализа затрат и выгод, но и того, довольны ли вы более сложной системой, которую получите в результате этих изменений.

Надежность — это одна из областей, в которой микросервисы дают вам массу возможностей, и многое из того, что последует в данной главе, будет посвящено тому, что вы можете сделать в своем ПО для повышения надежности системы. Просто помните, что это не только один из аспектов отказоустойчивости в целом, но и множество других факторов, не связанных с ПО, которые вам, возможно, придется учитывать.

Восстановление

То, насколько хорошо система восстанавливается после сбоев, — ключевая часть построения отказоустойчивой системы. Слишком часто я вижу, как люди тратят свои время и энергию, чтобы попытаться исключить возможность сбоя, но оказываются совершенно неподготовленными, когда он действительно происходит. Во что бы то ни стало делайте все возможное, чтобы защититься от неприятностей, которые, по вашему мнению, могут произойти, — повышая *надежность* своей системы. Но также стоит понимать, что по мере роста и усложнения системы устранение любой потенциальной проблемы становится непосильной задачей.

Можно улучшить способность восстанавливаться после инцидента, заранее все предусмотрев. Например, наличие резервных копий позволит восстановиться после потери данных (при условии, конечно, что резервные копии проверены!). Улучшение способности восстанавливаться также может включать в себя наличие плана действий, который мы можем выполнить в случае сбоя системы. Понимают ли люди, какова их роль в случае сбоя? Кто будет ответственным за урегулирование ситуации? Как скоро необходимо сообщить пользователям, что происходит? Как мы будем общаться с пользователями? Попытка четко продумать, как справиться с отключением, пока оно продолжается, будет проблематичной из-за присущих ситуации стресса и хаоса.

Наличие согласованного плана действий на случай возникновения подобного рода проблем может помочь эффективнее восстановиться.

Стабильная расширяемость

В случае с восстановлением и надежностью мы в первую очередь имеем дело с ожидаемым. Создаем механизмы для решения проблем, которые можем предвидеть. Но что происходит, если мы не готовы к неожиданностям? В итоге получаем уязвимую систему. По мере приближения к ожидаемым пределам прочности системы все разваливается — мы не в состоянии реагировать адекватно.

Организации с более плоской структурой, где ответственность распределена внутри компании, часто лучше подготовлены к неожиданностям. Если люди ограничены в своих действиях и должны придерживаться строгого набора правил, при возникновении неожиданных ситуаций их способность справляться с непредвиденным будет очень ограничена.

Часто, стремясь оптимизировать свою систему, мы можем в качестве неприятного побочного эффекта увеличить уязвимость системы. Возьмем в качестве примера автоматизацию. Автоматизация позволяет выполнять больше задач с имеющимся количеством людей, но она может привести и к сокращению численности сотрудников, поскольку с ее помощью можно сделать больше, а тратить меньше. Однако это сокращение штата может быть опасным подводным камнем. Автоматизация не справится с неожиданностью, ведь наша способность стабильно расширять систему, чтобы справляться с подобными ситуациями, зависит от наличия на месте людей с нужными навыками, опытом и ответственностью.

Непрерывная адаптивность

Непрерывная адаптивность требует от нас не быть самодовольными. Как сказал Дэвид Вудс, «независимо от того, насколько хорошо мы справлялись раньше и насколько успешными мы были, будущее может быть другим, а мы — неподготовленными. Наши системы могут оказаться ненадежными и хрупкими перед лицом нового будущего»[1]. То, что мы еще не пострадали от катастрофического отключения электроэнергии, не означает, что этого не может произойти. Нам необходимо бросить вызов самим себе, чтобы убедиться, что мы постоянно адаптируем свою деятельность в качестве организации для обеспечения будущей отказоустойчивости. Если все сделано правильно, такая концепция, как хаос-инжиниринг, которую мы кратко рассмотрим позже, может стать полезным инструментом, помогающим создать непрерывную адаптивность.

Непрерывная адаптивность часто требует более целостного взгляда на систему. Парадоксально, но именно в этом случае стремление к созданию небольших автономных команд с повышенной ответственностью на местном уровне может привести к тому, что мы упустим из виду общую картину. Как описывается в гла-

[1] *Bloomberg J.* Innovation: The Flip Side of Resilience // Forbes. 23 сентября 2014 года. https://oreil.ly/avSmU.

ве 15, когда речь заходит об организационной динамике, существует баланс между глобальной и локальной оптимизацией и он не статичен. В той главе мы рассмотрим роль целенаправленных, потоковых команд, владеющих микросервисами, необходимыми им для предоставления функциональности, ориентированных на пользователя и получающих для этого повышенный уровень ответственности. Мы также поговорим о роли команд поддержки, сопровождающих потоковые команды, и то, какую роль могут сыграть команды поддержки в достижении непрерывной адаптивности на организационном уровне.

Создание культуры производства, в которой приоритет отдается предоставлению людям возможности свободно обмениваться информацией, не опасаясь наказания, жизненно важно для поощрения обучения после инцидента. Для изучения таких сюрпризов и извлечения ключевых выводов требуется время, энергия и люди — все то, что сократит ресурсы, доступные вам для предоставления функций в краткосрочной перспективе. Решение использовать непрерывную адаптивность отчасти связано с поиском точки равновесия между краткосрочными результатами и долгосрочной адаптивностью.

Работа над непрерывной адаптивностью означает, что вы стремитесь открыть для себя новое и неизведанное. Такой процесс требует постоянных, а не разовых вложений — в этой теме очень важен термин «непрерывная». Речь идет о том, чтобы сделать непрерывную адаптивность ключевой частью вашей организационной стратегии и культуры.

Микросервисная архитектура

Как обсуждалось ранее, можно увидеть, как микросервисная архитектура может помочь достичь свойства надежности, но этого недостаточно, если вам нужна *отказоустойчивость*.

В более широком смысле способность обеспечивать отказоустойчивость — это свойство не самого ПО, а людей, создающих и эксплуатирующих систему. Учитывая направленность книги, многое из того, что последует в текущей главе, будет сосредоточено в первую очередь на том, что микросервисная архитектура способна дать с точки зрения отказоустойчивости, которая почти полностью ограничивается повышением *надежности* приложений.

Сбои повсюду

Мы понимаем, что все может пойти не так. Жесткие диски могут выйти из строя. Программное обеспечение может дать сбой. И как любой, кто читал статью https://oreil.ly/aYIjx, могу сказать вам, что сеть ненадежна. Мы можем сделать все, что в наших силах, чтобы попытаться ограничить причины сбоев, но в определенном масштабе сбои становятся неизбежными. Жесткие диски,

например, сейчас надежнее, чем когда-либо, но рано или поздно они сломаются. Чем больше у вас жестких дисков, тем выше вероятность отказа отдельного устройства в любой конкретный день. В масштабах компании отказ становится статистической вероятностью.

Даже для тех из нас, кто не мыслит в экстремальных масштабах, принятие возможности неудачи сделает только лучше. Например, если мы можем корректно справиться с отказом микросервиса, то из этого следует, что мы также способны выполнить обновление сервиса на месте, поскольку с запланированным отключением справиться гораздо легче, чем с незапланированным.

Мы также можем потратить немного меньше времени на попытки остановить неизбежное и немного больше на поиски стабильного решения. Удивительно, как много организаций внедряют процессы и средства контроля, чтобы попытаться предотвратить возникновение сбоев, но практически не задумываются о том, чтобы в первую очередь облегчить восстановление после сбоя. Понимание, что может привести к сбою, станет ключом к повышению *надежности* системы.

Мысль о том, что все может и не сработать, заставляет по-другому думать о способах решения проблем. Помните историю с серверами Google, описанную в главе 10? Системы Google были построены таким образом, что сбой компьютера не приводил к прерыванию обслуживания клиента, что повышало *надежность* системы в целом. Компания Google развивается в этом направлении, пытаясь повысить надежность своих серверов другими способами. Там решили, что каждому серверу необходим собственный локальный источник питания, чтобы гарантировать работоспособность, если в дата-центре произойдет сбой[1]. В главе 10 описывалось, что жесткие диски на этих серверах крепились с помощью липучек, а не винтов, чтобы упростить замену в случае поломки. Это дало возможность сотрудникам Google быстро запустить компьютер при сбое диска и, в свою очередь, помогло этому компоненту системы восстановиться более эффективно.

Итак, позвольте мне повторить: при увеличении масштаба системы, даже если вы покупаете самое лучшее и дорогое оборудование, полностью избежать отказов не выйдет. Следовательно, вам нужно предположить, что может произойти сбой. Если вы встроите это мышление во все, что делаете, и спланируете действия на случай непредвиденного отказа, то сможете пойти на осознанный компромисс. Если вам известно, что ваша система может справиться с тем фактом, что сервер может и будет выходить из строя, возможно, будет снижаться отдача от траты все бóльших средств на отдельные машины. Вместо этого гораздо целесообразнее иметь много дешевых машин (возможно, с использованием дешевых компонентов и неких липучек!), как у Google.

[1] См.: Google Uncloaks Once-Secret Server (https://oreil.ly/k7b3h) Стивена Шенкленда для получения дополнительной информации по этому вопросу, включая интересный обзор, почему Google считает, что этот подход может быть лучше традиционных систем с ИБП.

Слишком много — это сколько?

Мы затронули тему кросс-функциональных требований в главе 9. Их понимание предполагает учет таких аспектов, как долговечность данных, доступность сервисов, пропускная способность и приемлемая задержка операций. Многие методы, рассмотренные в текущей главе, рассказывают о подходах к выполнению этих требований, но только вы точно знаете, какими они могут быть. Поэтому, читая дальше, помните о требованиях к своей системе.

Наличие системы автоматического масштабирования, способной реагировать на повышенную нагрузку или отказ отдельных узлов, может стать гигантским преимуществом. Но это может быть и излишним для системы отчетности, которая должна запускаться только два раза в месяц, где отключение на день или два не так уж и страшно. Аналогичным образом поиск путей непрерывного развертывания для исключения прерывания работы сервиса может иметь смысл для системы электронной коммерции в Интернете, но для внутрисетевой корпоративной базы знаний это, вероятно, будет лишним.

То, сколько сбоев можно допустить или насколько быстрой должна быть система, зависит от пользователей. Эта информация, в свою очередь, поможет понять, какие технические методы будут иметь для вас наибольший смысл. Тем не менее пользователи не всегда могут четко сформулировать свои требования. Поэтому нужно задавать вопросы, чтобы извлечь нужную информацию и помочь им понять относительные затраты на предоставление различных уровней обслуживания.

Как упоминалось ранее, эти кросс-функциональные требования варьируются от сервиса к сервису, но я бы предложил определить некоторые общие кросс-функциональные требования для конкретных случаев использования. Когда дело доходит до рассмотрения вопроса, следует ли и как масштабировать систему, чтобы лучше справляться с нагрузкой или сбоями, начните с попытки понять следующие требования.

Время отклика/задержка

Сколько времени должны занимать различные операции? Может быть полезно измерить этот процесс с разным количеством пользователей, чтобы понять, как увеличение нагрузки повлияет на время отклика. Учитывая природу сетей, у вас всегда будут отклонения, поэтому может быть полезно установить целевые показатели для заданного процентиля отслеживаемых ответов. Цель также должна включать количество одновременных подключений/пользователей, которые, как ожидается, будет обрабатывать ваше ПО. Поэтому можно сказать: «Ожидается, что время отклика сайта 2 секунды будет равно 90-му процентилю при обработке 200 одновременных подключений в секунду».

Доступность

Можно ли ожидать перебоев в работе сервиса? Считается ли ваша система сервисом 24/7? Некоторым людям нравится смотреть на периоды приемлемого

простоя при оценке доступности, но насколько это полезно для тех, кто вызывает ваш сервис? Либо я могу рассчитывать на то, что ваш сервис ответит, либо нет. Измерение периодов простоя фактически более полезно с точки зрения исторической отчетности.

Долговечность данных

Насколько допустима потеря данных? Как долго следует их хранить? Весьма вероятно, что ответы на эти вопросы будут меняться в каждом конкретном случае. Например, можно хранить логи сеансов пользователей в течение года или меньше, чтобы сэкономить место, но записи о финансовых транзакциях, возможно, потребуется хранить в течение многих лет.

Принятие этих идей и формулирование их в качестве целей SLO, рассмотренных в главе 10, может стать хорошим способом закрепить эти требования в качестве основной части процесса доставки вашего ПО.

Снижение функциональности

Важной частью построения отказоустойчивой системы, особенно когда функциональные возможности распределены по нескольким различным микросервисам, которые могут то работать, то не работать, является способность безопасно снижать функциональность. Представим себе стандартную веб-страницу на нашем сайте электронной коммерции. Чтобы объединить различные части этого веб-сайта, нам может понадобиться несколько микросервисов. Один может отображать подробную информацию о товаре, выставляемом на продажу, другой — цену и уровень запасов. Вероятно, надо будет также показывать содержимое корзины покупок, что может выполняться еще одним микросервисом. Если один из этих сервисов выходит из строя и это приводит к тому, что вся веб-страница становится недоступной, то, возможно, мы создали менее отказоустойчивую систему, чем та, для которой требуется доступность только одного сервиса.

Нам необходимо понять влияние каждого сбоя и решить, как грамотно снизить функциональность. С точки зрения бизнеса требуется, чтобы рабочий процесс обработки заказов был максимально устойчив и мы с радостью приняли бы некоторое ухудшение функциональности, взамен на гарантии, что система все еще работает. Если сведения об уровне запасов недоступны, можно принять решение о продолжении продажи и проработать детали позже. Если микросервис корзины покупок недоступен, у нас, вероятно, возникнут большие проблемы, но все равно можно отобразить веб-страницу с перечнем товаров. Возможно, мы просто спрячем корзину покупок или заменим ее значком с надписью «Скоро вернемся!».

С монолитным однопроцессным приложением не требуется принимать много решений. Работоспособность системы в этом контексте в некоторой

степени бинарная: процесс либо работает, либо нет. Но при микросервисной архитектуре нужно учитывать гораздо больше нюансов. Правильное решение в любой ситуации часто не будет техническим. Мы можем знать, что технически возможно сохранять работоспособность, когда корзина не работает, но пока мы не поймем бизнес-контекст — не поймем, какие действия следует предпринять. Например, возможно, мы закроем весь сайт, но по-прежнему позволим людям просматривать каталог товаров или заменим часть UI, содержащую элемент управления корзиной, номером телефона для размещения заказа. Но для каждого ориентированного на клиента интерфейса, использующего несколько микросервисов, или для каждого микросервиса, зависящего от нескольких нижестоящих сервисов, вам нужно спросить себя: «Что произойдет, если это не удастся?» — и знать, что делать.

Если задуматься о критичности каждой из наших возможностей с точки зрения кросс-функциональных требований, мы будем гораздо лучше знать, что можно сделать. Теперь рассмотрим некоторые задачи, которые стоит решить с технической точки зрения, чтобы убедиться, что при возникновении сбоя мы справимся с ним корректно.

Шаблоны стабильности

Есть несколько шаблонов, которые можно использовать, чтобы предотвратить неприятные побочные эффекты. Очень важно понимать эти идеи, и вам следует серьезно подумать об использовании их в своей системе, чтобы гарантировать, что один плохой гражданин не разрушит весь мир у вас на глазах. Сейчас мы рассмотрим несколько ключевых мер безопасности, которые вам следует принять во внимание, но прежде, чем мы это сделаем, я хотел бы поделиться краткой историей, описывающей, что может пойти не так.

Много лет назад я был техническим руководителем проекта AdvertCorp. Организация AdvertCorp (название и данные компании изменены, чтобы защитить невиновных!) размещала онлайн-объявления через очень популярный веб-сайт. Сам веб-сайт обрабатывал довольно большие объемы и приносил хороший доход бизнесу. Перед проектом, над которым я работал, стояла задача объединить ряд существующих сервисов, использующихся для предоставления схожей функциональности для различных типов рекламы. Существующая функциональность постепенно переносилась в новую систему, которую мы создавали, при этом некоторая часть по-прежнему отображалась в старых сервисах. Чтобы сделать переход прозрачным для конечного потребителя, мы перехватывали все вызовы на различные типы объявлений в нашей новой системе и перенаправляли их на старые системы, где это необходимо, как показано на рис. 12.1. На самом деле это пример шаблона «Душитель», который мы кратко обсудили в разделе «Полезные шаблоны декомпозиции» главы 3.

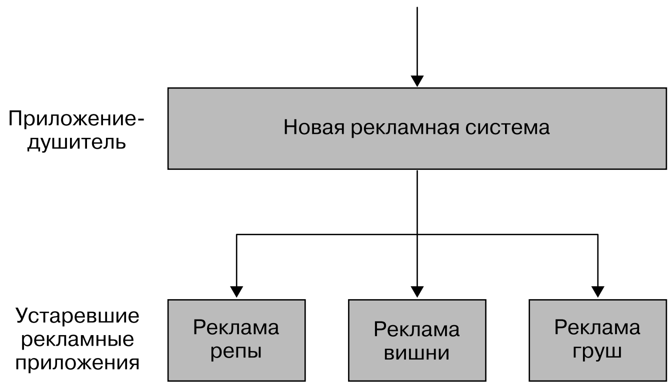


Рис. 12.1. Шаблон «Душитель», используемый для направления вызовов на устаревшие системы

Мы только что перенесли самый массовый и приносящий наибольший доход продукт в новую систему, но большая часть остальной рекламы по-прежнему обслуживалась рядом старых приложений. С точки зрения количества поисковых запросов и денег, заработанных этими приложениями, существовал очень длинный шлейф информации — многие из этих неактуальных приложений получали маленькие объемы трафика и приносили небольшие суммы дохода. Новая система работала уже некоторое время и вела себя очень хорошо, справляясь с немалой нагрузкой. В то время мы, должно быть, обрабатывали около 6000–7000 запросов в секунду в пике, и хотя большая их часть была очень сильно кэширована обратными прокси, расположенными перед серверами приложений, поиск продуктов (наиболее важный аспект сайта) в основном не кэшировался и требовал полного прохода сервера туда и обратно.

Однажды утром, как раз перед достижением нашего ежедневного полуденного пика, система начала работать медленно, а затем стала давать сбои. У нас был некоторый уровень мониторинга нового основного приложения, достаточный, чтобы сказать, что каждый из узлов приложения достиг 100 % загрузки процессора, что значительно превышает нормальный уровень даже на пике. За короткий промежуток времени весь сайт вышел из строя.

Нам удалось отследить виновника и вернуть сайт в рабочее состояние. Оказалось, что это одна из нижестоящих рекламных систем, которая в этом анонимном примере будет отвечать за рекламу репы. Рекламный сервис репы, один из старейших и наименее активно поддерживаемых сервисов, начал отвечать очень медленно. Очень медленное реагирование — один из худших режимов сбоя, с которым можно столкнуться. Если система просто не работает, вы обнаружите это довольно быстро. А когда она работает *медленно*, вы в итоге ждете некоторое время, прежде чем сдаться, — этот процесс ожидания может затормозить работу всей системы, вызвать конфликт ресурсов и, как это произошло в нашем случае, привести к каскадному сбою. Но какова бы ни была причина сбоя, мы создали

систему, уязвимую для последующих подобных проблем. Нижестоящий сервис, над которым у нас практически отсутствовал контроль, смог вывести из строя всю нашу систему.

Пока одна команда изучала проблемы с системой рекламы репы, остальные начали разбираться, что пошло не так в нашем приложении. Мы обнаружили несколько проблем, которые описаны на рис. 12.2. Мы использовали пул HTTP-соединений для обработки наших нижестоящих подключений. У потоков в самом пуле были настроены тайм-ауты на продолжительность ожидания при выполнении нисходящего HTTP-вызова, и это было правильно. Проблема заключалась в том, что все исполнители брали тайм-аут из-за медленного сервиса ниже по потоку. Пока они ждали, в пул поступало еще больше запросов, требующих потоков исполнителей. При отсутствии доступных исполнителей эти запросы сами зависали. Оказалось, что библиотека пула подключений, которую мы использовали, действительно содержала тайм-аут для ожидания исполнителей, но *по умолчанию он был отключен*! Это привело к огромному накоплению заблокированных потоков. Наше приложение обычно поддерживало до 40 одновременных подключений в любой момент времени. В течение 5 минут эта ситуация привела к достижению пика примерно в 800 подключений, что привело к выходу системы из строя.

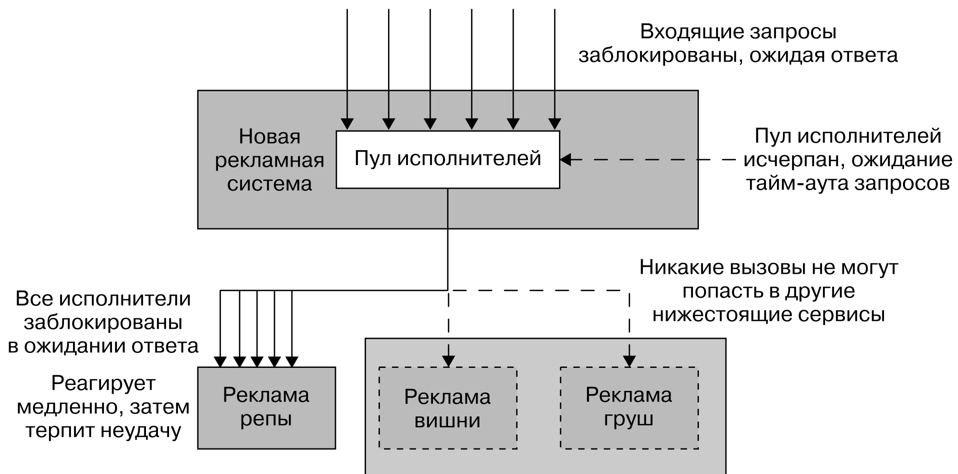


Рис. 12.2. Краткое описание проблем, вызвавших отключение

Хуже всего, что сервис, с которым мы общались, представлял функциональность, которой пользовалось менее 5 % нашей клиентской базы, а доход от него был еще меньше. Если разобраться, мы на собственном горьком опыте убедились, что с системами, работающими просто медленно, *гораздо* сложнее работать, чем с теми, которые быстро выходят из строя. В распределенной системе задержка убивает.

Даже если бы в пуле были правильно установленные тайм-ауты, мы также совместно использовали бы один пул HTTP-соединений для всех исходящих запросов. Это означало, что один медленный сервис, стоящий ниже в потоке, мог сам по себе исчерпать количество доступных исполнителей, даже если все остальное было исправно. Наконец из-за частых тайм-аутов и ошибок стало ясно, что рассматриваемый нижестоящий сервис не был исправен, но, несмотря на это, мы продолжали отправлять трафик тем же путем. В нашей ситуации это означало, что мы фактически усугубляли и без того плохую ситуацию, поскольку у нижестоящего сервиса не было шансов восстановиться. В итоге мы внедрили три исправления, чтобы избежать повторения ситуации: откорректировали использование *тайм-аутов*, внедрили *переборки* для разделения различных пулов соединений и внедрили *автоматический выключатель*, чтобы избежать изначальной отправки вызовов в нерабочую систему.

Тайм-ауты

Тайм-ауты легко упустить из виду, но в распределенной системе важно использовать их правильно. Как долго стоит ждать, прежде чем отказаться от вызова нижестоящего сервиса? Если ждать слишком долго для принятия решения, что вызов не удался, может замедлиться работа всей системы. Если же тайм-аут наступит слишком быстро, то вызов, который мог сработать, будет считаться неудачным. Если у вас вообще нет тайм-аутов, то выход из строя нижестоящего сервиса может привести к зависанию всей системы.

В случае с AdvertCorp было две проблемы, связанные с тайм-аутом. Во-первых, у нас был пропущенный тайм-аут в пуле HTTP-запросов. Это означало, что при запросе исполнителя для выполнения последующего HTTP-запроса поток запроса блокировался, пока исполнитель не станет доступным. Во-вторых, когда у нас наконец появился HTTP-исполнитель, доступный для отправки запроса в рекламную систему репы, мы слишком долго ждали, прежде чем отказаться от вызова. Итак, как показано на рис. 12.3, нужно было добавить один новый тайм-аут и изменить существующий.

Тайм-ауты для нижестоящих HTTP-запросов были установлены на 30 секунд, поэтому ответа от системы репы требовалось ждать 30 секунд, прежде чем отказаться от запроса. Проблема в том, что в более широком контексте, в котором был сделан этот вызов, столь долгое ожидание не имело смысла. Объявления, связанные с репой, запрашивались в результате того, что один из пользователей просматривал веб-сайт с помощью браузера. Даже когда это происходило, никто не ждал 30 секунд для загрузки страницы. Подумайте о том, что произойдет, если веб-страница не загрузится через 5, 10 или, возможно, 15 секунд. Что вы будете делать? Вы обновите страницу. Итак, мы ждали ответа рекламной системы репы 30 секунд, но задолго до этого первоначальный запрос перестал

быть действительным, поскольку пользователь только что обновил страницу, что вызвало дополнительный входящий запрос. Это, в свою очередь, привело к появлению еще одного запроса для рекламной системы и т. д.

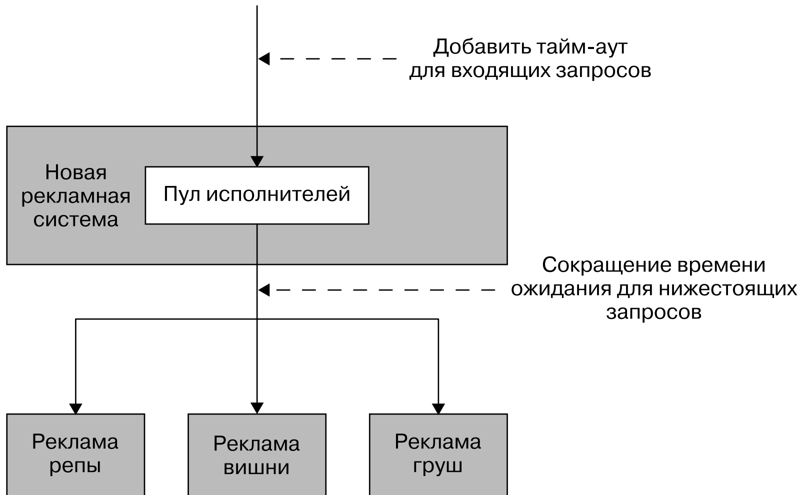


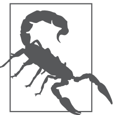
Рис. 12.3. Изменение тайм-аутов в системе AdvertCorp

Рассматривая нормальное поведение рекламной системы репы, можно заметить, что обычно мы ожидаем ответа менее чем одну секунду, поэтому ожидание 30 секунд было излишним. Кроме того, у нас была цель отобразить страницу пользователю в течение 4–6 секунд. Исходя из этого, мы сделали тайм-аут гораздо более агрессивным, установив его равным 1 секунде. Мы также установили тайм-аут, равный 1 секунде на ожидание доступности HTTP-исполнителя. Это означало, что в худшем случае ожидать получения информации от системы репы надо было около 2 секунд.

Тайм-ауты невероятно полезны. Установите тайм-ауты для всех внепроцессных вызовов и укажите их по умолчанию для всех элементов системы. Записывайте возникновение тайм-аутов в логи, смотрите, что происходит, и меняйте их соответствующим образом. Посмотрите на «нормальное» время отклика для нижестоящих сервисов и используйте его, чтобы определить порог тайм-аута.

Установка тайм-аута для одного вызова сервиса может оказаться недостаточной. Что произойдет, если этот тайм-аут происходит как часть более широкого набора операций, от которых, возможно, стоит отказаться еще до того, как он наступит? В случае с AdvertCorp, например, нет смысла ждать последних цен

на репу, если есть большая вероятность, что пользователь уже прервал запрос. В такой ситуации может иметь смысл установить тайм-аут для всей операции и отказаться от нее, если он будет превышен. Чтобы это сработало, время, оставшееся для операции, должно быть передано вниз по потоку. Например, если общая операция по отображению страницы должна была завершиться в течение 1000 миллисекунд и к тому времени, когда мы обратились к нижестоящему сервису рекламы репы, уже прошло 300 миллисекунд, необходимо будет убедиться, что ожидание завершения остальных вызовов не превышает 700 миллисекунд.



Не думайте только о тайм-ауте для отдельного вызова сервиса — также подумайте о тайм-ауте для всей операции и прервите ее, если общее время ожидания превышено.

Повторные попытки

Некоторые проблемы с нисходящими вызовами носят временный характер. Пакеты могут быть перепутаны, или у шлюзов может произойти странный скачок нагрузки, вызывающий тайм-аут. Нередко повторный вызов может иметь большой смысл. Как часто вы обновляли не загружающуюся веб-страницу только для того, чтобы убедиться, что она работает? Это повторная попытка в действии.

Может быть полезно подумать, какого рода сбои при последующих вызовах следует повторять. Например, при использовании такого протокола, как HTTP, можно получить обратно некоторую полезную информацию в кодах ответов, которая поможет определить, оправдана ли повторная попытка. Если вы получили ответ `404 Not Found`, повторная попытка вряд ли будет хорошей идеей. С другой стороны, ошибки `503 Service Unavailable` или `504 Gateway Time-out` могут рассматриваться как временные ошибки, и повторный запрос будет оправдан.

Скорее всего, перед повторной попыткой потребуется немного подождать. Если первоначальный тайм-аут или ошибка были вызваны тем, что нижестоящий микросервис находился под нагрузкой, то бомбардировка его дополнительными запросами может оказаться плохой идеей.

Если вы собираетесь повторить попытку, необходимо принять это во внимание при определении порога тайм-аута. Если пороговое время ожидания для последующего вызова установлено равным 500 миллисекундам, но вы допускаете до трех повторных попыток с интервалом 1 секунда между каждой повторной попыткой, то в конечном счете вам может потребоваться ждать до 3,5 секунды, прежде чем отказаться от запроса. Как упоминалось ранее, наличие общего времени операции может быть полезной идеей. Вы можете не решиться на третью (или даже вторую) попытку, если уже превысили общий порог тайм-аута. С другой стороны, если это происходит как часть операции, не связанной с пользователем, более длительное ожидание выполнения чего-либо может быть вполне приемлемым.

Переборки

В книге «Release It!»[1] Майкл Нейгард вводит концепцию *переборки* как способа защитить себя от неудачи. В судоходстве переборка — это часть судна, которая может быть герметизирована для защиты остальной части судна. Так что, если на корабле появится течь, вы можете закрыть двери в переборке, потеряв часть корабля, но сохранив при этом сам корабль.

С точки зрения архитектуры ПО можно рассмотреть множество различных переборок. Возвращаясь к моему собственному опыту работы с AdvertCorp, мы фактически упустили шанс реализовать переборку в отношении нижестоящих вызовов. Мы должны были использовать разные пулы соединений для каждого нисходящего подключения. Таким образом, если один пул подключений будет исчерпан, это не повлияет на другие соединения, как показано на рис. 12.4.

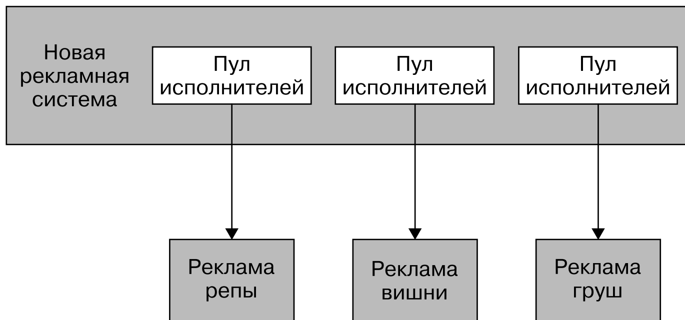


Рис. 12.4. Использование пула подключений для каждого сервиса ниже по потоку для обеспечения переборок

Разделение ответственности также может быть способом реализации переборок. Разделяя функциональность на отдельные микросервисы, мы снижаем вероятность того, что сбой в одной области повлияет на другую.

Посмотрите на все аспекты своей системы, которые могут сбоить, как внутри ваших микросервисов, так и между ними. У вас есть переборки в этих местах? Я бы предложил начать по крайней мере с отдельных пулов подключений для каждого нисходящего соединения. Однако можно пойти дальше и попробовать использовать автоматические выключатели, о которых мы поговорим чуть позже.

Во многих отношениях переборки будут наиболее важными из рассматриваемых до сих пор моделей. Тайм-ауты и автоматические выключатели помогают высвободить драгоценные ограниченные ресурсы, но переборки

[1] *Нейгард М.* Release It! Проектирование и дизайн ПО для тех, кому не все равно. — Питер, 2016.

изначально не допустят ограниченности ресурсов. Они также могут дать возможность отклонять запросы при определенных условиях, чтобы гарантировать отсутствие насыщения ресурсов — явление, известное как *сброс нагрузки*. Иногда отклонение запроса — лучший способ предотвратить перегрузку системы и не превращать ее в узкое место для множества вышестоящих сервисов.

Автоматические выключатели

В вашем собственном доме автоматические выключатели существуют для защиты электрических устройств от скачков напряжения. Если происходит скачок напряжения, срабатывает автоматический выключатель, защищая вашу дорогую бытовую технику. Его можно также отключить вручную, чтобы обесточить часть дома для безопасной работы с электрической проводкой. В другом примере из книги «Release It!» Нейгард показывает, как та же идея может творить чудеса в качестве механизма защиты нашего ПО.

Можно рассматривать автоматические выключатели как автоматический механизм для герметизации переборки не только для защиты потребителя от проблем, возникающих в дальнейшем, но и для потенциальной защиты нижестоящего сервиса от дополнительных вызовов, способных оказать неблагоприятное воздействие. Учитывая опасность каскадного сбоя, я бы рекомендовал установить автоматические выключатели для всех синхронных вызовов, идущих по потоку. Вам также не нужно писать свой собственный вариант выключателя. За годы, прошедшие с момента написания первого издания, реализации автоматических выключателей стали широкодоступными.

Возвращаясь к AdvertCorp, рассмотрим проблему, с которой мы столкнулись, когда система репы реагировала очень медленно, прежде чем в итоге выдать ошибку. Даже при правильном расчете тайм-аутов нам пришлось бы долго ждать возникновения ошибки. А затем мы повторяли бы попытку при следующем поступлении запроса и ждали бы. Мало того, что нижестоящий сервис работал со сбоями, это также замедляло работу всей системы.

При использовании автоматического выключателя после сбоя определенного количества запросов к нижестоящему ресурсу (из-за ошибки или тайм-аута) автоматический выключатель срабатывает. Все последующие запросы, проходящие через него, быстро завершаются сбоем, пока выключатель находится в «разомкнутом» состоянии, как показано на рис. 12.5. Через определенный промежуток времени клиент отправляет несколько запросов, чтобы проверить, восстановился ли нижестоящий сервис, и если он получает достаточно исправных ответов, он сбрасывает автоматический выключатель.

Способ применения автоматического выключателя зависит от того, что означает «неудачный запрос». Когда я внедряю их для HTTP-соединений, я обычно принимаю неудачу за тайм-аут или подмножество кодов возврата HTTP 5XX.

Таким образом, когда у нижестоящего ресурса истекает время ожидания или от него возвращаются ошибки, после достижения определенного порога автоматически прекращается отправка трафика и начинается быстрый выход из строя. И мы можем автоматически начать все сначала, когда вернется вся работоспособность элементов.

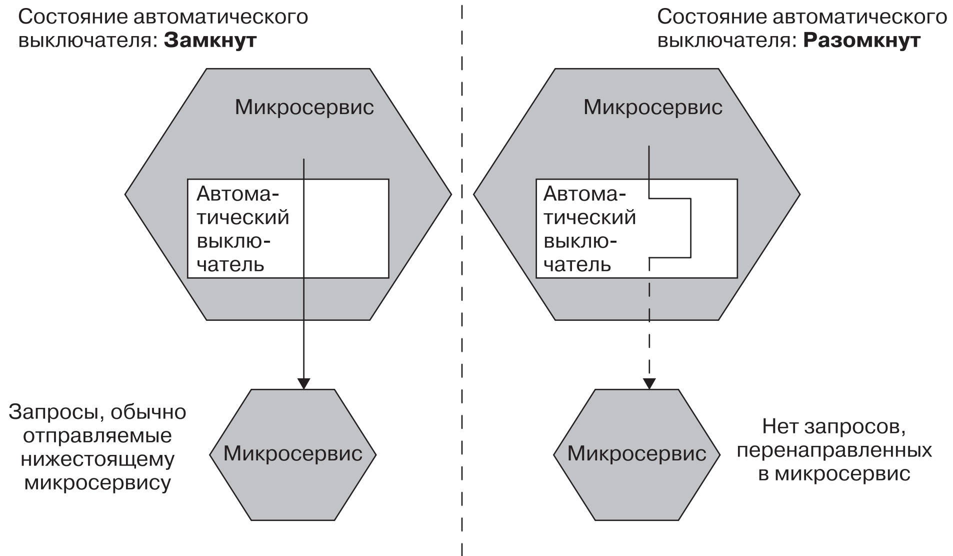


Рис. 12.5. Обзор автоматических выключателей

Правильно все настроить может быть сложновато. Не нужно слишком быстро отключать автоматический выключатель, а также нельзя допустить, чтобы его отключение занимало слишком много времени. Аналогично стоит действительно убедиться, что нижестоящий сервис снова исправен, прежде чем отправлять трафик. Как и в случае с тайм-аутами, я бы выбрал несколько разумных значений по умолчанию и придерживался их везде, а затем изменил бы их для конкретных случаев.

Когда автоматический выключатель сработал, у вас есть несколько вариантов. Например, поставить запросы в очередь и повторить их позже. В некоторых случаях это может быть уместно, особенно если вы выполняете какую-то работу в рамках асинхронного задания. Однако, если этот вызов выполняется как часть цепочки синхронных вызовов, вероятно, лучше быстро завершить работу с ошибкой. Это может означать распространение ошибки вверх по цепочке вызовов или едва уловимое снижение функциональности.

В случае AdvertCorp мы обернули нисходящие вызовы к устаревшим системам автоматическими выключателями, как показано на рис. 12.6. Когда они

сработали, мы программно обновили веб-сайт, чтобы показать, что в настоящее время невозможно показывать рекламу, скажем, репы. Мы сохранили работу остальной части веб-сайта и четко сообщили клиентам, что проблема ограничена одной частью нашего продукта, и все это полностью автоматизированным способом.

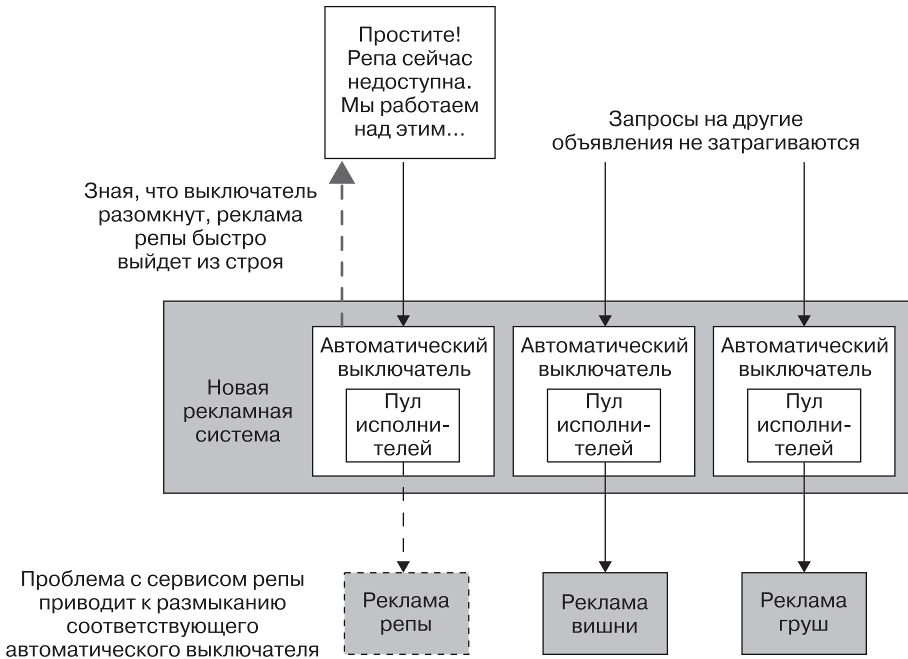


Рис. 12.6. Добавление автоматических выключателей в AdvertCorp

Мы смогли настроить свои автоматические выключатели таким образом, чтобы у нас было по одному для каждой из устаревших систем ниже по потоку, — это хорошо сочеталось с тем, что мы решили использовать разные пулы исполнителей запросов для каждого сервиса ниже по потоку.

Если такой механизм есть (как в случае с автоматами в нашем доме), его можно использовать вручную, чтобы сделать свою работу более безопасной. Например, если мы хотим отключить микросервис в рамках планового технического обслуживания, можно было бы вручную размыкать все автоматические выключатели вышестоящих потребителей, чтобы они быстро выходили из строя, пока микросервис находится в автономном режиме. Как только он вернется к работе, мы сможем замкнуть автоматические выключатели и все должно вернуться в нормальное русло. Следующим разумным шагом может стать написание скрипта процесса ручного размыкания и замыкания

автоматического выключателя в рамках процесса автоматизированного развертывания.

Автоматические выключатели помогают приложению быстро выходить из строя — а быстрый сбой всегда лучше, чем постепенный. Применение автоматических выключателей позволяет системе выйти из строя, прежде чем она начнет тратить драгоценное время (и ресурсы) на ожидание ответа неработоспособного нижестоящего микросервиса. Вместо ожидания попыток использования нижестоящего микросервиса для вызова сбоя можно было бы проверить состояние наших автоматических выключателей раньше. Если микросервис, на который мы будем полагаться как на часть операции, в данный момент недоступен, рекомендуется прервать операцию еще до ее начала.

Изоляция

Чем сильнее связь между микросервисами, тем больше работоспособность одного из них влияет на способность другого выполнять свою работу. Если можно использовать технологию, позволяющую нижестоящему серверу находиться в автономном режиме, например, с помощью промежуточного ПО или какого-либо другого типа системы буферизации вызовов, вышестоящие микросервисы с меньшей вероятностью будут затронуты плановыми или незапланированными отключениями нижестоящих.

Существует еще одно преимущество усиления изоляции между сервисами. Когда они изолированы друг от друга, требуется гораздо меньше взаимодействия между владельцами сервисов. Чем меньше необходимо координаций между командами, тем большей автономией они обладают, поскольку могут более свободно управлять и развивать свои сервисы.

Изоляция также применима к переходу от рассмотрения логических аспектов к физическим. Рассмотрим два микросервиса, кажущихся полностью изолированными друг от друга. Они никак не сообщаются. Проблема с одним из них не должна влиять на другой, верно? Но что, если оба микросервиса запущены на одном хосте и один из микросервисов начинает использовать все ресурсы процессора, вызывая проблемы на этом хосте?

Рассмотрим другой пример. У двух микросервисов есть своя собственная логически изолированная БД. Но оба они развернуты в одной и той же инфраструктуре баз данных. Сбой в этой инфраструктуре повлияет на каждый микросервис.

При рассмотрении вариантов развертывания микросервисов следует стремиться обеспечить определенную степень изоляции сбоев, чтобы избежать подобных проблем. Например, обеспечение работы микросервисов на независимых хостах с их собственной ОС и вычислительными ресурсами будет разумным шагом — это то, чего мы достигаем, когда запускаем экземпляры микросервисов на их собственной виртуальной машине или в контейнере. Однако и такого рода изоляция ведет к повышенным издержкам.

Мы можем более эффективно изолировать свои микросервисы друг от друга, запустив их на разных машинах. Это означает, что нужно больше инфраструктуры и инструментов для управления ею, что влечет за собой прямые издержки, а также усложняет систему, открывая новые пути потенциального отказа. Каждый микросервис может получить свою собственную полностью выделенную инфраструктуру базы данных, но это затрудняет управление. Мы могли бы использовать промежуточное ПО для обеспечения временной развязки между двумя микросервисами, но теперь у нас появляется брокер, требующий сопровождения.

Изоляция, как и многие другие уже рассмотренные методы, может помочь повысить надежность наших приложений, но редко это дается бесплатно. Выбор приемлемого компромисса между изоляцией и затратами вкупе с повышенной сложностью, как и во многих других вещах, может представлять жизненно важное значение.

Избыточность

Наличие большего количества какого-либо ресурса может быть отличным способом повысить надежность компонента. Присутствие более одного человека, знающего, как работает БД, кажется разумным на случай, если кто-то покинет компанию или уедет в отпуск. Наличие более одного экземпляра микросервиса имеет смысл, поскольку это позволяет допускать выход из строя одного из этих экземпляров и при этом сохранить шанс обеспечить требуемую функциональность.

Определение, сколько резервирования требуется и где, будет зависеть от того, насколько хорошо вы понимаете потенциальные режимы отказа каждого компонента, влияние недоступности этой функциональности и стоимость добавления избыточности.

Например, AWS не предоставляет SLA для времени безотказной работы одного экземпляра EC2 (виртуальной машины). Вы должны работать, исходя из предположения, что данный экземпляр может у вас и умереть. Так что есть смысл поддерживать несколько экземпляров. Но будем разбираться дальше. Экземпляры EC2 развертываются в зонах доступности (виртуальные дата-центры), и у вас также нет гарантий относительно наличия одной зоны доступности, а это означает, что желательно разместить второй экземпляр в *другой* зоне, чтобы распределить риски.

Наличие большего количества копий чего-либо может помочь, когда дело доходит до реализации резервирования, но это также полезно, когда речь идет о масштабировании наших приложений для обработки увеличенной нагрузки. В следующей главе мы рассмотрим примеры масштабирования системы и увидим, чем может отличаться масштабирование для резервирования и для нагрузки.

Промежуточное ПО

В подразделе «Брокеры сообщений» главы 5 мы рассмотрели роль промежуточного ПО в виде брокеров сообщений, помогающих реализовать взаимодействия на основе как запроса — ответа, так и событий. Одним из полезных свойств большинства брокеров сообщений стала их способность обеспечивать гарантированную доставку. Вы отправляете сообщение нижестоящей стороне, и брокер гарантирует его доставку с некоторыми оговорками, описанными ранее. Чтобы обеспечить гарантию, ПО брокера сообщений должно реализовать повторные попытки и тайм-ауты от вашего имени, — те же самые операции, которые вам пришлось бы выполнять самостоятельно, но они выполняются в программном обеспечении, написанном экспертами с глубоким пониманием таких вещей. Привлечение умных людей к работе на вас — хорошая идея.

В случае нашего конкретного примера с AdvertCorp использование промежуточного ПО для управления связью «запрос — ответ» с нижестоящей системой репы, возможно, не сильно помогло бы. Мы все равно не получали бы ответов от наших клиентов. Одним из потенциальных преимуществ могло бы стать уменьшение конкуренции за ресурсы, но это просто привело бы к увеличению числа ожидающих запросов, хранящихся в брокере. Хуже того, многие из этих запросов с требованием последних цен на репу могут быть уже неактуальны.

Неплохая альтернатива — инвертирование взаимодействия и использование промежуточного ПО, чтобы система репы транслировала последние рекламные объявления репы с целью их последующего использования. Но если бы у нижестоящей системы репы возникла проблема, мы все равно не смогли бы помочь клиенту в поиске лучших цен на продукт.

Таким образом, использование промежуточного ПО, такого как брокеры сообщений, для устранения некоторых проблем с надежностью может быть полезным, но не в каждой ситуации.

Идемпотентность

В *идемпотентных* операциях результат не меняется после первого применения, даже если операция впоследствии применяется несколько раз. Если операции являются идемпотентными, можно повторить вызов несколько раз без неблагоприятных последствий. Это очень полезно, когда мы хотим воспроизвести сообщения, в обработке которых не уверены, — типичный способ восстановления после ошибки.

Рассмотрим простой вызов для добавления нескольких бонусных баллов в результате размещения заказа одним из наших клиентов. Можно выполнить вызов с полезной нагрузкой, показанной в примере 12.1.

Пример 12.1. Зачисление баллов на учетную запись

```
<credit>
  <amount>100</amount>
  <forAccount>1234</account>
</credit>
```

Если вызов будет получен несколько раз, каждый добавит 100 баллов. Таким образом, в нынешнем виде данный вызов не является идемпотентным. Однако, получив немного больше информации, мы позволяем банку баллов сделать этот вызов идемпотентным, как показано в примере 12.2.

Пример 12.2. Добавление дополнительной информации к зачислению баллов, чтобы сделать его идемпотентным

```
<credit>
  <amount>100</amount>
  <forAccount>1234</account>
  <reason>
    <forPurchase>4567</forPurchase>
  </reason>
</credit>
```

Мы знаем, что это начисление относится к конкретному заказу — `4567`. Предполагая, что можно получить только одно зачисление за конкретный заказ, мы могли бы применить это начисление снова, не увеличивая общее количество баллов.

Такой механизм работает так же хорошо при совместной работе на основе событий и может быть особенно полезен, если у вас есть несколько экземпляров одного и того же типа сервиса, подписывающегося на события. Даже если хранить перечень обработанных событий, при некоторых формах асинхронной доставки сообщений могут быть небольшие окна, в которых два исполнителя могут видеть одно и то же сообщение. Обрабатывая события идемпотентным образом, мы гарантируем, что это не вызовет никаких проблем.

Некоторые люди, увлекающиеся этой концепцией, предполагают, что последующие вызовы с теми же параметрами не могут оказать *никакого* влияния. Таким образом мы попадаем в интересное положение. Мы хотели бы действительно зафиксировать факт получения вызова, например, в логах. Необходимо записать время отклика на вызов и собрать эти данные для мониторинга. Ключевым моментом здесь станет то, что мы считаем идемпотентной именно базовую бизнес-операцию, а не все состояние системы.

Некоторые из HTTP-методов, такие как GET и PUT, определены в спецификации HTTP как идемпотентные, но для того, чтобы это было так, необходимо, чтобы ваш сервис обрабатывал эти вызовы идемпотентным образом. Если сделать эти методы неидемпотентными, а вызывающие абоненты будут думать, что они могут безопасно выполнять их повторно, вы можете попасть в затруднительное положение. Помните: одно лишь использование HTTP в качестве базового протокола не означает, что вы получаете идемпотентность автоматически!

Распределение рисков

Один из способов повысить отказоустойчивость — не класть все яйца в одну корзину. То есть убедиться, что у вас нет нескольких сервисов на одном хосте, где перебои в работе повлияют на все из них. Но давайте рассмотрим, что означает *хост*. В большинстве современных ситуаций хост — это виртуальная концепция. Что же делать, если все сервисы хранятся на разных хостах, но все эти хосты на самом деле виртуальные и работают на одном физическом сервере? Если он упадет, можно потерять несколько сервисов. Некоторые платформы виртуализации позволяют вам распределить хосты по нескольким различным физическим блокам, чтобы уменьшить вероятность возникновения такой ситуации.

Для внутренних платформ виртуализации обычной практикой стало сопоставление корневого раздела виртуальной машины с единой сетью хранения данных (storage area network, SAN). Если такая сеть SAN выйдет из строя, она может отключить все подключенные виртуальные машины. Сети SAN большие, дорогие и рассчитаны на бесперебойную работу. Тем не менее за последние 10 лет у меня по крайней мере дважды случались крупные дорогостоящие сбои сетей SAN, и каждый раз последствия были довольно серьезными.

Другой распространенной формой разделения для уменьшения количества сбоев может служить обеспечение работы всех сервисов не в одной стойке в центре обработки данных или распределение сервисов по нескольким дата-центрам. При использовании базового поставщика услуг важно знать, предлагается ли им SLA, и планировать деятельность соответствующим образом. Если необходимо, чтобы ваши сервисы простаивали не более 4 часов в квартал, но ваш хостинг-провайдер может гарантировать только максимальное время простоя 8 часов в квартал, необходимо изменить SLA или придумать альтернативное решение.

Например, система AWS разделена на регионы, которые можно рассматривать как отдельные облачные хосты. Каждый регион, в свою очередь, разделен на две или более зоны доступности (мы обсуждали это ранее). Они представляют собой эквивалент дата-центра AWS. Важно, чтобы сервисы были распределены по нескольким зонам доступности, поскольку AWS не дает никаких гарантий доступности отдельного узла или даже всей зоны. Что касается вычислительных услуг, система обеспечивает только 99,95 % безотказной работы в течение определенного месячного периода по региону в целом, поэтому вам потребуется распределить рабочие нагрузки по нескольким зонам доступности внутри одного региона. Для некоторых людей этого недостаточно, и они запускают свои сервисы в нескольких регионах.

Конечно, следует отметить, что, поскольку поставщики предоставляют вам «гарантию» в виде SLA, они будут стремиться ограничить свою ответственность! Если они не достигают своих целей, что приводит к потере вами клиентов

и большой суммы денег, вы, скорее всего, будете перечитывать договоры в попытках узнать, можно ли получить хоть какую-то компенсацию. Поэтому я бы настоятельно посоветовал вам понять последствия невыполнения поставщиком своих обязательств перед вами и решить, нужно ли вам держать в кармане план Б (или В). Например, у нескольких клиентов, с которыми я работал, была платформа хостинга для аварийного восстановления от другого поставщика, чтобы снизить свою уязвимость от ошибок одной компании.

Теорема CAP

Хотелось бы получить все положительные свойства системы сразу, но, к сожалению, мы знаем, что такой возможности нет. И когда дело доходит до распределенных систем, которые мы строим с использованием микросервисных архитектур, у нас даже есть математическое доказательство, извещающее, что нельзя получить все одновременно. Возможно, вы уже слышали о CAP-теореме, особенно в дискуссиях о достоинствах различных типов хранилищ данных. По сути, она говорит нам о том, что в распределенной системе есть три понятия, которые можно противопоставить друг другу: *согласованность*, *доступность* и *устойчивость к разделению* (CAP — consistency, availability and partition tolerance). В частности, теорема говорит, что мы можем сохранить только два из трех свойств.

Согласованность — это системная характеристика, с помощью которой мы получим один и тот же ответ, если перейдем к нескольким узлам. Доступность означает, что каждый запрос получает ответ, а устойчивость к разделению — это способность системы справляться с тем, что связь между ее частями иногда невозможна.

С тех пор как Эрик Брюер опубликовал свою первоначальную гипотезу, эта идея получила математическое доказательство. Я не собираюсь углубляться в математику самого доказательства, поскольку, во-первых, книга не об этом, а во-вторых, я обязательно где-нибудь ошибусь. Давайте лучше воспользуемся некоторыми отработанными примерами, которые помогут нам понять, что теорема CAP представляет собой квинтэссенцию очень логичного набора рассуждений.

Предположим, что микросервис `Запасы` развернут в двух отдельных центрах обработки данных, как показано на рис. 12.7. Резервная копия экземпляра сервиса в каждом центре представляет собой базу данных, и эти две БД взаимодействуют друг с другом, пытаясь синхронизироваться. Чтение и запись выполняются через локальный узел БД, а репликация используется для синхронизации данных между узлами.

Теперь подумаем о том, что происходит, когда что-то в системе ломается. Представьте, что такая простая вещь, как сетевое соединение между двумя дата-центрами, перестает работать. Синхронизация на этом этапе завершается

сбоем. Записи, сделанные в основную БД в дата-центре 1 (ДЦ1), не будут распространяться на дата-центр 2 (ДЦ2), и наоборот. Большинство баз данных, поддерживающих эти настройки, также поддерживают какую-то технику очередей, чтобы гарантировать возможность восстановиться. Но что происходит во время сбоя?

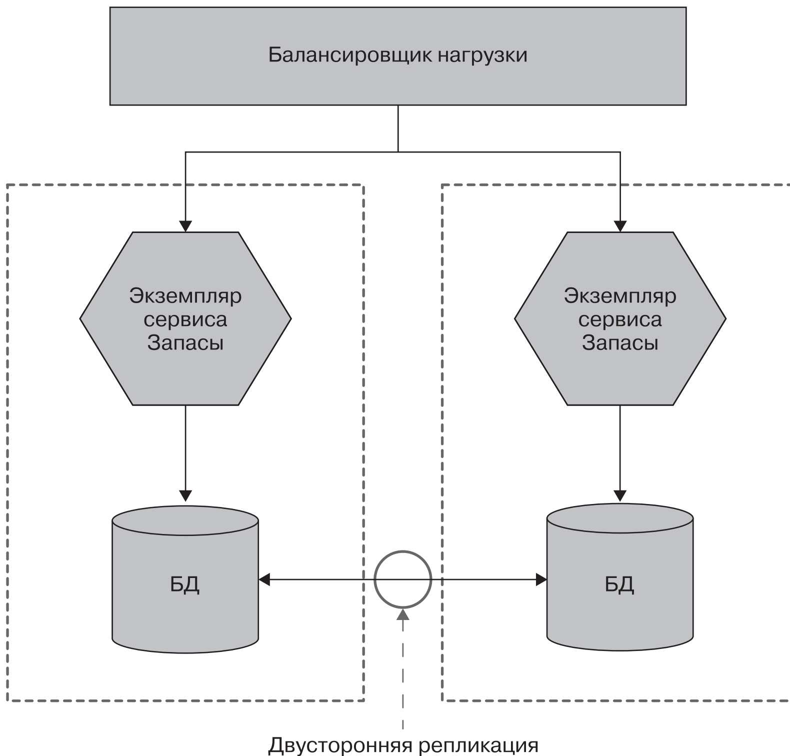


Рис. 12.7. Использование многократной репликации для обмена данными между двумя узлами базы данных

Жертвуя согласованностью

Предположим, что микросервис `Запасы` не закрывается полностью. Если сейчас внести изменения в данные в ДЦ1, база данных в ДЦ2 их не увидит. Это означает, что любые запросы, сделанные к нашему узлу инвентаризации в ДЦ2, видят потенциально устаревшие данные. Другими словами, система по-прежнему *доступна* в том смысле, что оба узла могут обслуживать запросы, и мы сохранили работоспособность системы, несмотря на *разделение*, но потеряли *согласованность*. Нельзя сохранить все три характеристики. Это часто называют AP-системой из-за ее доступности и устойчивости к разделению.

Во время этого разделения если прием записей продолжается, то мы принимаем тот факт, что в какой-то момент в будущем базы должны быть повторно синхронизированы. Чем дольше длится разделение, тем сложнее может стать повторная синхронизация.

Реальность такова, что даже если в сети между узлами БД не было сбоя, репликация данных не происходит мгновенно. Как упоминалось ранее, системы, которые легко уступают в согласованности ради сохранения устойчивости к разделению и доступности, считаются *в конечном счете согласованными*. То есть ожидается, что в какой-то момент в будущем все узлы увидят обновленные данные, но это произойдет не сразу, поэтому придется смириться, что пользователи видят старые данные.

Жертвуя доступностью

Чтобы сохранить согласованность, придется отказаться от чего-то другого. Что ж, для обеспечения согласованности каждый узел БД должен знать, что копия имеющихся у него данных совпадает с копией другого узла БД. Теперь при разделении, если узлы БД не могут взаимодействовать друг с другом, они не координируют свои действия для обеспечения согласованности. Нельзя гарантировать согласованность, поэтому единственный вариант — отказаться отвечать на запрос. Другими словами, мы пожертвовали доступностью. Наша система является согласованной и устойчивой к разделению, то есть CP-системой. В этом режиме сервис должен был бы решить, как ухудшить функциональность, пока разделение не будет восстановлено и узлы базы данных не будут повторно синхронизированы.

Согласованность между несколькими узлами действительно сложна. В распределенных системах мало что (возможно, даже нет ничего) сложнее. Задумайтесь об этом на мгновение. Представьте, что я хочу прочитать запись с локального узла БД. Как мне узнать, что он обновлен? Я должен спросить у другого узла. Но я также должен попросить первый узел БД не допускать обновления до завершения считывания второго. Другими словами, мне нужно инициировать транзакционное чтение между несколькими узлами БД, чтобы обеспечить согласованность. Но обычно люди не выполняют транзакционное чтение, правда? Потому что оно происходит медленно. И требует блокировок. Считывание может заблокировать всю систему.

Как обсуждалось ранее, распределенные системы должны ожидать сбоя. Рассмотрим транзакционное чтение по набору согласованных узлов. Я прошу удаленный узел заблокировать указанную запись во время инициирования чтения. Завершаю чтение и прошу его снять блокировку, но теперь я не могу с ним взаимодействовать. Что будет дальше? Блокировки действительно трудно применять правильно даже в системе с одним процессом, и их значительно сложнее грамотно реализовать в распределенной системе.

Помните, мы говорили о распределенных транзакциях в главе 6? Основная причина их сложности заключается в обеспечении согласованности между несколькими узлами.

Добиться правильной согласованности с несколькими узлами настолько нелегко, что я настоятельно, *категорически* советую при возникновении потребности в этом не пытаться изобрести решение самостоятельно. Лучше выберите хранилище данных или службу блокировки, обладающую требуемыми характеристиками. Например, Consul, который мы обсуждали в подразделе «Динамические реестры сервисов» главы 5, реализует строго согласованное хранилище значений ключей, предназначенное для совместного использования конфигурации между несколькими узлами. Наряду с выражением «Друзья не позволяют друзьям писать свою собственную криптографию» следует добавить «Друзья не позволяют друзьям писать свое собственное распределенное согласованное хранилище данных». Если вы считаете, что вам нужно написать собственное хранилище данных CP, сначала прочитайте все статьи по этому вопросу, защитите докторскую диссертацию, а затем, потратив несколько лет на разработку, с удивлением обнаружите, что это неправильно работает. А я тем временем буду использовать что-то готовое или, что более вероятно, буду *очень стараться* построить согласованные AP-системы.

Жертвуя устойчивостью к разделению

Мы должны выбрать два из трех, верно? Итак, у нас были AP-система и согласованная, но сложная в построении и масштабировании CP-система. Почему бы не быть и CA-системе? Итак, как можно пожертвовать устойчивостью к разделению? Если у нашей системы нет устойчивости к разделению, она не может работать через сеть. Другими словами, это должен быть единый процесс, работающий локально. CA-системы не существуют в мире распределенных систем.

AP или CP?

Что правильно, AP или CP? Что ж, все *зависит от обстоятельств*. Как люди, создающие систему, мы знаем, что существует компромисс. Мы знаем, что AP-системы легче масштабируются и их проще создавать, в то время как CP-система потребует больше работы из-за проблем с поддержкой распределенной согласованности. Но мы не всегда понимаем, какое влияние этот компромисс окажет на бизнес. Устаревшие на 5 минут записи для нашей системы инвентаризации — нормально? Если да, то AP-система может стать подходящим вариантом. Но как насчет баланса, хранящегося для клиента в банке? Могут ли такие данные быть устаревшими? Не зная контекста, в котором используется операция, невозможно определить, какая система будет правильной. Знание CAP-теоремы поможет понять, что компромисс существует.

Это не «все или ничего»

Наша система в целом не обязательно должна быть исключительно AP или CP. Каталог для MusicCorp мог бы быть AP-типа, так как мы не слишком беспокоимся об устаревании записей. Но для сервиса `Запасы` оптимальный вариант — CP, поскольку мы не хотим продавать клиенту то, чего у нас нет, а потом извиняться.

Но отдельные сервисы даже не обязательно должны относиться к типу CP или AP.

Давайте подумаем о микросервисе `Баланс баллов`, где хранятся записи о том, сколько баллов лояльности накопили клиенты. Допустим, нам все равно, устарел ли отображаемый покупателю баланс, однако, когда дело дойдет до обновления баланса, необходимо, чтобы он был актуальным. Это гарантирует, что покупатель не использует больше баллов, чем есть у него на счету. Будет ли это микросервис CP, или AP, или и тот и другой? На самом деле то, что мы сделали, — это свели компромиссы CAP-теоремы к индивидуальным возможностям микросервисов.

Другая сложность заключается в том, что ни согласованность, ни доступность не представляют собой «все или ничего». Многие системы позволяют прийти к более тонким компромиссам. Например, с помощью Cassandra можно найти различные компромиссы для отдельных вызовов. Поэтому, если требуется строгая согласованность, можно выполнить чтение, блокируемое до тех пор, пока все реплики не ответят, подтверждая согласованность значения, или пока не ответит определенный кворум реплик или даже просто отдельный узел. Очевидно, что, если ввести блокировку в ожидании ответа всех реплик, а одна из них будет недоступна, система останется заблокированной в течение длительного времени. С другой стороны, если бы требовался самый быстрый из возможных ответ на считывание, можно было бы подождать ответа только от одного узла. Но в этом случае существует вероятность непоследовательного представления данных.

Вы часто будете видеть сообщения о том, как люди «побеждают» CAP-теорему. Они этого не сделали. Что они сделали, так это создали систему, в которой некоторые возможности относятся к CP, а некоторые к AP. Математическое доказательство, лежащее в основе CAP-теоремы, остается в силе.

А теперь реальный мир

Многое из того, что мы обсудили, относится к электронному миру — битам и байтам, хранящимся в памяти. Мы говорим о согласованности почти по-детски: воображаем, что в рамках созданной нами системы можно остановить мир и все это обретет смысл. И все же многое из того, что мы строим, представляет собой всего лишь отражение реального мира, и мы не можем это контролировать, не так ли?

Вернемся к системе инвентаризации MusicCorp. Она сопоставляется с реальными физическими объектами. В системе ведется подсчет количества альбомов, имеющихся на складах компании. В начале дня у нас было 100 копий *Give Blood* от Brakes. Один мы продали. Теперь осталось 99 копий. Легко, правда? Но что

будет, если во время отправки заказа кто-то уронит копию альбома на пол, наступит на нее и сломает? Наши системы говорят, что на полках осталось 99 экземпляров, но на самом деле их всего 98.

Что, если бы мы сделали AP-систему инвентаризации и нам иногда приходилось бы связываться с пользователем и сообщать ему, что одного из его товаров на самом деле нет на складе? Разве это была бы худшая идея на свете? Конечно, гораздо проще создать и масштабировать систему и убедиться, что она корректна.

Стоит признать: какими бы последовательными ни были наши системы, они не могут знать всего, что происходит, особенно когда мы ведем учет предметов реального мира. Это одна из главных причин того, почему AP-системы в конечном счете оказываются правильным решением во многих ситуациях. Помимо сложности построения, CP-системы все равно не могут решить все наши проблемы.

АНТИХРУПКОСТЬ

В первом издании я рассказывал о концепции антихрупкости, популяризированной Нассимом Талебом. Эта концепция описывает, как системы на самом деле выигрывают от сбоев и беспорядка, и была подчеркнута как источник вдохновения для работы некоторых частей Netflix, особенно в отношении хаос-инжиниринга. Однако, рассматривая концепцию отказоустойчивости в более широком смысле, мы понимаем, что антихрупкость — это всего лишь подмножество концепции отказоустойчивости. Когда мы рассматриваем введенные ранее концепции стабильной расширяемости и непрерывной адаптивности, это становится ясным.

Я думаю, что антихрупкость стала такой раздутой концепцией в ИТ на фоне того, что мы узко мыслили в отношении отказоустойчивости — мы думали только о надежности и, возможно, о восстановлении, но игнорировали остальное. Поскольку область проектирования отказоустойчивости на сегодняшний день получает все большее признание и распространение, представляется целесообразным выйти за рамки термина «антихрупкость». В то же время стоит обеспечить рассмотрение некоторых идей, лежащих в его основе, — они стали неотъемлемой частью устойчивости в целом.

Хаос-инжиниринг

С момента выхода первого издания все большее внимание к себе привлекает техника *хаос-инжиниринга*. Названная в честь используемых в Netflix методов, она может стать полезным подходом для повышения отказоустойчивости либо с точки зрения обеспечения *надежности* ваших систем, какой вы себе ее представляете, либо как часть подхода к *непрерывной адаптивности* системы.

Термин *хаос-инжиниринг* вызывает некоторые трудности из-за путаницы в объяснении его значения. Для многих это означает «запустить инструмент в своем ПО и посмотреть, что будет», чему, возможно, не способствует тот факт, что многие из самых ярых сторонников хаос-инжиниринга часто сами продают инструменты для запуска в вашем ПО, чтобы реализовать этот самый хаос-инжиниринг.

Получить четкое определение того, что означает хаос-инжиниринг для его практиков, довольно сложно. Лучшее определение (по крайней мере, на мой взгляд), с которым я сталкивался, таково.

> Хаос-инжиниринг — это дисциплина, заключающаяся в экспериментировании с системой с целью укрепления уверенности в способности системы выдерживать турбулентные условия в эксплуатации.
>
> *Принципы хаос-инжиниринга (https://principlesofchaos.org)*

Слово «система» здесь берет на себя огромную роль. Некоторые рассматривают его в узком смысле как программные и аппаратные компоненты. Но в контексте разработки отказоустойчивости важно, чтобы мы рассматривали *систему* как совокупность людей, процессов, культуры и, да, ПО и инфраструктуры, участвующих в создании нашего продукта. Это означает, необходимо рассматривать хаос-инжиниринг более широко, чем просто «давайте выключим несколько машин и посмотрим, что произойдет».

Игровые дни

Задолго до того, как хаос-инжиниринг получил свое название, люди проводили упражнения в так называемый игровой день, чтобы проверить готовность людей к определенным событиям. Спланированный заранее, но в идеале запущенный неожиданно (для участников), он дает вам возможность проверить своих сотрудников и процессы в реалистичной, но вымышленной ситуации. Во время моей работы в Google это было довольно распространенным явлением для различных систем, и я, конечно, думаю, что многие организации могли бы извлечь выгоду из регулярного проведения подобных упражнений. Google выходит за рамки простых тестов, имитируя отказ сервера, и в рамках своих упражнений по восстановлению после катастроф — DiRT (disaster recovery test) — имитирует крупномасштабные катастрофы, такие как землетрясения[1].

Игровые дни можно использовать для поиска предполагаемых слабых мест в системе. В своей книге «Изучение хаос-инжиниринга»[2] Расс Майлз делится примером упражнения игрового дня, которое он организовал, чтобы частично изучить чрезмерную зависимость от одного сотрудника по имени Боб. В течение игрового дня Боб был изолирован в комнате и не мог помочь команде во время имитируемого отключения. Боб наблюдал, однако в итоге ему пришлось вмешаться, когда команда, пытаясь устранить проблемы с «поддельной» системой, по ошибке вошла в эксплуатируемую систему и находилась в процессе уничтожения данных. Можно только предположить, сколько из этого упражнения было извлечено уроков.

---

[1] *Krishnan K.* Weathering the Unexpected // Acmqueue 10, № 9 (2012), https://oreil.ly/BCSQ7.

[2] *Miles R.* Learning Chaos Engineering. — O’Reilly, 2019.

Эксперименты в эксплуатационной среде

Масштабы, в которых работает Netflix, хорошо известны, как и то, что Netflix полностью базируется на инфраструктуре AWS. Эти два фактора означают, что компания должна хорошо переносить сбои. В Netflix поняли, что готовиться к отказам и фактически *знать*, что ваше ПО справится с ними, когда они произойдут, — это две разные вещи. С этой целью Netflix *провоцирует* сбои, чтобы гарантировать, что системы устойчивы при запуске в них различных инструментов.

Самый известный из этих инструментов — Chaos Monkey, который в определенные часы дня отключает случайные машины в эксплуатационной среде. Знание того, что это может произойти и произойдет, означает, что разработчики, создающие системы, действительно должны быть готовы к такому. Chaos Monkey — лишь одна из «обезьяньей армии» ботов Netflix. Chaos Gorilla используется для отключения всей зоны доступности (эквивалент дата-центра AWS), в то время как Latency Monkey имитирует медленное сетевое подключение между машинами. Для многих окончательным испытанием надежности вашей системы может стать натиск собственной «армии обезьян» на эксплуатационную инфраструктуру.

От надежности к запредельному

Используемый в самой узкой форме, хаос-инжиниринг может стать полезным занятием с точки зрения повышения надежности приложения. Помните, что надежность в контексте разработки отказоустойчивости означает степень, в которой система может справиться с ожидаемыми проблемами. В Netflix знали, что не могут полагаться на доступность какой-либо конкретной виртуальной машины в их эксплуатационной среде, поэтому они создали Chaos Monkey, чтобы гарантировать, что система сможет выдержать эту *ожидаемую* проблему.

Однако, если вы используете инструментарий хаос-инжиниринга как часть подхода, позволяющего постоянно подвергать сомнению отказоустойчивость своей системы, у него может быть гораздо больше вариантов применения. Использование инструментов в данной области, помогающих ответить на возникающие вопросы «что, если», постоянно подвергая сомнению ваше понимание, может оказать гораздо большее влияние. Chaos Toolkit (https://chaostoolkit.org) — это очень популярный проект с открытым исходным кодом, помогающий вам ставить эксперименты в своей системе. Reliably (https://reliably.com) — компания, основанная создателями Chaos Toolkit, предлагает более широкий спектр инструментов, помогающих в хаос-инжиниринге в целом, хотя, пожалуй, самым известным поставщиком в этой сфере стала компания Gremlin (https://www.gremlin.com).

Просто помните, что использование инструмента хаос-инжиниринга не делает вашу систему отказоустойчивой.

Поиск виновных

Когда что-то идет не так, есть много способов справиться с ситуацией. Очевидно, что сразу после случившегося мы сосредоточимся на восстановлении, что вполне разумно. А после этого, слишком часто, следуют взаимные обвинения. По умолчанию принято искать виноватых. Концепция «анализа первопричин» подразумевает *наличие* первопричины. Удивительно, как часто всем хочется, чтобы этой первопричиной был человек.

Несколько лет назад, когда я работал в Австралии, у Telstra, основной телекоммуникационной компании (а ранее монополии), произошел серьезный сбой, повлиявший на услуги голосовой связи и телефонии. Инцидент был особенно проблематичным из-за масштабов и продолжительности отключения. В Австралии много очень изолированных сельских общин, и подобные перебои, как правило, особенно серьезны. Почти сразу после отключения главный операционный директор Telstra выступил с заявлением, в котором дал понять, что они точно знают, что вызвало проблему[1].

> «Мы отключили данный узел, так как, к сожалению, человек, который занимался этим вопросом, не следовал инструкциям и повторно подключил клиентов к неправильно функционирующему узлу, вместо того чтобы перевести их на девять других резервных узлов, — сказала г-жа Маккензи журналистам во вторник днем. — Мы приносим извинения всем нашим клиентам. Это досадная человеческая ошибка».

Во-первых, обратите внимание, что заявление было сделано через несколько часов после отключения. За это время в Telstra успели вскрыть, должно быть, чрезвычайно сложную систему, чтобы точно знать, кто виноват — один человек. Если это правда, что один человек, совершивший ошибку, действительно может поставить на колени всю телекоммуникационную компанию, то можно подумать, что это больше говорит о телекоммуникационной компании, чем о человеке. Кроме того, в то время Telstra ясно дала понять своим сотрудникам, что она с удовольствием укажет на них пальцем и переложит вину[2].

Проблема обвинения людей после подобных инцидентов в том, что перекладывание ответственности в конечном счете создает культуру страха, когда люди не хотят говорить вам, когда что-то идет не так. В результате вы теряете

---

[1] *Aubusson K., Biggs T.* Major Telstra Mobile Outage Hits Nationwide, with Calls and Data Affected // Sydney Morning Herald, 9 февраля 2016 года, https://oreil.ly/4cBcy.

[2] Более подробно об инциденте с Telstra — «Telstra, человеческая ошибка и культура обвинений», https://oreil.ly/OXgUQ. Когда я опубликовал эту статью, я упустил из виду, что Telstra были клиентами компании, в которой я работал. Мой тогдашний работодатель действительно очень хорошо справился с ситуацией, хотя это и доставило хлопот, когда некоторые из моих комментариев были подхвачены национальной прессой.

возможность учиться на неудачах, культивируя условия для повторения тех же ошибок. Создание организации, в которой люди могут спокойно признавать допущенные ошибки, представляет важное значение для формирования культуры обучения и, в свою очередь, может иметь большое значение для построения компании, способной разрабатывать более надежное ПО, помимо очевидных преимуществ в виде приятного места для работы.

Возвращаясь к Telstra, учитывая, что причина вины была четко установлена в ходе углубленного расследования, проведенного всего через несколько часов после общенационального отключения, мы явно не ожидаем никаких последующих отключений, правда? К сожалению, Telstra столкнулась с целым рядом последующих сбоев. Еще одна ошибка человека? Возможно, в Telstra так и думали — после серии инцидентов исполнительный директор подал в отставку.

Если вы хотите узнать больше о том, как создать фирму, позволяющую извлечь максимум пользы из ошибок и организовать более благоприятную среду для своих сотрудников, книга Джона Оллспоу «Постмортемы без упреков и культура справедливости» станет отличной отправной точкой[1].

В конечном счете, как я уже неоднократно подчеркивал в данной главе, *отказоустойчивости* необходим вопрошающий ум — стремление постоянно исследовать слабые места в нашей системе. Это требует культуры обучения, и зачастую лучше обучаться на практике. Поэтому жизненно важно предоставить гарантию, что, когда произойдет худшее, вы сделаете все возможное для создания среды, в которой можно будет максимально использовать собранную после катастрофы информацию, чтобы уменьшить вероятность повторения такой ситуации.

Резюме

По мере того как наше ПО становится все более важным для жизни наших пользователей, возрастает и стремление к повышению его отказоустойчивости. Однако, как мы отметили в этой главе, нельзя добиться отказоустойчивости, *просто* проявляя внимание к своему ПО и инфраструктуре, — необходимо также заботиться о своих сотрудниках, процессах и организациях. Мы рассмотрели четыре основные концепции отказоустойчивости, описанные Дэвидом Вудсом.

Надежность

Способность поглощать ожидаемые возмущения.

Восстановление

Способность восстанавливаться после травмирующего события.

---

[1] *Allspaw J.* Blameless Post-Mortems and a Just Culture // Code as Craft (блог), Etsy, 22 мая 2012 года, https://oreil.ly/7LzmL.

Стабильная расширяемость

Насколько хорошо мы справляемся с неожиданной ситуацией.

Непрерывная адаптивность

Способность постоянно адаптироваться к меняющимся условиям, заинтересованным сторонам и требованиям.

Рассматривая микросервисы более узко, можно заметить, что они дают множество способов повышения *надежности* наших систем. Но эта улучшенная надежность не бесплатна — вам все равно придется решать, какие варианты использовать. Ключевые шаблоны стабильности, такие как автоматические выключатели, тайм-ауты, резервирование, изоляция, идемпотентность и т. п., — все это инструменты в вашем распоряжении, но необходимо определиться, когда и где их применять. Однако, помимо этих узких концепций, нам также нужно постоянно быть начеку в отношении того, чего мы не знаем.

Вы также должны определить желаемый уровень отказоустойчивости, а это почти всегда определяется пользователями и владельцами бизнеса вашей системы. Как технолог, вы можете отвечать за то, как все делается, но знание того, какая отказоустойчивость необходима, потребует хорошего и частого тесного общения с пользователями и владельцами продуктов.

Вернемся к цитате Дэвида Вудса, которую мы использовали при обсуждении непрерывной адаптивности.

> Независимо от того, насколько хорошо мы справлялись раньше и насколько успешными мы были, будущее может быть другим, а мы — неподготовленными. Наши системы могут оказаться ненадежными и хрупкими перед лицом нового будущего.

Задавая одни и те же вопросы снова и снова, вы не сможете понять, готовы ли вы к неопределенному будущему. Вы не знаете того, чего не знаете, — принятие подхода, при котором вы постоянно учитесь и ставите все под сомнение, будет ключом к повышению отказоустойчивости.

Один из рассмотренных типов модели стабильности — резервирование — может оказаться очень эффективным. Эта идея прекрасно переходит в следующую главу, где мы рассмотрим различные способы масштабирования микросервисов. Масштабирование не только помогает справляться с большей нагрузкой, но и может стать эффективным способом реализации резервирования в системах и, следовательно, повышения надежности наших систем.

ГЛАВА 13

Масштабирование

Тебе понадобится лодка побольше.

Шеф Броуди, из х/ф «Челюсти»

Мы масштабируем наши системы по одной из двух причин: повысить производительность системы и повысить ее надежность. В данной главе мы поговорим о модели для описания различных типов масштабирования, а затем подробно рассмотрим, как каждый тип масштабирования может быть реализован с применением микросервисной архитектуры. В конце главы вы должны получить набор методов для решения возникающих проблем с масштабированием.

Однако для начала я расскажу про различные типы масштабирования, которые вы, возможно, захотите применить.

Четыре оси масштабирования

Не существует единственно верного способа масштабирования системы, поскольку используемый метод зависит от типа ваших ограничений. Есть несколько различных типов масштабирования, которые можно применить для повышения производительности, надежности или, возможно, и того и другого. Модель, которую я часто использовал для описания различных типов масштабирования, — это куб масштабирования из книги «Искусство масштабирования»[1]. Эта модель разбивает масштабирование на три категории, охватывающие в контексте компьютерных систем функциональную декомпозицию, горизонтальное дублирование и разделение данных. Ценность этой модели в том, что она помогает понять, что вы можете масштабировать систему по одной, двум или всем трем из этих осей в зависимости от потребностей.

---

[1] *Abbott M. L., Fisher M. T.* The Art of Scalability: Scalable Web Architecture, Processes and Organizations for the Modern Enterprise. 2 ed. — Addison-Wesley, 2015.

Однако, особенно в мире виртуализированной инфраструктуры, мне всегда казалось, что данной модели не хватает четвертой оси вертикального масштабирования. Хотя при таком подходе модель перестанет быть кубом. Тем не менее я думаю, что это полезный набор механизмов для определения того, как лучше масштабировать архитектуры микросервисов. Прежде чем мы подробно рассмотрим эти типы масштабирования, а также их относительные плюсы и минусы, необходимо сделать краткий обзор.

Вертикальное масштабирование

В двух словах, это означает приобретение более мощной машины.

Горизонтальное дублирование

Наличие нескольких устройств, способных выполнять одну и ту же работу.

Разделение данных

Разделение работы на основе какого-либо атрибута данных, например группы клиентов.

Функциональная декомпозиция

Разделение работы в зависимости от типа, например декомпозиция микросервиса.

Понимание того, какая комбинация этих методов масштабирования станет наиболее подходящей, в корне зависит от характера проблемы масштабирования, с которой вы сталкиваетесь. Чтобы изучить этот вопрос более подробно, а также рассмотреть примеры, как эти концепции могут быть реализованы для MusicCorp, мы рассмотрим их пригодность для реальной компании FoodCo[1]. FoodCo осуществляет прямую доставку продуктов питания клиентам в ряде стран по всему миру.

Вертикальное масштабирование

Некоторые операции могут просто выиграть от большей нагрузки. Приобретение большего сервера с более быстрым процессором и улучшенным I/O часто может снизить задержку и повысить пропускную способность, позволяя выполнять больше работы за меньшее время. Итак, если ваше приложение работает недостаточно быстро или не может обрабатывать достаточное количество запросов, почему бы не приобрести машину посерьезнее?

В случае с компанией FoodCo одной из проблем, с которой она столкнулась, стала растущая конкуренция при записи в ее основную базу данных. Обычно вертикальное масштабирование представляется оптимальным вариантом для быстрого масштабирования операций записи в реляционной БД. И действительно, компания FoodCo уже несколько раз модернизировала инфраструктуру БД. Проблема в том, что FoodCo уже продвинулась в этом вопросе на максимально допустимый комфортный для себя уровень. Вертикальное масштабирование

[1] Как и прежде, я анонимизировал компанию, FoodCo — это не настоящее название!

работало в течение многих лет, но, учитывая прогнозы роста компании, даже если FoodCo сможет приобрести более мощную машину, это вряд ли решит проблему в долгосрочной перспективе.

Исторически сложилось, что, когда вертикальное масштабирование требовало покупки оборудования, этот метод становился более проблематичным. Время, необходимое для покупки оборудования, означало, что к этому нельзя было относиться легкомысленно, и если окажется, что наличие более мощной машины не решит ваших проблем, то вы, скорее всего, потратили кучу денег зря. Кроме того, обычно при получении новых более мощных машин возникает множество хлопот с получением бюджетных разрешений, ожиданием доставки машины и т. д., что, в свою очередь, приводит к появлению значительных неиспользуемых мощностей в дата-центрах.

Однако переход к виртуализации и появление общедоступного облака в значительной степени помогли справиться с этой формой масштабирования.

Реализация

Реализация будет варьироваться в зависимости от того, на чьей инфраструктуре вы работаете. Если вы работаете в своей собственной виртуализированной инфраструктуре, можно просто изменить размер виртуальной машины, чтобы использовать больше базового оборудования, — это то, что можно реализовать быстро и почти без риска. Если виртуальная машина настолько велика, что занимает все базовое оборудование, данный вариант, конечно, не подходит — возможно, придется покупать больше аппаратного обеспечения. Аналогично если вы работаете на своих серверах «на голом железе» и у вас нет запасного более мощного оборудования, чем уже используемое, то опять же вам придется докупать еще.

В общем, если наступил момент, когда мне приходится покупать новую инфраструктуру, чтобы попробовать вертикальное масштабирование, из-за возросших, оказывающих существенное влияние затрат (и времени), я вполне могу пока пропустить эту форму масштабирования и рассмотреть горизонтальное дублирование, о котором мы поговорим далее.

Но появление облачных технологий также позволило нам легко арендовать на почасовой основе (а в некоторых случаях и на более короткий срок) полностью управляемые машины у облачных провайдеров. Кроме того, основные поставщики облачных услуг предлагают более широкий выбор машин для решения различных типов задач. Будет ли ваша рабочая нагрузка более требовательная к памяти? Порадуйте себя экземпляром AWS `u-24tb1.metal`, предоставляющим 24 Тбайт памяти (да, это не опечатка). Сейчас количество рабочих нагрузок, которым на самом деле может потребоваться такой объем памяти, довольно невелико, но возможность использования таких мощностей уже есть. Существуют также машины, адаптированные к высокой производительности I/O, ЦП или ГП. Если ваша система уже работает на технологии общедоступного облака, масштабирование такого типа не станет сложной задачей.

Ключевые преимущества

В виртуализированной инфраструктуре, особенно в облаке, внедрение такой формы масштабирования будет быстрым. Большая часть работы по масштабированию приложений сводится к экспериментам — наличию идеи о чем-то, что может улучшить вашу систему, внесению изменений и измерению их влияния. Действия, которые можно опробовать быстро и без риска, всегда стоит делать на ранней стадии процесса усовершенствования системы. И вертикальное масштабирование подходит под эти требования как нельзя лучше.

Стоит также отметить, что вертикальное масштабирование упростит выполнение других типов масштабирования. Конкретным примером может послужить перемещение инфраструктуры БД на более мощную машину, что позволит разместить логически изолированные базы данных для вновь созданных микросервисов в рамках функциональной декомпозиции.

Ваш код или БД вряд ли потребуют каких-либо изменений для использования более крупной базовой инфраструктуры при условии, что ОС и чипсеты останутся прежними. Даже если в ваше приложение понадобится внести модификации для обеспечения совместимости с новым аппаратным обеспечением, они могут быть ограничены такими вещами, как увеличение объема памяти, доступной для вашей среды выполнения, с помощью флагов среды выполнения.

Ограничения

По мере увеличения масштаба наших рабочих машин процессоры не становятся быстрее — мы просто получаем больше ядер. Этот сдвиг произошел за последние 5–10 лет. Раньше считалось, что каждое новое поколение аппаратного обеспечения предусматривало значительное повышение тактовой частоты процессора. Это означало, что программы получали большие скачки в производительности. Однако повышение тактовой частоты резко замедлилось, и вместо этого мы получаем все больше и больше ядер процессора, с которыми можно поэкспериментировать. Проблема в том, что раньше ПО писалось без расчета на использование многоядерного оборудования. То есть не факт, что переход приложения с 4- на 8-ядерную систему приведет к значительным улучшениям, даже если ваша существующая система зависима от работы процессора. Изменение кода для использования преимуществ многоядерного оборудования может быть масштабным мероприятием и наверняка потребует полного изменения идиом программирования.

Наличие машины большей мощности также, вероятно, мало что даст для повышения надежности. Более крупный и новый сервер мог бы повысить надежность, но в конечном счете если машина сломалась, то она сломалась. В отличие от других форм масштабирования, которые мы рассмотрим, вертикальное масштабирование вряд ли окажет большое влияние на повышение надежности системы.

Наконец, по мере роста мощности машины растет и цена — но не всегда пропорционально увеличению доступных вам ресурсов. Иногда это означает, что экономически эффективнее содержать много слабых машин, чем мало, но мощных.

Горизонтальное дублирование

При горизонтальном дублировании вы дублируете часть своей системы, чтобы обрабатывать больше рабочих нагрузок. Точные механизмы различаются, но, по сути, горизонтальное дублирование требует, чтобы у вас был способ распределить работу между дубликатами.

Как и в случае с вертикальным масштабированием, данный тип достаточно прост в реализации и часто является одной из идей, которые стоит попробовать реализовать на ранней стадии. Если ваша монолитная система не справляется с нагрузкой, запустите несколько ее копий и посмотрите, поможет ли это!

Реализации

Вероятно, наиболее очевидной формой горизонтального дублирования, приходящей на ум, является использование балансировщика нагрузки для распределения запросов по нескольким копиям функциональности, как показано на рис. 13.1, где мы балансируем нагрузку между несколькими экземплярами микросервиса `Каталог` MusicCorp. Возможности балансировщиков нагрузки различаются, но все они должны иметь механизм распределения нагрузки между узлами, а также определять, когда узел недоступен, и удалять его из пула. С точки зрения потребителя, балансировщик нагрузки полностью прозрачен в реализации — в этом отношении можно считать его частью логической границы микросервиса. Исторически сложилось так, что балансировщики нагрузки рассматривались в первую очередь с точки зрения выделенного оборудования, но это уже давно перестало быть обычным явлением — вместо этого большая часть балансировки нагрузки выполняется программным обеспечением, часто работающим на стороне клиента.

Другим примером горизонтального дублирования может служить шаблон конкурирующих потребителей, подробно описанный в книге «Шаблоны интеграции корпоративных приложений»[1]. На рис. 13.2 показано, как новые песни загружаются в MusicCorp. Эти песни необходимо перекодировать, чтобы использовать их в рамках нового стримингового предложения MusicCorp. У нас есть общая очередь работ, в которую помещаются эти задания, и набор экземпляров сервиса `Транскодер песен`, потребляющих задания из очереди, — разные экземпляры конкурируют за задания. Чтобы повысить пропускную способность системы, стоит увеличить количество экземпляров сервиса `Транскодер песен`.

---

[1] *Хоуп Г., Вулф Б.* Шаблоны интеграции корпоративных приложений.

Рис. 13.1. Микросервис Каталог развернут в виде нескольких экземпляров с балансировщиком нагрузки для распределения запросов

Рис. 13.2. Транскодирование для стриминговой передачи расширяется с использованием шаблона конкурирующих потребителей

В случае с FoodCo была использована форма горизонтального дублирования для снижения нагрузки чтения на основную БД за счет использования реплик для считывания, как показано на рис. 13.3. Это уменьшило нагрузку чтения на основной узел БД, высвободив ресурсы для обработки записей, и сработало очень эффективно, поскольку большая часть нагрузки на основную систему была связана с чтением. Эти операции считывания могут быть легко перенаправлены на реплики, и это обычная практика использования балансировщика нагрузки для нескольких реплик чтения.

Маршрутизация либо к основной БД, либо к реплике чтения обрабатывается внутри микросервиса. Для потребителей этого микросервиса совершенно неважно, попадает ли отправленный ими запрос в основную БД или в БД реплик для считывания.

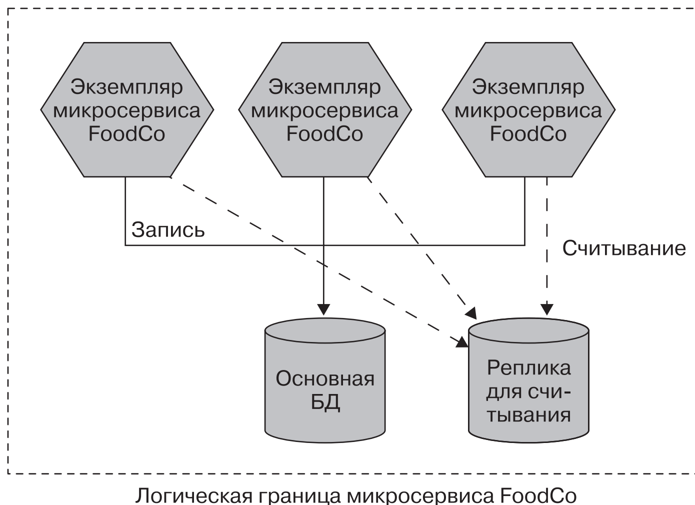


Рис. 13.3. В FoodCo используют реплики для считывания для масштабирования трафика чтения

Ключевые преимущества

Горизонтальное дублирование — это довольно просто. Редко возникает необходимость в обновлении приложения, поскольку работа по распределению нагрузки часто выполняется в другом месте, например через очередь, запущенную в брокере сообщений, или, возможно, в балансировщике нагрузки. Если вертикальное масштабирование недоступно, эта форма масштабирования, как правило, становится следующим подходом, на который стоит обратить внимание.

Если предположить, что работа может быть легко распределена между дубликатами, это элегантный способ распределения нагрузки и уменьшения конкуренции за необработанные вычислительные ресурсы.

Ограничения

Как и практически все варианты масштабирования, которые мы рассмотрим, горизонтальное дублирование требует дополнительной инфраструктуры, что, конечно, может стоить дороже. Оно также может быть немного грубым инструментом. Например, можно запустить несколько полных копий своего монолитного приложения, даже если только часть монолита испытывает проблемы с масштабированием.

Большая часть работы здесь заключается в реализации ваших механизмов распределения нагрузки. Они могут варьироваться от простых, таких как балансировка нагрузки HTTP, до более сложных, таких как использование брокера сообщений или настройка реплик БД для считывания. Вы рассчитываете, что этот механизм распределения нагрузки будет выполнять свою работу, а ключевым моментом станет понимание любых его ограничений и принципов работы.

Некоторые системы могут предъявлять дополнительные требования к механизму распределения нагрузки. Например, они могут потребовать, чтобы каждый запрос, связанный с одним и тем же сеансом пользователя, направлялся на одну и ту же реплику. Это решается с помощью балансировщика нагрузки, требующего постоянной балансировки нагрузки сессии, что, в свою очередь, может ограничить выбор механизмов распределения нагрузки. Стоит отметить, что системы, требующие подобной балансировки, подвержены другим проблемам, и в целом я бы избегал создания систем с такими требованиями.

Разделение данных

Начав с простых форм масштабирования, мы теперь вступаем на более сложную территорию. Разделение данных требует, чтобы мы распределяли нагрузку на основе какого-либо аспекта данных.

Реализация

Принцип работы разделения данных заключается в том, что мы берем ключ, связанный с рабочей нагрузкой, и применяем к нему функцию, в результате чего получается *раздел* (иногда называемый сегментом), на который мы будем распределять работу. На рис. 13.4 показаны два раздела. Наша функция довольно проста: мы отправляем запрос в одну БД, если фамилия начинается с букв от «А» до «М», или в другую, если фамилия начинается с букв от «Н» до «Я». На самом деле это плохой пример алгоритма разделения (вскоре мы разберемся почему), но, надеюсь, он достаточно прост для иллюстрации идеи.

В этом примере мы выполняем разделение на уровне БД. Запрос к микросервису `Покупатель` может попасть в любой экземпляр микросервиса. Но когда

мы выполняем операцию, требующую обращения к БД (чтение или запись), этот запрос направляется на соответствующий узел базы данных на основе имени клиента. В случае реляционной БД схема обоих узлов БД была бы идентичной, но содержимое каждого из них относилось бы только к определенному подмножеству покупателей.

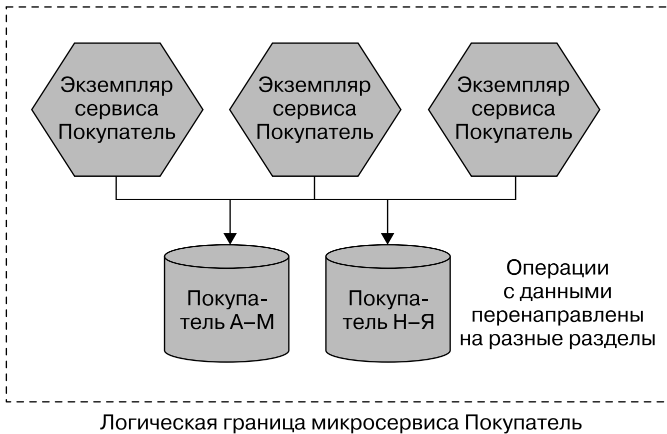


Рис. 13.4. Данные клиентов разделены на две разные базы данных

Разделение на уровне БД часто имеет смысл, если используемая технология базы данных изначально поддерживает такую концепцию, поскольку затем можно перенести эту проблему на существующую реализацию. Однако мы могли бы вместо этого выполнить разделение на уровне экземпляра микросервиса, как показано на рис. 13.5. Здесь нам нужно определить из входящего запроса, к какому разделу должен быть отправлен запрос. В нашем примере это делается через некоторую форму прокси-сервера. В случае нашей модели разделения на основе данных покупателя достаточно, если его имя будет указано в заголовках запроса. Данный подход имеет смысл, если вам нужны выделенные экземпляры микросервиса для разделения, что полезно при кэшировании в памяти. Это также означает, что вы можете масштабировать каждый раздел на уровне как БД, так и экземпляра микросервиса.

Как и в примере с репликами для считывания, масштабирование необходимо выполнять таким образом, чтобы потребители микросервиса не знали об этой детали реализации. Когда потребитель делает запрос к микросервису `Покупатель` на рис. 13.5, нужно, чтобы его запрос динамически направлялся в правильный раздел. Тот факт, что мы внедрили разделение данных, следует рассматривать как внутреннюю деталь реализации этого микросервиса — это дает нам свободу изменять схему разделения или, возможно, полностью заменять разделение.

Рис. 13.5. Запросы направляются в соответствующий экземпляр микросервиса

Другим распространенным примером разделения данных может служить разделение по географическому принципу. У вас может быть один раздел для каждой страны или региона.

Для FoodCo одним из вариантов решения конфликтов в основной базе данных стало разделение данных по странам. Таким образом, клиенты из Ганы попадают в одну БД, а клиенты из Джерси — в другую. Эта модель не имела бы смысла для FoodCo из-за ряда факторов. Основная проблема в том, что FoodCo планирует продолжить географическую экспансию и надеется повысить эффективность за счет возможности обслуживать несколько географических точек из одной системы. Идея о необходимости постоянно создавать новые разделы для каждой страны резко увеличила бы стоимость выхода на новые рынки.

Чаще всего разделение выполняется подсистемой, на которую вы полагаетесь. Например, Cassandra использует сегменты для распределения операций чтения и записи между узлами в заданном «кольце», а Kafka поддерживает распределение сообщений по разделенным топикам.

Ключевые преимущества

Разделение данных очень хорошо масштабируется для транзакционных рабочих нагрузок. Например, если ваша система ограничена в операциях записи, разделение данных может значительно улучшить ситуацию.

Создание нескольких разделов также облегчает последствия сокращения воздействия и объема работ по техническому обслуживанию. Развертывание обновлений может выполняться для каждого раздела, а операции, которые в противном случае потребовали бы простоя, могут оказать меньшее влияние, поскольку они затронут только один раздел. Например, при разделении по географическим регионам операции, способные привести к прерыванию обслуживания, способны выполняться в наименее напряженное время суток, возможно ранним утром. Географическое разделение также может быть очень полезно, если вам необходимо, чтобы данные не покидали определенные юрисдикции, например, чтобы данные, связанные с гражданами ЕС, хранились внутри ЕС.

Секционирование данных хорошо работает при горизонтальном дублировании — каждый раздел может состоять из нескольких узлов, способных выполнять эту работу.

Ограничения

Стоит отметить, что разделение данных имеет ограниченную полезность с точки зрения повышения надежности системы. Если произойдет сбой раздела, запросы, направленные к этому разделу, не выполнятся. Например, если ваша нагрузка равномерно распределена по четырем разделам, а один из них выходит из строя, то 25 % ваших запросов завершатся неудачей. Это не так плохо, как полный провал, но все равно довольно нехорошо. Вот почему для повышения надежности конкретного раздела обычно комбинируют разделение данных с горизонтальным дублированием.

Правильно подобрать ключ раздела может быть непросто. На рис. 13.5 использована тривиальная схема разделения, где мы разделили рабочую нагрузку на основе фамилии клиента. Покупатели с фамилией, начинающейся с букв «А» — «М», переходят в раздел 1, а покупатели с фамилией, начинающейся с букв «Н» — «Я», — в раздел 2. Как я уже отмечал, это не очень хорошая стратегия разделения. При разделении данных требуется равномерное распределение нагрузки. Но не стоит ожидать равномерного распределения по схеме, которую я обрисовал. В Китае, например, всегда существовало очень небольшое количество фамилий, и даже сегодня их насчитывается менее 4000. Самые популярные 100 фамилий, которые приходятся более чем на 80 % населения, приходятся на диапазон первых букв «Н» — «Я» в мандаринском языке. Это пример схемы масштабирования, которая вряд ли обеспечит равномерное распределение нагрузки, а в разных странах и культурах может дать совершенно различные результаты.

Более разумной альтернативой может быть разделение на основе уникального идентификатора, присвоенного каждому клиенту при регистрации. Это с гораздо большей вероятностью даст равномерное распределение нагрузки, а также позволит справиться с ситуацией, когда кто-то меняет свое имя.

Добавление новых разделов к существующей схеме обычно выполняется без особых проблем. Например, добавление нового узла в кольцо Cassandra не требует какой-либо ручной перебалансировки данных. Вместо этого в Cassandra есть встроенная поддержка динамического распределения данных по узлам. Kafka также позволяет довольно легко добавлять новые разделы постфактум, хотя сообщения, уже находящиеся в разделе, не перемещаются, но производители и потребители могут получать динамические уведомления.

Все усложняется, когда вы понимаете, что ваша схема разделения просто вам не подходит, как в случае с описанной выше схемой на основе фамилий. В такой ситуации, возможно, придется потрудиться. Я помню, как много лет назад общался с клиентом, которому пришлось отключить свою работающую систему на три дня, чтобы изменить схему разделения для основной БД.

Также можно решить проблему с запросами. Искать отдельную запись легко, так как можно просто применить функцию хеширования, чтобы определить, в каком экземпляре должны находиться данные, а затем извлечь их из правильного сегмента. Но как насчет запросов, которые охватывают данные в нескольких узлах, например, поиск всех клиентов старше 18 лет? Если необходимо запросить все сегменты, вам нужно либо запросить каждый отдельный сегмент и объединить его в памяти, либо создать альтернативное хранилище для чтения, где доступны оба набора данных. Часто запросы по сегментам обрабатываются асинхронным механизмом с использованием кэшированных результатов. Mongo, например, использует модель map/reduce для выполнения этих запросов.

Как вы, возможно, поняли, масштабирование БД для операций записи — это этап, на котором все становится очень сложным и где возможности разных БД действительно начинают различаться. Я часто вижу, как люди меняют технологию баз данных, когда сталкиваются с ограничениями по масштабированию существующего объема записи. Если это случилось с вами, то покупка более мощной машины часто самый быстрый способ решения проблемы, но параллельно стоит рассмотреть другие типы баз данных, которые лучше справятся с вашими требованиями. Учитывая обилие различных типов доступных баз данных, выбор новой БД может оказаться непростой задачей, но в качестве отправной точки я настоятельно рекомендую приятную и лаконичную книгу по NoSQL[1], предоставляющую обзор различных стилей доступных NoSQL БД — от высокореляционных хранилищ, таких как базы данных графов, до хранилищ документов, столбцов и значений ключей.

---

[1] *Прамодкумар Дж. С., Мартин Ф.* NoSQL: новая методология разработки нереляционных баз данных.

По сути, разделение данных — это очень трудоемкая задача, особенно потому, что она может потребовать значительных изменений в данных вашей существующей системы. Однако код приложения, скорее всего, будет затронут лишь незначительно.

Функциональная декомпозиция

С помощью функциональной декомпозиции выполняется извлечение функциональности и масштабирование происходит независимо. Извлечение функциональности из существующей системы и создание нового микросервиса — это почти канонический пример функциональной декомпозиции. На рис. 13.6 показан пример из MusicCorp, в котором функциональность заказа извлекается из основной системы, чтобы позволить нам масштабировать ее отдельно от остальных сервисов.

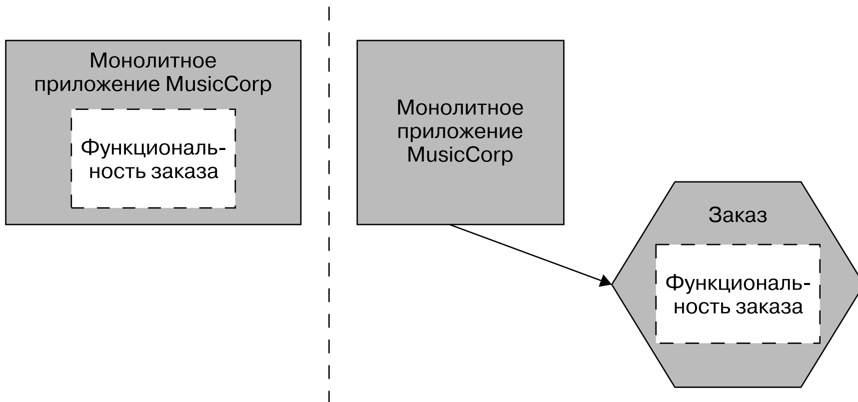


Рис. 13.6. Микросервис Заказ извлекается из существующей системы MusicCorp

В случае с FoodCo это станет дальнейшим направлением ее развития. Компания исчерпала потенциал вертикального масштабирования, использовала горизонтальное дублирование, насколько это было возможно, отказалась от разделения данных — осталось только начать двигаться в сторону функциональной декомпозиции. Ключевые данные и рабочие нагрузки удаляются из основной системы и основной БД, чтобы это изменение произошло. Было выявлено несколько выгодных решений, в том числе перенос данных, связанных с доставкой, и меню из основной БД в выделенные микросервисы. Подобный подход дает дополнительное преимущество, поскольку создает возможности для растущей команды доставки FoodCo начать организовываться вокруг владения этими новыми микросервисами.

Реализация

Я не буду заострять внимание на этом механизме масштабирования, поскольку мы уже детально рассмотрели основы микросервисов. Более подробное обсуждение того, как можно добиться подобных изменений, см. в главе 3.

Ключевые преимущества

Тот факт, что мы разделили различные типы рабочих нагрузок, означает, что теперь можно правильно настроить базовую инфраструктуру, необходимую для нашей системы. Прошедшая декомпозицию функциональность, которая используется лишь изредка, может быть отключена, когда в ней нет необходимости. Функциональность со скромными требованиями нагрузки может быть развернута на небольших машинах. С другой стороны, для ограниченного в настоящее время функционала может потребоваться больше аппаратного обеспечения, возможно комбинация функциональной декомпозиции с одной из других осей масштабирования, например, запуском нескольких копий нашего микросервиса.

Возможность корректировать размер инфраструктуры, необходимой для выполнения рабочих нагрузок, дает больше гибкости в оптимизации стоимости инфраструктуры, необходимой для запуска системы. Это ключевая причина, по которой крупные поставщики SaaS так активно используют микросервисы, поскольку умение найти правильный баланс затрат на инфраструктуру поможет повысить рентабельность.

Сама по себе функциональная декомпозиция не сделает систему более надежной, но она по крайней мере открывает перед нами возможность построить систему, способную выдержать частичный отказ функциональности, что мы более подробно рассмотрели в главе 12.

Если вы выбрали путь функциональной декомпозиции микросервисов, у вас будет больше возможностей использовать различные технологии, способные масштабировать отделенный микросервис. Например, можно было бы перенести функциональность на язык программирования и среду выполнения, которые более эффективны для типа выполняемой вами работы, или, возможно, вы могли бы перенести данные в БД, более подходящую для вашего трафика чтения или записи.

Хотя в этой главе мы сосредоточились в основном на масштабировании в контексте нашей программной системы, функциональная декомпозиция также облегчает масштабирование организации. К этой теме мы вернемся в главе 15.

Ограничения

В главе 3 мы подробно рассмотрели, что разделение функциональных возможностей может быть сложным мероприятием и вряд ли принесет пользу в краткосрочной перспективе. Из всех рассмотренных нами форм масштабирования именно эта, вероятно, окажет наибольшее влияние на код вашего

приложения — как на фронтенд, так и на бэкенд. Она может потребовать значительного объема работы на уровне данных, если вы также решите перейти на микросервисы.

В конечном счете увеличится количество запущенных микросервисов, что повысит общую сложность системы, а это потенциально приведет к увеличению количества элементов, требующих обслуживания и поддержки, повышения надежности и масштабирования. В общем, когда дело доходит до масштабирования системы, я стараюсь выжать по максимуму из других возможностей, прежде чем рассматривать функциональную декомпозицию. Мое мнение на этот счет может измениться, если переход на микросервисы потенциально принесет с собой множество других полезных организации вещей. Например, в случае с FoodCo ключевым моментом стало стремление расширить команду разработчиков, чтобы поддерживать большее количество стран и предоставлять больше функций, поэтому переход на микросервисы дает компании шанс решить некоторые проблемы масштабирования не только системы, но и самой организации.

Сочетание моделей

Одной из основных движущих сил оригинального куба масштабирования было стремление помешать нам мыслить узко в терминах одного типа масштабирования и помочь понять, что часто имеет смысл масштабировать свое приложение по нескольким осям, в зависимости от потребностей. Давайте вернемся к описанному на рис. 13.6 примеру. Мы извлекли функциональность `Заказа`, и теперь она может работать на собственной инфраструктуре. Логичным следующим шагом было бы масштабировать микросервис `Заказ` изолированно, создав несколько его копий, как показано на рис. 13.7.

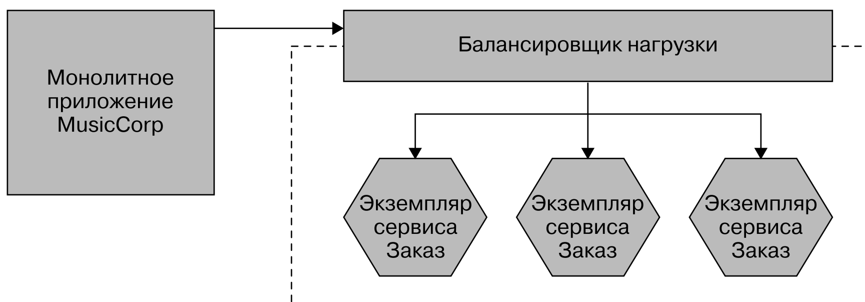


Рис. 13.7. Извлеченный микросервис Заказ теперь дублируется для масштабирования

Далее можно было бы принять решение о запуске разных сегментов микросервиса `Заказ` для разных географических регионов, как показано на рис. 13.8.

Горизонтальное дублирование применяется в пределах каждой географической границы.

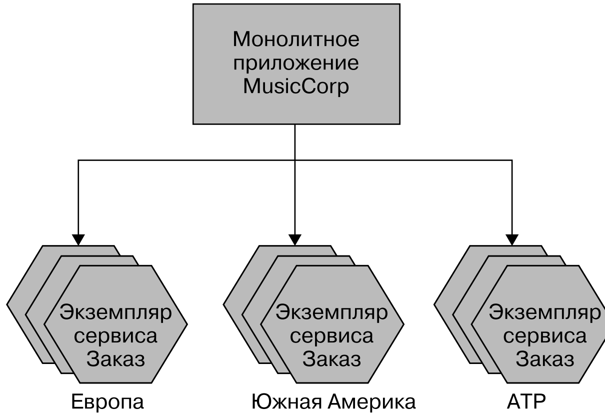


Рис. 13.8. Микросервис Заказ в MusicCorp теперь разделен по географическому принципу с дублированием в каждой группе

Стоит отметить, что при масштабировании вдоль одной оси может стать проще продвинуться по остальным. Например, функциональная декомпозиция сервиса `Заказ` позволяет позже развернуть несколько его дубликатов, а также разделить нагрузку на обработку заказов. Без такой первоначальной функциональной декомпозиции мы были бы ограничены применением этих методов к монолиту в целом.

При масштабировании не обязательно, чтобы оно происходило по всем осям, но мы должны знать, что в нашем распоряжении есть эти различные механизмы. Учитывая данный выбор, важно понимание плюсов и минусов каждого механизма, чтобы определить, какие из них имеют наибольший смысл.

Начните с малого

В книге «Искусство программирования» Дональд Кнут, как известно, сказал:

> Реальная проблема в том, что программисты тратят слишком много времени, заботясь об эффективности в неподходящих местах и в неподходящее время. Преждевременная оптимизация — корень всего зла (или по крайней мере большей его части) в программировании.

Оптимизация системы для решения проблем, которых у нас нет, — это отличный способ потратить драгоценное время, а также чрезмерно усложнить систему вместо того, чтобы заниматься другими видами деятельности. Любая

форма оптимизации должна быть продиктована реальной потребностью. Как мы уже говорили в опдразделе «Надежность» главы 12, добавление новых сложностей в нашу систему также может привести к появлению новых источников уязвимости. Масштабируя одну часть приложения, мы создаем слабость в другой. Микросервис `Заказ`, возможно, теперь работает на своей собственной инфраструктуре, помогая нам лучше справляться с нагрузкой на систему. Но есть еще один микросервис, от которого требуются гарантии доступности, если мы хотим, чтобы система функционировала и стало еще больше инфраструктуры, которой нужно управлять и сделать надежной.

Даже если вам кажется, что вы определили узкое место, процесс экспериментирования необходим, чтобы убедиться, что вы правы и дальнейшая работа оправданна. Меня поражает, как много людей, которые с радостью назвали бы себя математиками и техническими специалистами в одном лице, похоже, не имеют даже элементарного представления о научном методе[1]. Если вы определили проблему, попробуйте выполнить небольшой объем работы для подтверждения, сработает ли предложенное вами решение или нет. В контексте масштабирования систем для обработки нагрузки, например, наличие набора автоматизированных нагрузочных тестов может быть невероятно полезным. Запустите тесты, чтобы получить основу и воссоздать узкое место, с которым вы столкнулись, внесите изменения и понаблюдайте за различиями. Это не ракетостроение, но, даже если в очень малой степени, попытка хотя бы отдаленно использовать научные методы.

CQRS И ИСТОЧНИКИ СОБЫТИЙ

Шаблон CQRS (Command Query Responsibility Segregation — «Разделение ответственности командных запросов») относится к альтернативной модели хранения и запроса информации. Вместо единой схемы для одновременной обработки и извлечения данных, как это обычно бывает, ответственность за чтение и запись обрабатывается по отдельным моделям. Эти независимые модели чтения и записи, реализованные в коде, могут быть развернуты как индивидуальные модули, что дает нам возможность независимо масштабировать операции чтения и записи. Шаблон CQRS часто, хотя и не всегда, используется в сочетании с *источником событий*, где мы проецируем состояние сущности, просматривая историю событий, связанных с ней, вместо того чтобы хранить текущее ее состояние в виде отдельной записи.

Возможно, CQRS делает на уровне приложения нечто очень похожее на то, что делают реплики для считывания на уровне данных, хотя из-за большого количества различных способов реализации CQRS это упрощение.

[1] Не заставляйте меня заводить разговор о людях, которые начинают рассуждать о гипотезах, а затем собирают информацию, чтобы подтвердить свои уже сложившиеся убеждения.

Лично я, хотя и вижу ценность в шаблоне CQRS в некоторых ситуациях, считаю его сложным для качественного выполнения. Я общался с очень умными людьми, которые столкнулись с серьезными проблемами в обеспечении работы CQRS. Таким образом, если вы рассматриваете CQRS как способ масштабировать свое приложение, относитесь к нему как к одной из самых сложных для реализации форм масштабирования. Попробуйте лучше что-то попроще. Например, если вы просто ограничены в операции считывания, репликация — значительно менее рискованный и более быстрый подход для начала. Мои опасения по поводу сложности реализации распространяются и на поиск источников событий — в некоторых ситуациях он подходит действительно хорошо, но сопряжен с множеством проблем, с которыми необходимо смириться. Оба шаблона требуют от разработчиков значительного сдвига в мышлении, что всегда усложняет задачу. Если вы решите использовать любой из этих шаблонов, просто убедитесь, что эта повышенная когнитивная нагрузка на ваших разработчиков того стоит.

Последнее замечание о CQRS и источниках событий: с точки зрения архитектуры микросервиса решение использовать или не использовать эти методы является внутренней деталью реализации микросервиса. Если вы решили внедрить микросервис, например разделив ответственность за чтение и запись между различными процессами и моделями, это должно быть незаметно для потребителей микросервиса. Если входящие запросы необходимо перенаправить на соответствующую модель в зависимости от выполняемого запроса, сделайте это обязанностью микросервиса, реализующего CQRS. Скрытие этих деталей реализации от потребителей дает вам большую гибкость, позволяя впоследствии передумать или изменить способ использования этих шаблонов.

Кэширование

Кэширование — это широко используемая оптимизация производительности, при которой предыдущий результат некоторой операции сохраняется, чтобы последующие запросы могли использовать это сохраненное значение, а не тратить время и ресурсы на пересчет значения.

В качестве примера рассмотрим микросервис `Рекомендации`, который должен проверять уровень запасов, прежде чем рекомендовать товар, — нет никакого смысла рекомендовать то, чего нет на складе! Но мы решили сохранить локальную копию уровней запасов в сервисе `Рекомендации` (форма кэширования на стороне клиента). Чтобы уменьшить задержку операций, мы избегаем необходимости проверять уровни запасов всякий раз, когда нужно что-то порекомендовать. Источником достоверности для уровней запасов становится микросервис `Запасы`, он же считается *источником* клиентского кэша в микросервисе `Рекомендации`. Когда сервису `Рекомендации` необходимо просмотреть уровень запасов, он может сначала просмотреть свой локальный кэш. Если нужная запись найдена, это считается *попаданием в кэш*. Если данные не найдены — это *кэш-промах*, что приводит к необходимости извлечения информации из нижестоящего микросервиса `Запасы`. Поскольку данные в источнике, конечно, могут меняться, нам нужен какой-то способ *аннулировать* записи в кэше сервиса `Рекомендации`, чтобы

мы знали, когда локально кэшированные данные настолько устарели, что их больше нельзя использовать.

Кэши могут хранить как результаты простых поисков, как в этом примере, так и любую часть данных, например результат сложного вычисления. Можно использовать кэширование для повышения производительности системы в рамках сокращения задержек, масштабирования приложения, а в некоторых случаях даже для повышения надежности системы. В совокупности с тем фактом, что существует множество механизмов аннулирования и мест, где есть возможность кэширования, это означает, что нам предстоит обсудить ряд аспектов, когда речь заходит о кэшировании в микросервисной архитектуре. Давайте начнем с того, с какими проблемами могут помочь кэши.

Для производительности

При работе с микросервисами нас часто беспокоят негативное влияние сетевых задержек и затраты на взаимодействие с несколькими микросервисами для получения некоторых данных. Извлечение данных из кэша может значительно здесь помочь, поскольку мы избегаем необходимости выполнения сетевых вызовов, что также положительно влияет на нагрузку на нижестоящие микросервисы. Помимо того что это позволяет избежать сетевых переходов, кэширование уменьшает необходимость создавать данные при каждом запросе. Рассмотрим ситуацию, в которой мы запрашиваем список самых популярных товаров по жанрам. Это может повлечь за собой дорогостоящий JOIN-запрос на уровне БД. Мы могли бы кэшировать результаты этого запроса. В итоге нам нужно будет повторно создавать результаты только тогда, когда кэшированные данные станут недействительными.

Для масштабирования

Если у вас есть возможность перенаправлять операции чтения в кэши, вы можете избежать конфликтов в отдельных частях вашей системы, что позволит ей лучше масштабироваться. Примером может служить использование реплик для считывания БД. Трафик чтения обслуживается репликами, что снижает нагрузку на основной узел БД и позволяет эффективно масштабировать операции чтения. Операции считывания реплики выполняются из данных, которые могут быть устаревшими. Реплика в конечном счете будет обновлена путем репликации с первичного узла на узел реплики, эта форма аннулирования кэша обрабатывается автоматически технологией БД.

В более широком смысле кэширование для масштабирования полезно в любой ситуации, в которой источник становится спорным моментом. Размещение кэшей между клиентами и источником может снизить нагрузку на последний, позволяя ему лучше масштабироваться.

Для надежности

Наличие всего набора данных, доступных в локальном кэше, означает, что вы можете работать, даже если источник недоступен, это, в свою очередь, способствует повышению надежности системы. Есть несколько моментов, на которые следует обратить внимание в отношении кэширования для обеспечения надежности. Вам, вероятно, потребуется настроить механизм аннулирования кэша так, чтобы он не удалял в автоматическом режиме устаревшие данные, а хранил их в кэше до момента обновления. В противном случае, поскольку данные становятся недействительными, они будут удалены, что приведет к кэш-промаху и невозможности получения каких-либо данных из-за недоступности источника. Это означает, что вы должны быть готовы к чтению довольно старых данных, если источник находится в автономном режиме. Для некоторых ситуаций это нормально, но в других может стать проблемой.

По сути, использование локального кэша для обеспечения надежности в ситуации, когда источник недоступен, означает, что вы отдаете предпочтение доступности, а не согласованности.

Метод, который, как я видел, использовался в *The Guardian*, а затем и в других компаниях, заключался в периодическом сканировании существующего «живого» сайта для создания его статической версии, которую можно было предоставить в случае сбоя. Хотя отсканированная версия была не такой свежей, как кэшированный контент, предоставляемый из «живой» системы, однако это могло гарантировать отображение сайта.

Где кэшировать

Как мы уже неоднократно говорили, микросервисы предоставляют дополнительные возможности. И это абсолютно верно в случае с кэшированием. Существует множество мест, где можно реализовать кэширование. У различных расположений кэша, о которых я расскажу здесь, имеются компромиссы, и то, какую оптимизацию вы пытаетесь провести, скорее всего, укажет на наиболее подходящее для вас расположение кэша.

Чтобы изучить доступные варианты кэширования, давайте вернемся к ситуации из раздела «Проблемы декомпозиции данных» главы 3, где мы извлекали информацию о продажах в MusicCorp. На рис. 13.9 микросервис `Продажи` ведет учет проданных товаров. Он отслеживает только идентификатор проданного товара и временную метку продажи. Иногда требуется запросить у микросервиса `Продажи` список десяти бестселлеров за предыдущие семь дней.

Проблема в том, что сервис `Продажи` не знает имен записей, только идентификаторы (ID). Бессмысленно говорить: «Бестселлером на этой неделе был ID 366548 и мы продали 35 345 экземпляров!» Нам также необходимо знать

название компакт-диска с идентификатором 366548. Эту информацию хранит микросервис `Каталог`. Как показано на рис. 13.9, отвечая на запрос о десяти бестселлерах, микросервис `Продажи` должен запросить имена, соответствующие данным идентификаторам. Давайте посмотрим, как кэширование может нам помочь и какие типы кэшей мы могли бы использовать.

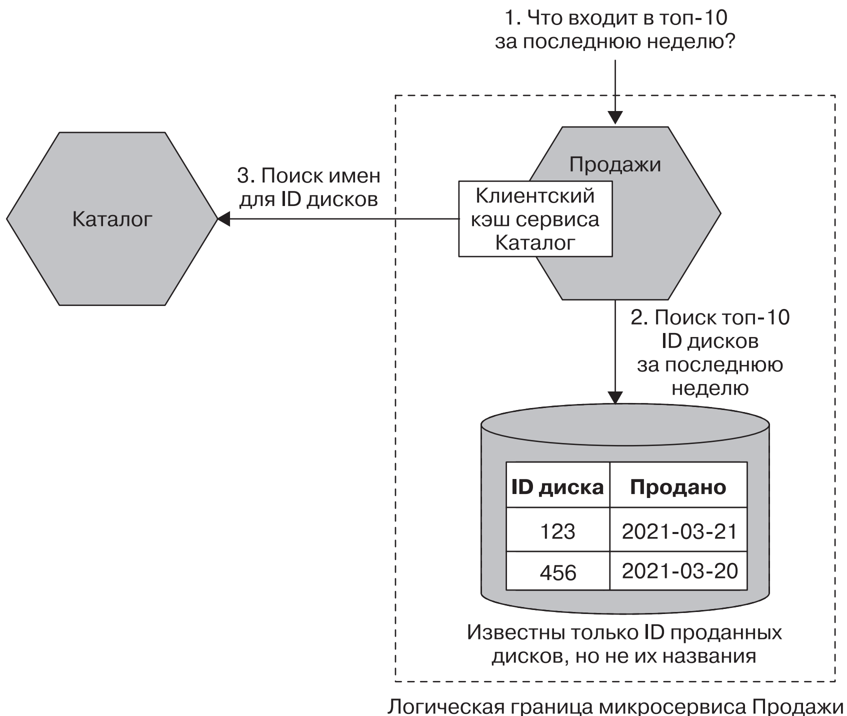


Рис. 13.9. MusicCorp обрабатывает бестселлеры

На стороне клиента

При кэшировании на стороне клиента данные кэшируются за пределами области видимости источника. В нашем примере это можно сделать так же просто, как сохранить в памяти хеш-таблицу с отображением идентификатора и соответствующего ему названия альбома внутри запущенного процесса `Продажи`, как показано на рис. 13.10. Это означает, что при создании десятки лучших товаров любое взаимодействие с сервисом `Каталог` выходит за рамки, предполагая, что мы получаем попадание в кэш для каждого выполняемого поиска. Важно отметить, что клиентский кэш может решить кэшировать только часть информации, получаемой от микросервиса. Например, при запросе информации о компакт-диске

можно получить о нем много информации, но если все, что нас интересует, — название альбома, тогда это единственное, что нужно сохранить в локальном кэше.

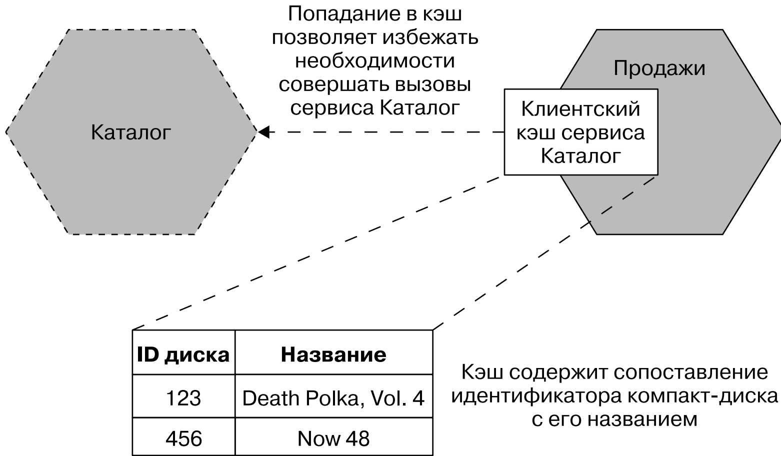


Рис. 13.10. Сервис Продажи содержит локальную копию данных сервиса Каталог

В целом кэши на стороне клиента, как правило, довольно эффективны, поскольку позволяют избежать сетевого вызова нижестоящего микросервиса. Это делает их пригодными для кэширования с целью повышения не только устойчивости, но и надежности.

Однако кэширование на стороне клиента не лишено и некоторых недостатков. Во-первых, вы более ограничены в своих возможностях в отношении механизмов аннулирования — вскоре мы это рассмотрим. Во-вторых, когда происходит много кэширования на стороне клиента, может возникнуть значительная степень несогласованности между клиентами. Рассмотрим ситуацию, в которой у микросервисов `Продажи`, `Рекомендации` и `Акции` есть кэшированные на стороне клиента данные из сервиса `Каталог`. Когда данные в сервисе `Каталог` изменяются, какой бы механизм аннулирования ни применялся, он не гарантирует, что данные обновляются в один и тот же момент времени во всех клиентах. Это означает, что в каждом из этих клиентов одновременно можно было бы видеть разное представление кэшированных данных. Чем больше в системе клиентов, тем серьезнее проблема. Такие методы, как аннулирование на основе уведомлений, которые мы вскоре рассмотрим, помогут уменьшить это явление, но не устранить его.

Еще одним смягчающим фактором для данного подхода станет наличие общего кэша на стороне клиента, возможно, с применением специального инструмента кэширования, например Redis или memcached, как показано на рис. 13.11. Здесь мы избегаем проблемы несогласованности между разными

клиентами. Это также может быть более эффективным с точки зрения использования ресурсов, поскольку мы сокращаем количество копий данных, которыми нам нужно управлять (кэши часто оказываются в памяти, а память нередко становится одним из самых больших ограничений инфраструктуры). Обратная сторона медали — нашим клиентам теперь необходимо совершать перемещение к общему кэшу.

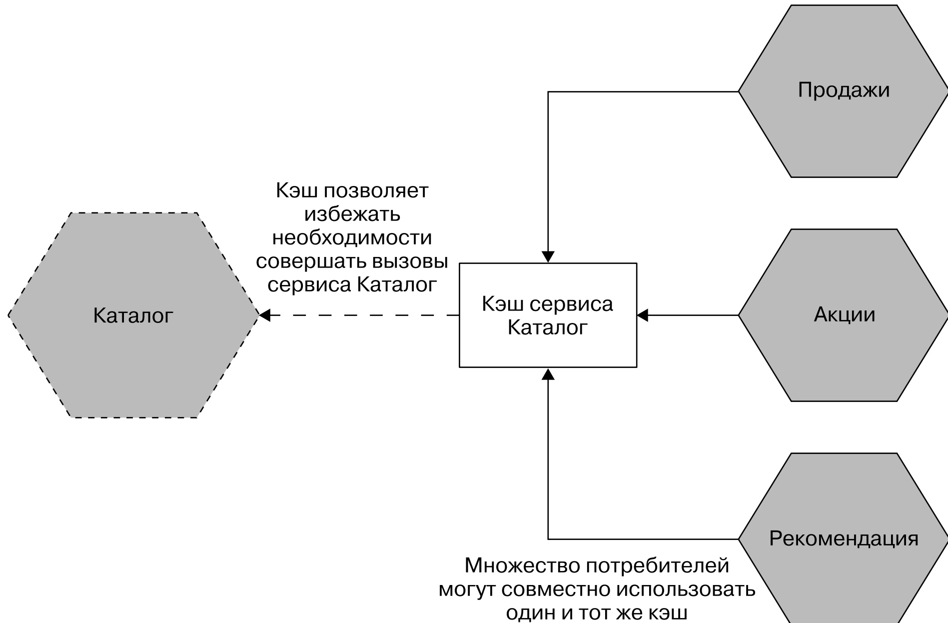


Рис. 13.11. Несколько потребителей сервиса Каталог, использующих один общий кэш

Еще одна вещь, которую следует учитывать, — кто несет ответственность за этот общий кэш. В зависимости от того, кому он принадлежит и как он реализован, подобный общий кэш может размыть границы между кэшированием на стороне клиента и кэшированием на стороне сервера, что мы рассмотрим далее.

На стороне сервера

На рис. 13.12 показан пример с топ-десять продаж, использующих кэширование на стороне сервера. Здесь микросервис `Каталог` сам поддерживает кэш от имени своих потребителей. Когда микросервис `Продажи` запрашивает названия компакт-дисков, эта информация прозрачно обрабатывается кэшем.

В данном случае микросервис `Каталог` несет полную ответственность за управление кэшем. Из-за характера реализации этих кэшей (например, структура данных в памяти или локальный выделенный узел кэширования) проще

реализовать более замысловатые механизмы аннулирования кэша. Например, кэши со сквозной записью (которые мы вскоре рассмотрим) было бы намного проще применить в этой ситуации. Наличие кэша на стороне сервера также позволяет избежать проблемы, связанной с тем, что разные пользователи видят разные кэшированные значения, которые могут возникать при кэшировании на стороне клиента.

Хотя с точки зрения потребителя такое кэширование невидимо (это проблема внутренней реализации), это не означает, что мы должны реализовывать его путем кэширования в коде экземпляра микросервиса. Можно было бы, например, поддерживать обратный прокси-сервер в пределах логической границы микросервиса, использовать скрытый узел Redis или перенаправлять запросы чтения на реплику базы данных для считывания.

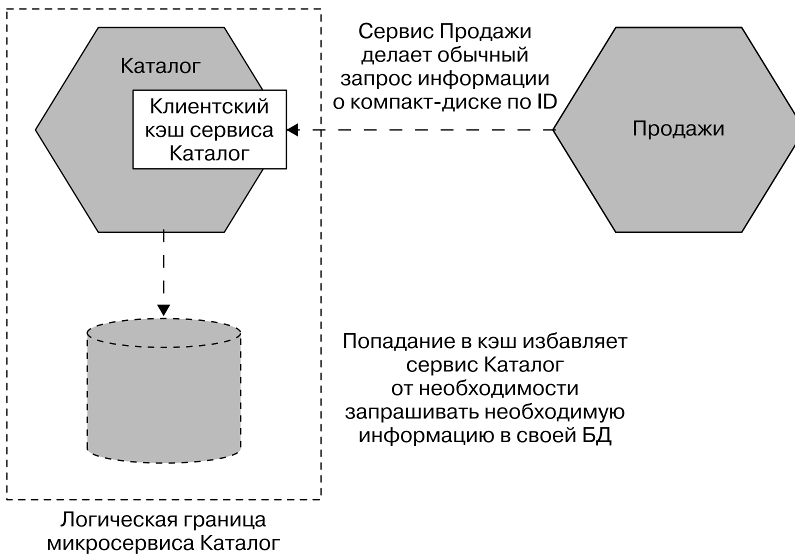


Рис. 13.12. В сервисе Каталог реализовано внутреннее кэширование, что делает процесс невидимым для потребителей

Основная проблема подобной формы кэширования в том, что она сокращает возможности оптимизации задержек, поскольку потребителям все еще требуется обращаться к микросервису. При кэшировании по периметру микросервиса или вблизи него нам не нужно выполнять дополнительные дорогостоящие операции (например, запросы к БД), кроме вызова. Это также снижает эффективность такого вида кэширования для любой формы надежности.

Учитывая вышесказанное, может показаться, что данная форма кэширования менее полезна. Но прозрачное повышение производительности для всех

потребителей микросервиса из-за реализациии кэширования внутри него представляет большую ценность. Микросервис, который широко используется в организации, может извлечь огромную выгоду от внедрения некоторой формы внутреннего кэширования, что, возможно, улучшит время отклика для ряда потребителей, а также позволит микросервису более эффективно масштабироваться.

В случае сценария с топ-10 нам пришлось бы подумать, а сможет ли помочь эта форма кэширования. Решение будет зависеть от того, что нас больше всего беспокоит. Если речь идет о сквозной задержке операции, сколько времени сэкономит кэш на стороне сервера? Кэширование на стороне клиента, скорее всего, даст нам больший прирост производительности.

Кэш запросов

В кэше запросов можно хранить кэшированный ответ на исходный запрос. Так, например, на рис. 13.13 мы храним актуальные топ-10 записей, а последующие запросы приводят к возвращению кэшированного результата. Не требуется никаких поисков в данных сервиса `Продажи`, никаких перемещений к сервису `Каталог` — это, безусловно, самый эффективный кэш с точки зрения оптимизации скорости.

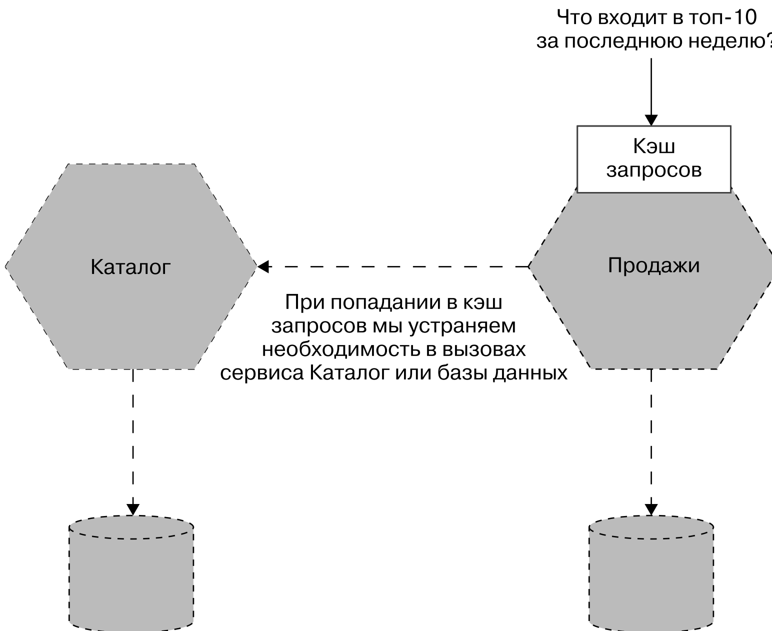


Рис. 13.13. Кэширование результатов запроса топ-10

Преимущества здесь очевидны. Прежде всего, это очень эффективно. Однако мы должны признать, что данная форма кэширования довольно специфична. Мы кэшировали только результат этого конкретного запроса. В то время как другие операции, попадающие в сервис `Продажи` или `Каталог`, не будут попадать в кэш и, следовательно, никоим образом не выиграют от этой формы оптимизации.

Аннулирование

> В информатике есть только две трудные задачи: аннулирование кэша и именование элементов.
>
> *Фил Карлтон*

Аннулирование — это процесс, с помощью которого мы удаляем данные из кэша. Это идея, которая проста по концепции, но сложна в исполнении хотя бы по той причине, что существует множество вариантов ее реализации и компромиссов, которые необходимо учитывать с точки зрения использования устаревших данных. По сути, однако, все сводится к определению, в каких ситуациях часть кэшированных данных должна быть удалена. Иногда это происходит из-за поступления сообщения о наличии новой версии фрагмента данных. В иных случаях это может потребовать от нас предположить, что кэшированная копия устарела, и извлечь новую копию из источника.

Учитывая возможности, связанные с аннулированием, я думаю, что рассмотреть несколько вариантов, которые вы могли бы использовать в микросервисной архитектуре, будет хорошей идеей. Однако, пожалуйста, не считайте это исчерпывающим обзором!

Время жизни (TTL)

Это один из самых простых механизмов, используемых для аннулирования кэша. Предполагается, что каждая запись в кэше действительна только в течение определенного времени. По его истечении данные становятся недействительными и извлекается новая копия. Можно указать период действия, используя простое время жизни (time to live, TTL) — таким образом, значение TTL 5 минут означает, что кэш точно предоставит содержащиеся в нем данные в течение 5 минут, после чего кэшированная запись считается недействительной и требуется новая копия. Вариации на эту тему могут включать использование временной метки для определения истечения срока действия, что в некоторых ситуациях может оказаться более эффективным, особенно если вы выполняете считывание через несколько уровней кэша.

HTTP поддерживает как TTL (через заголовок `Cache-Control`), так и временную метку через заголовок `Expires` для ответов, что может быть невероятно полезно. Это означает, что сам источник может сообщить нижестоящим клиентам, как долго они должны считать данные актуальными. Возвращаясь к микросервису `Запасы`, мы могли бы представить ситуацию, в которой сервис `Инвентаризация` выдает более короткий TTL для уровней запасов быстро продающихся товаров или для продукции, которой почти нет на складе. Для товаров, продающихся нечасто, можно определить более длительный TTL. Такой подход представляет собой несколько расширенное использование элементов управления HTTP-кэшем, и я бы занялся настройкой управления кэшем на основе каждого ответа только при настройке эффективности кэша. Простой универсальный TTL для любого заданного типа ресурсов станет разумной отправной точкой.

Даже если вы не используете HTTP, идея источника, дающего клиенту подсказки относительно того, как (и если) данные должны быть кэшированы, является действительно мощной концепцией. То есть нет нужды гадать об этих вещах на стороне клиента — вы действительно в состоянии сделать осознанный выбор, как обрабатывать фрагмент данных.

HTTP обладает более продвинутыми возможностями кэширования, и мы сейчас рассмотрим условные методы GET в качестве примера.

Одна из проблем аннулирования на основе TTL заключается в том, что, хотя такая модель проста в реализации, она представляет собой довольно грубый инструмент. Если запросить новую копию данных с пятиминутным TTL, а секунду спустя данные в источнике изменятся, то кэш будет работать с устаревшими данными в течение всего оставшегося времени. Таким образом, простота реализации должна быть сбалансирована с тем, насколько вы допускаете работу с устаревшими данными.

Условные GET-запросы

Мы только что коснулись вопроса, что HTTP предоставляет возможность указывать заголовки `Cache-Control` и `Expires` в ответах, чтобы обеспечить более разумное кэширование на стороне клиента. Но при работе непосредственно с HTTP у нас есть еще один вариант в арсенале этого протокола: теги сущностей, или ETag. ETag позволяет определить, изменилось ли значение ресурса. При обновлении записи клиента URI ресурса остается тем же, но значение отличается, поэтому ожидается, что и ETag изменится. Это очень полезно, когда мы используем то, что называется *условным GET*. При выполнении GET-запроса можно указать дополнительные заголовки, указывающие сервису отправлять нам ресурс только при соблюдении определенных критериев.

Например, представим, что мы извлекаем запись клиента и ее ETag возвращается как `o5t6fkd2sa`. Позже, возможно, из-за того, что директива `Cache-Control`

указала нам, что ресурс следует считать устаревшим, необходимо убедиться в получении последней версии. При последующем GET-запросе можно передать `If-None-Match: o5t6fkd2sa`. Это укажет серверу, что нам нужен ресурс с указанным URI, если только он уже не соответствует этому значению ETag. Если у нас имеется обновленная версия, сервис отправляет ответ `304 Not Modified`, сообщая о наличии у нас последней версии. Если доступна новейшая модификация, мы получаем ответ `200 OK` с измененным ресурсом и новым ETag для него.

Конечно, при условном GET-запросе мы все равно отправляем запрос от клиента к серверу. Если вы применяете кэширование, чтобы сократить количество перемещений по сети туда и обратно, это вам не сильно поможет. Но это очень полезно, когда необходимо избежать лишних затрат на ненужную регенерацию ресурсов. При аннулировании на основе TTL клиент запрашивает новую копию ресурса, даже если последний не изменился, — микросервис, получающий данный запрос, должен повторно создать ресурс, даже если он в итоге будет точно таким же, как тот, что уже есть у клиента. Если цена создания ответа высока, возможно, по причине необходимости выполнения ресурсоемкого набора запросов к БД, то условные GET-запросы могут быть эффективным механизмом.

На основе уведомлений

При аннулировании на основе уведомлений мы используем события, чтобы помочь абонентам узнать, нужно ли аннулировать их записи в локальном кэше. На мой взгляд, это самый элегантный механизм аннулирования, хоть он и сложнее, чем аннулирование на основе TTL.

На рис. 13.14 показано, что микросервис `Рекомендации` поддерживает кэш на стороне клиента. Записи в кэш становятся недействительными, когда микросервис `Запасы` запускает событие `Изменение запасов`, давая сервису `Рекомендации` (или любому другому подписчику этого события) знать, что уровень запасов указанного товара увеличился или уменьшился.

Основное преимущество данного механизма в том, что он уменьшает потенциальное окно, в котором кэш предоставляет устаревшие данные. Окно ограничивается временем, необходимым для отправки и обработки уведомления. И в зависимости от используемого вами механизма это может быть довольно быстро.

Недостатком является сложность реализации. Необходимо, чтобы источник мог отправлять уведомления, а заинтересованные стороны — отвечать на них. Теперь это естественное место для использования чего-то вроде брокера сообщений, поскольку данная модель четко вписывается в типичные взаимодействия в стиле издатель — подписчик, предоставляемые многими брокерами. Дополнительные гарантии, которые предоставляет нам брокер, также могут оказаться полезными. Тем не менее, как мы уже обсуждали в разделе «Брокеры сообщений» главы 5, управление промежуточным ПО обмена сообщениями со-

пряжено с накладными расходами, и это может оказаться излишним, если вы используете брокер только для этой цели. Однако, если вы применяете брокеры для других форм взаимодействия между микросервисами, было бы разумно воспользоваться уже имеющейся под рукой технологией.

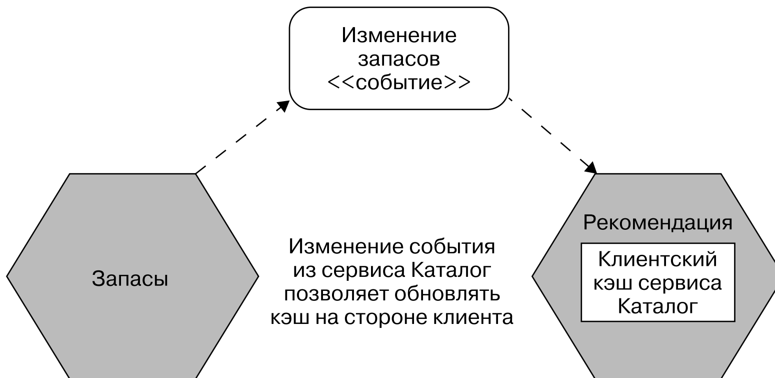


Рис. 13.14. Сервис Перечень запускает событие Изменение запасов, которое сервис Рекомендации может использовать для обновления своего локального кэша

Одна из проблем, о которой следует помнить при использовании аннулирования на основе уведомлений, заключается в том, что вам может потребоваться узнать, действительно ли работает механизм уведомления или нет. Рассмотрим ситуацию, в которой мы некоторое время не получали никаких событий `Изменение запасов` из сервиса `Запасы`. Получается, что мы не продавали товары или не пополняли запасы за это время? Возможно. А еще это может означать, что механизм уведомлений не работает и обновления просто больше не приходят. Если это вызывает беспокойство, то можно отправить событие «пульс» через тот же механизм уведомлений — в нашем случае сервис `Рекомендации`, — чтобы сообщить абонентам, что уведомления все еще приходят. Если событие «пульс» не получено, клиент, предположив о наличии проблемы, может выполнить наиболее подходящие действия, например сообщить пользователю, что он видит устаревшие данные, или просто отключить функциональность.

Вам также необходимо учитывать, что содержится в уведомлении. Если в нем всего лишь говорится «это значение изменилось», без указания, в чем заключается изменение, то при получении уведомления потребителю придется перейти к источнику и извлечь новые данные. С другой стороны, если уведомление содержит текущее состояние данных, то потребители могут загружать их непосредственно в свой локальный кэш. Наличие уведомления, содержащего больше данных, скорее всего, вызовет проблемы с размером, а также имеется

риск слишком широкого раскрытия конфиденциальных данных. Ранее мы более подробно исследовали этот компромисс, рассматривая событийную коммуникацию в разделе «Что входит в событие» главы 4.

Сквозное кэширование

При сквозном кэшировании данные обновляются одновременно с состоянием в источнике. «Одновременно» значит, что сквозное кэширование становится сложным. Сама реализация механизма сквозной записи в кэше на стороне сервера довольно проста, поскольку вы можете обновить БД и кэш в памяти в рамках одной транзакции без особых трудностей. Но если же кэш находится в другом месте, сложнее рассуждать о том, что означает «одновременно» с точки зрения обновления этих записей.

Из-за подобного нюанса кэширование со сквозной записью обычно используется в микросервисной архитектуре на стороне сервера. Преимущества довольно очевидны — временное окно, в котором клиент видит устаревшие данные, может быть фактически устранено. В то же время кэши на стороне сервера в целом могут быть менее полезными, ограничивая обстоятельства, при которых сквозное кэширование было бы эффективно в микросервисах.

Кэш с последующей записью

При использовании кэша с последующей записью *сначала* обновляется сам кэш, а затем источник. Концептуально думайте о кэше как о буфере. Запись в кэш выполняется быстрее, чем обновление источника. Таким образом, мы записываем результат в кэш, что позволяет ускорить последующие чтения, и полагаем, что источник обновится позже.

Основная проблема данного вида кэширования — потенциальная потеря данных. Если сам кэш ненадежен, мы рискуем потерять данные до того, как они будут записаны в источник. Кроме того, сейчас мы находимся в интересном положении: что есть *источник* в этой ситуации? Можно предположить, что источник — микросервис, откуда берутся эти данные, но, если сначала обновить кэш, кто будет источником фактически? Каков же источник истины? При использовании кэширования важно разделять, какие данные кэшируются (и потенциально устарели), а какие можно считать актуальными. Кэширование с последующей записью в области микросервисов делает этот вопрос гораздо менее понятным.

В то время как кэширование с последующей записью часто используется для оптимизации процесса, я видел, что такой подход гораздо реже применяется для микросервисных архитектур. Отчасти из-за того, что другие, более простые формы кэширования достаточно хороши, но в основном из-за сложности обработки потери незаписанных кэшированных данных.

Золотое правило кэширования

Будьте осторожны с повсеместным кэшированием! Чем больше кэшей между вами и источником свежих данных, тем более устаревшими они могут быть и тем сложнее определить их актуальность. Также становится сложнее определить, где данные должны быть признаны недействительными. Компромисс между кэшированием/балансировкой свежести данных и оптимизацией вашей системы с точки зрения нагрузки или задержки достаточно деликатен, и, если вы не можете легко понять, насколько свежими (или нет) могут быть данные, это становится затруднительным.

Рассмотрим ситуацию, в которой микросервис `Запасы` кэширует уровни запасов. Запросы к сервису `Запасы` для определения уровней запасов могут обрабатываться из кэша на стороне сервера, что, соответственно, ускоряет выполнение запроса. Давайте также предположим, что мы установили TTL для этого внутреннего кэша равным 1 минуте, то есть серверный кэш может отставать от фактического уровня запасов на одну минуту. Теперь выясняется, что мы также кэшируем на стороне клиента внутри сервиса `Рекомендации`, где также используем TTL в 1 минуту. Когда срок действия записи в кэше на стороне клиента истекает, происходит запрос из сервиса `Рекомендации` в `Запасы`, чтобы получить актуальный уровень запасов, но без нашего ведома запрос попадает в кэш на стороне сервера, который, в свою очередь, к этому моменту может оказаться устаревшим на одну минуту. Таким образом, в клиентском кэше может храниться запись, старше одной минуты. Это означает, что уровни запасов, используемые сервисом `Рекомендация`, потенциально могут быть устаревшими на *две* минуты, хотя с точки зрения сервиса `Рекомендации` считается, что этим данным не больше минуты.

Существует несколько способов избежать подобных проблем. Лучше поначалу использовать временные метки, чем TTL, но это также пример эффективно вложенного кэширования. Если вы кэшируете результат операции, который основан на кэшированных входных данных, насколько можно быть уверенными, что конечный результат актуален?

Возвращаясь к приведенной ранее известной цитате Кнута, преждевременная оптимизация может вызвать проблемы. Кэширование добавляет сложности, а мы хотим их избежать. Идеальное количество мест для кэширования равно нулю. Все остальное — оптимизация, которую вы *должны* сделать, но имейте в виду, какую сложность это может принести.

Относитесь к кэшированию в первую очередь как к оптимизации производительности. Кэшируйте в как можно меньшем количестве мест, чтобы было легче определить свежесть данных.

Свежесть или оптимизация

Вернемся к нашему примеру аннулирования на основе TTL. Ранее я говорил, что если запросить новую копию данных с пятиминутным TTL, а через секунду данные в источнике изменятся, то кэш будет работать с устаревшими данными в течение оставшихся 4 минут 59 секунд. Если это неприемлемо, одним из вариантов будет уменьшить TTL, тем самым сократив продолжительность времени, в течение которого система могла работать с устаревшими данными, например, до минуты. Это означает, что окно ожидания сократилось до одной пятой от первоначального значения, но в то же время было выполнено в пять раз больше обращений к источнику, поэтому необходимо учитывать связанные с этим задержки и влияние нагрузки.

Балансом будет понимание требований конечного пользователя и системы в целом. Очевидно, что пользователи всегда будут стремиться работать с самыми свежими данными, но только если система не падает под нагрузкой. Аналогично иногда самым безопасным решением является отключение функций в случае сбоя кэша, чтобы избежать перегрузки источника, вызывающей более серьезные проблемы. Когда дело доходит до точной настройки того, что, где и как кэшировать, вам часто приходится балансировать между несколькими осями масштабирования. Это просто еще одна причина стараться делать все как можно проще: чем меньше кэшей, тем легче вникнуть в систему.

Отравление кэша: поучительная история

Что касается кэширования, мы часто думаем, что если реализуем его неправильно, то худшее, что может случиться, — система некоторое время будет обслуживать пользователей с устаревшими данными. А если система будет всегда работать с устаревшими данными? Еще в главе 12 я говорил о компании AdvertCorp, где велась работа над переносом ряда существующих устаревших приложений на новую платформу при помощи шаблона«Душитель». Этот процесс включал в себя перехват вызовов нескольких более старых приложений и, если эти приложения были перенесены на новую платформу, переадресацию вызовов. Наше новое ПО эффективно работало в качестве прокси-сервера. Трафик для предыдущих приложений, которые мы еще не перенесли, был перенаправлен через актуальную версию к нижестоящим старым приложениям. Для вызовов устаревших приложений были выполнены несколько служебных действий. Например, мы убедились, что к результатам из более старой версии были применены надлежащие заголовки HTTP-кэша.

Однажды, вскоре после обычного рутинного релиза, начало происходить что-то странное. Появилась ошибка, из-за которой небольшое количество страниц не выполняло логическое условие в коде вставки заголовка кэша, в результате чего заголовок вообще не менялся. К сожалению, это нижестоящее приложение

также некоторое время назад претерпело изменения, позволяющие включить HTTP-заголовок `Expires: Never`. Ранее это не имело никакого эффекта, так как мы переопределяли данный заголовок, а теперь — нет.

Приложение активно использовало Squid для кэширования HTTP-трафика, и мы довольно быстро заметили проблему, так как появилось больше запросов, обходящих сам Squid, чтобы попасть на серверы приложений. Мы исправили код заголовка кэша и выпустили релиз, а также вручную очистили соответствующую область кэша Squid. Однако этого было недостаточно.

Как мы только что обсуждали, можно кэшировать данные в нескольких местах, но иногда наличие большого количества кэшей усложняет вашу жизнь, а не облегчает. Когда дело доходит до предоставления контента пользователям публичного веб-приложения, между вами и вашим клиентом может оказаться несколько кэшей. Возможно, не только вы размещаете на своем веб-сайте что-то вроде сети доставки контента, но и некоторые интернет-провайдеры используют кэширование. Можете ли вы контролировать эти кэши? И даже если бы вы могли, есть один, который вы практически не контролируете, — кэш в браузере пользователя.

Страницы с заголовком `Expires: Never` застревали в кэшах многих наших пользователей и никогда не признавались недействительными, пока кэш не заполнялся или пользователь не очищал его вручную. Очевидно, что мы не могли добиться ни того ни другого. Нашим единственным вариантом было изменить URL-адреса этих страниц, чтобы они были повторно загружены.

Кэширование действительно очень мощное решение, но вам нужно понимать полный путь данных, которые кэшируются от источника к месту назначения, чтобы по-настоящему оценить его сложности и то, что может пойти не так.

Автоматическое масштабирование

Если вам посчастливилось получить полностью автоматизированную инициализацию виртуальных хостов и вы можете полностью автоматизировать развертывание экземпляров своих микросервисов, то у вас есть строительные блоки, позволяющие автоматически масштабировать микросервисы.

Например, масштабирование может быть вызвано хорошо известными тенденциями. Возможно, вы знаете, что пиковая нагрузка системы приходится на период с 9 утра до 5 вечера, поэтому дополнительные экземпляры имеет смысл запускать в 8:45 утра и отключать их в 5:15 вечера. Если вы используете что-то вроде AWS (содержащий очень хорошую встроенную поддержку автоматического масштабирования), отключение неиспользуемых экземпляров поможет сэкономить деньги. Вам понадобятся данные, чтобы понять, как меняется нагрузка с течением времени, изо дня в день и от недели к неделе. У некоторых компаний также есть очевидные сезонные циклы, поэтому вам могут понадобиться возвращающиеся назад данные, чтобы сделать правильные выводы.

С другой стороны, можно реагировать, вызывая дополнительные экземпляры, когда происходит увеличение нагрузки или сбой экземпляра, и удалять экземпляры, когда они вам больше не нужны. Очень важно знать, как быстро вы сможете провести масштабирование, когда обнаружите тенденцию к росту. Если вы знаете, что получите уведомление об увеличении нагрузки всего за пару минут, а масштабирование займет не менее 10 минут, тогда необходимо запомнить, что требуется зарезервировать дополнительные мощности для преодоления этого разрыва. Наличие хорошего набора нагрузочных тестов здесь практически необходимо. Их можно применять для проверки созданных правил автоматического масштабирования. Если у вас нет тестов, воспроизводящих различные нагрузки, которые вызовут масштабирование, то только во время эксплуатации системы вы узнаете, что что-то пошло не так. А последствия сбоев не самые радужные!

Новостной сайт — отличный пример того типа бизнеса, в котором может понадобиться сочетание прогнозирующего и реактивного масштабирования. На последнем новостном сайте, над которым я работал, мы видели очень четкие ежедневные тенденции: просмотры росли с утра до обеда, а затем начинали снижаться. Эта картина повторялась изо дня в день, и в выходные трафик, как правило, был ниже. Это дало нам довольно четкую тенденцию, способную привести к превентивному масштабированию ресурсов, будь то в сторону увеличения или уменьшения. С другой стороны, крупная новость может вызвать неожиданный всплеск, требующий больших мощностей и часто в короткие сроки.

На самом деле я заметил, что автоматическое масштабирование используется гораздо чаще для обработки сбоев экземпляров, чем для реагирования на условия загрузки. AWS позволяет указывать такие правила, как «В этой группе должно быть не менее пяти экземпляров», чтобы в случае отказа одного из них автоматически запускался новый. Я видел, как такой подход приводил к забавной игре в «ударь крота», когда кто-то забывал отключить правило, а затем пытался удалить экземпляры для обслуживания, но постоянно видя, что они продолжают возвращаться!

И реактивное, и прогнозирующее масштабирование очень полезны и могут способствовать вашей экономичности, если вы используете платформу, позволяющую платить только за используемые вычислительные ресурсы. Но они также требуют тщательного изучения имеющихся у вас данных. Я бы предложил сначала использовать автомасштабирование для условий сбоя, пока вы собираете данные. Как только вы захотите запустить автоматическое масштабирование, чтобы справиться с повышением нагрузки, убедитесь, что уменьшения масштаба происходят слишком быстро. В большинстве ситуаций иметь под рукой вычислительной мощности больше, чем требуется, гораздо лучше, чем не иметь достаточно мощностей!

Начинаем все сначала

Архитектура, с которой вы начинаете, может оказаться совсем не той, что поможет вам продолжать работать, когда системе придется обрабатывать очень разные объемы нагрузки. Как мы уже видели, существуют некоторые формы масштабирования, способные оказать крайне ограниченное влияние на архитектуру вашей системы, например, вертикальное масштабирование и горизонтальное дублирование. Однако в определенные моменты вам следует сделать что-то довольно радикальное, чтобы изменить архитектуру системы для поддержки следующего уровня роста.

Вспомните историю Gilt, затронутую в подразделе «Изолированное выполнение» главы 8. Простое применение монолитного приложения Rails хорошо проявило себя в течение двух лет в Gilt. Бизнес компании становился все более успешным, что означало рост числа клиентов и увеличение нагрузки, в результате чего компании пришлось перепроектировать приложение, чтобы справиться с наблюдаемой нагрузкой.

Перепроектирование может означать разделение существующего монолита, как это произошло с Gilt, или выбор новых хранилищ данных, которые лучше справляются с нагрузкой. Это также подразумевает и внедрение новых методов, таких как переход от синхронных запросов-ответов к событийным системам, внедрение современных платформ развертывания, изменение целых технологических стеков или всего, что находится между ними.

Существует опасность, что люди увидят необходимость перестройки при достижении определенных пороговых значений масштабирования как причину для создания масштабных проектов с самого начала. Это может вызвать катастрофические последствия. Нередко в начале нового проекта мы не знаем, что хотим построить, и будет ли проект успешным. Нужно иметь возможность быстро экспериментировать и понимать, какой потенциал у нас заложен. При попытке создать масштабную систему заранее мы бы в конечном счете были перегружены огромным объемом работы по подготовке к нагрузке, которая рискует никогда не наступить, вместо того чтобы проанализировать, действительно ли кто-то захочет использовать наш продукт. Эрик Райс рассказывает историю, как он потратил полгода на создание продукта, который никто так и не скачал. Он подумал, что мог бы разместить ссылку на веб-странице, которая выдавала бы код 404, когда люди нажимают на нее, чтобы узнать, есть ли какой-либо спрос, или вместо этого провести шесть месяцев на пляже и получить те же знания!

Необходимость изменения систем, чтобы справиться с масштабированием, не может служить признаком неудачи. Это признак успеха.

Резюме

Как мы видим, какой бы тип масштабирования вы ни искали, микросервисы предоставляют множество различных вариантов с точки зрения подхода к решению проблемы.

Оси масштабирования могут быть полезной моделью в зависимости от того, какие типы масштабирования вам доступны.

Вертикальное масштабирование

В двух словах это означает приобретение более мощной машины.

Горизонтальное дублирование

Наличие нескольких устройств, способных выполнять одну и ту же работу.

Разделение данных

Разделение работы на основе какого-либо атрибута данных, например группы клиентов.

Функциональная декомпозиция

Разделение работы в зависимости от типа, например декомпозиция микросервиса.

Ключевым моментом во многом является понимание того, чего вы хотите, — методы, эффективные при масштабировании для снижения задержки, могут быть не столь эффективны при масштабировании для увеличения объема.

Однако я надеюсь, мне удалось показать, что многие из рассмотренных нами форм масштабирования приводят к увеличению сложности вашей системы. Поэтому важно быть нацеленными на то, что вы пытаетесь изменить, и избегать опасностей преждевременной оптимизации.

Далее мы перейдем к наиболее заметным частям нашей системы — пользовательскому интерфейсу.

ЧАСТЬ III

Люди

ГЛАВА 14

Пользовательские интерфейсы

До сих пор мы по-настоящему не касались темы пользовательского интерфейса. Некоторые из вас, возможно, просто предоставляют своим клиентам холодный, жесткий, мрачный API. Но многие из разработчиков осознают, что хотят создавать красивые, функциональные UI, которые будут радовать клиентов. В конце концов, пользовательский интерфейс — это место, где мы будем объединять все эти микросервисы во что-то имеющее смысл для наших потребителей.

Когда я только начал заниматься вычислительной техникой, мы в основном говорили о больших, тучных программных клиентах, работавших на наших настольных компьютерах. Я провел много часов с Motif, а затем Swing, пытаясь сделать свое ПО как можно более удобным в использовании. Часто эти системы предназначались только для создания локальных файлов и управления ими, но многие из них содержали серверный компонент. Моя первая работа в Thoughtworks заключалась в создании электронной торговой системы на базе Swing. Она была лишь винтиком в большом механизме, который находился на сервере.

Затем появился Интернет. Мы представляли себе, что пользовательские интерфейсы должны быть «тонкими», с большим количеством логики на стороне сервера. Сначала наши серверные программы отображали всю страницу целиком и отправляли ее в клиентский браузер, который делал очень мало. Любые взаимодействия обрабатывались на стороне сервера с помощью методов GET и POST, инициируемых пользователем, переходящим по ссылкам или заполняющим формы. Со временем JavaScript стал более популярным вариантом добавления динамического поведения в UI на основе браузера, и некоторые приложения, возможно, стали такими же «тучными», как старые клиенты на настольных компьютерах. Впоследствии мы наблюдали рост популярности мобильных приложений, и сегодня существует разнообразный ландшафт для

предоставления графических пользовательских интерфейсов (graphical user interface, GUI) пользователям — разные платформы и всяческие технологии для этих платформ. Этот набор технологий дает множество вариантов создания эффективных пользовательских интерфейсов, поддерживаемых микросервисами. Все это и многое другое мы рассмотрим в данной главе.

К цифровым технологиям

За последние пару лет организации начали отходить от мысли, что к веб- или мобильным приложениям следует относиться по-разному. Вместо этого они думают о цифровых технологиях более комплексно. Каков наилучший способ для наших заказчиков воспользоваться предлагаемыми услугами? И что эти пожелания делают с нашей системной архитектурой? Понимание того, что мы не можем точно предсказать, как потребитель в конечном счете будет взаимодействовать с нашими продуктами, привело к внедрению более детализированных API, подобных тем, которые предоставляются микросервисами. Комбинируя возможности, предоставляемые микросервисами различными способами, можно дарить особенные впечатления для наших клиентов в их настольных приложениях, мобильных и носимых устройствах и даже в физической форме, если они посещают наш обычный магазин.

Поэтому думайте о пользовательских интерфейсах как о месте, где сплетаются различные нити возможностей, которые мы хотим предложить своим пользователям. Имея это в виду, как нам собрать все эти нити воедино? Необходимо взглянуть на эту проблему с двух сторон: кто и как. Во-первых, подумаем об организационных аспектах: кто и какую несет ответственность, когда дело доходит до предоставления пользовательских интерфейсов? Во-вторых, мы рассмотрим набор шаблонов, которые можно использовать для реализации этих интерфейсов.

Модели владения

Как обсуждалось в главе 1, традиционная многоуровневая архитектура может вызвать проблемы, когда дело доходит до эффективной доставки ПО. На рис. 14.1 показан пример, в котором ответственность за уровень UI принадлежит одной интерфейсной команде, а работа с сервисами серверной части выполняется другой. В данном примере добавление простого элемента управления предполагает работу трех разных команд. Такого рода многоуровневые организационные структуры могут существенно повлиять на нашу скорость

доставки из-за необходимости постоянной координации изменений и передачи работы между рабочими коллективами.

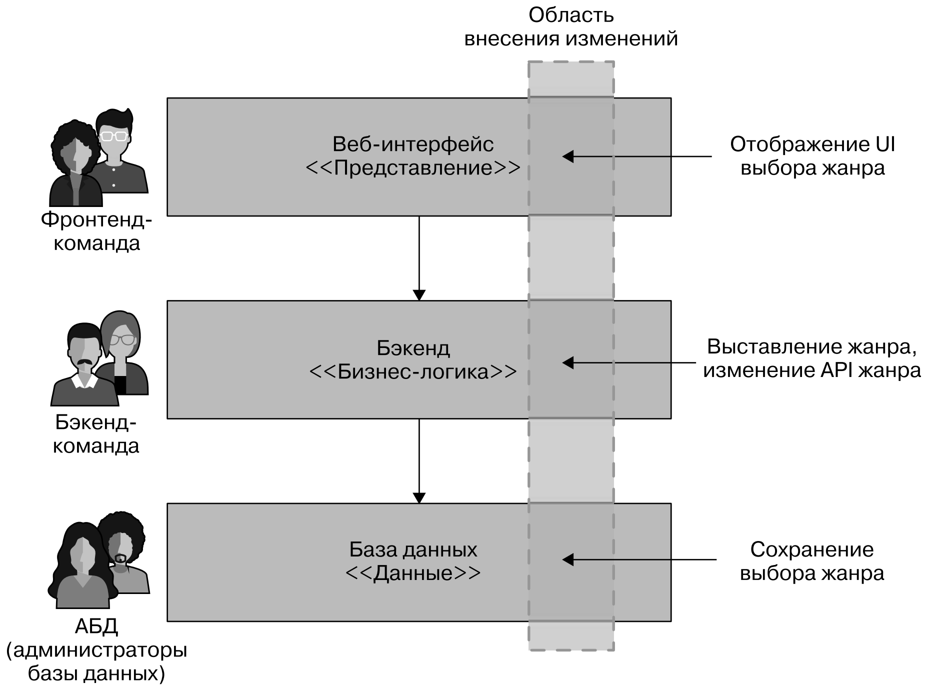


Рис. 14.1. Внесение изменений на всех трех уровнях требует больших усилий

Модель, которую предпочитаю я и которая, по моему мнению, лучше соответствует достижению цели независимого развертывания, заключается в том, чтобы UI был разделен на части и управлялся командой, которая также управляет серверными компонентами, как показано на рис. 14.2. Здесь одна команда в конечном счете несет ответственность за все изменения, которые нам нужно внести, чтобы добавить новый элемент управления.

Команды, полностью владеющие сквозной функциональностью, могут вносить изменения быстрее. Наличие полного владения побуждает каждый коллектив создавать прямой контакт с конечным пользователем ПО. Бэкенд-командам легко потерять представление о том, кто является конечным пользователем.

Несмотря на недостатки, я (к сожалению) по-прежнему рассматриваю выделенную фронтенд-команду как более распространенную организационную модель среди компаний, использующих микросервисы. Почему так?

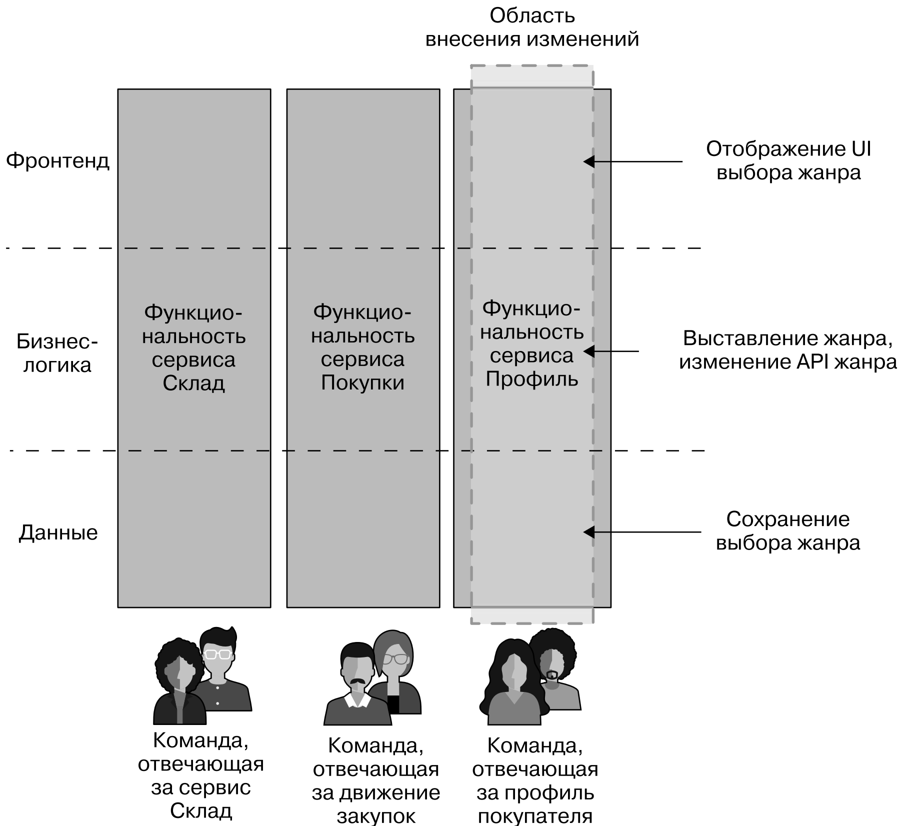


Рис. 14.2. Пользовательский интерфейс разделен на части и принадлежит команде, управляющей также серверными функциями, поддерживающими пользовательский интерфейс

Драйверы для специализированных фронтенд-команд

Стремление к созданию выделенных фронтенд-команд, по-видимому, сводится к трем ключевым факторам: к нехватке специалистов, стремлению к согласованности и техническим проблемам.

В первую очередь создание пользовательского интерфейса требует определенных специальных навыков. Есть аспекты взаимодействия и графического дизайна, а также технические ноу-хау, необходимые для создания отличного веб-интерфейса или нативного приложения. Специалистов с такими навыками бывает трудно найти, и, поскольку такие люди — большая редкость, возникает

соблазн собрать их всех вместе, чтобы убедиться, что они сосредоточены только на своей специализации.

Второй причиной для создания отдельной фронтенд-команды является согласованность. Если у вас есть одна команда, ответственная за разработку клиентского UI, можно будет обеспечить согласованный внешний вид и для вашего интерфейса. Вы используете единый набор элементов управления для решения аналогичных проблем, чтобы UI выглядел и ощущался как нечто цельное.

Наконец, с некоторыми технологиями пользовательского интерфейса может быть сложно работать в немонолитном режиме. Здесь я имею в виду конкретно одностраничные приложения (single-page application, SPA), которые, исторически по крайней мере, нелегко разделить на части. Традиционно веб-интерфейс пользователя состоит из нескольких веб-страниц, и есть возможность переходить с одной страницы на другую. В приложениях SPA вместо этого все приложение отображается на одной веб-странице. Такие фреймворки, как Angular, React и Vue, теоретически позволяют создавать более сложные пользовательские интерфейсы, чем «старомодные» веб-сайты. Позже мы рассмотрим набор шаблонов, способных предоставить различные варианты декомпозиции UI. А в отношении приложений SPA я покажу, как концепция микрофронтенда может позволить вам применять фреймворки SPA, избегая при этом необходимости в монолитном UI.

На пути к потоковым командам

Я думаю, что обзавестись специальной фронтенд-командой в целом будет ошибкой, если вы пытаетесь оптимизировать систему для обеспечения хорошей пропускной способности, — это создает новые точки передачи данных в организации, замедляя работу. В идеале команды должны быть объединены вокруг сквозных функциональных задач, что позволяет каждому коллективу предоставлять новые функции своим клиентам, сокращая при этом объем необходимой координации. Я предпочитаю модель, в которой команда владеет сквозной доставкой функциональности в определенной части домена. Это соответствует тому, что Маттью Скелтон и Мануэль Паис описывают как *потоковые команды* в своей книге «Топологии команд»[1].

> Потоковая команда — это коллектив, ориентированный на единый, значимый поток работы... Команда уполномочена создавать и предоставлять потребительские или пользовательские ценности как можно быстрее, безопаснее и независимей, не требуя передачи другим командам части работы для выполнения.

[1] *Skelton M., Pais M.* Team Topologies. — IT Revolution, 2019.

В некотором смысле мы говорим о *фуллстек-командах* (а не о фуллстек-разработчиках)[1]. Команда, несущая полную ответственность за предоставление функциональности, ориентированной на пользователя, также получит более очевидную прямую связь с конечным пользователем. Слишком часто я видел бэкенд-команды с туманным представлением о том, что делает ПО или что нужно пользователям. Это может вызвать всевозможные недоразумения, когда дело доходит до внедрения новых функций. С другой стороны, командам сквозной разработки гораздо проще установить прямую связь с людьми, использующими созданное ими ПО, — они смогут больше сосредоточиться на гарантиях предоставления пользователям именно того, что им требуется.

Чтобы вы понимали, я провел некоторое время, работая с FinanceCo, успешной и растущей финтех-фирмой, базирующейся в Европе. В FinanceCo практически все команды работают над ПО, которое напрямую влияет на качество обслуживания клиентов, и получают ориентированные на клиента ключевые показатели эффективности (key performance indicators, KPI): успех конкретной команды определяется не столько тем, сколько функций она предоставила, сколько тем, как она улучшает пользовательский опыт. Становится предельно ясно, как модификации могут повлиять на клиентов. Это возможно только благодаря тому, что большинство команд несут прямую ответственность перед пользователями за поставляемое ПО. Чем вы сильнее отдаляетесь от конечного пользователя, тем труднее понять, успешен ли ваш вклад, и в конечном счете вы рискуете сосредоточиться на целях, которые далеки от того, что волнует людей, использующих ваше ПО.

Вернемся к причинам, по которым существуют специальные фронтенд-команды, — специалистам, согласованности и техническим проблемам, — а теперь посмотрим, как эти проблемы можно решить.

Обмен специалистами

Хороших разработчиков бывает трудно найти. Но еще сложнее найти разработчиков с определенной специализацией. Например, в области UI, если вы предоставляете как собственные мобильные, так и веб-интерфейсы, вам понадобятся люди, имеющие опыт работы как с iOS, так и с Android, а также с современной веб-разработкой. И это помимо того факта, что вам могут понадобиться специальные дизайнеры по взаимодействию, графические дизайнеры, эксперты по доступности и т. п. Людей с нужной глубиной навыков для этих более узких областей может не хватать, и у них всегда может быть больше работы, чем времени.

[1] Как говорит Чарити Мейджорс, «вы не можете считаться фуллстек-разработчиком, если не создаете чипы».

Как упоминалось ранее, традиционный подход к организационным структурам предполагает, что вы объединяете всех людей, обладающих одинаковым набором навыков, в одну команду. Такой вариант позволяет жестко контролировать, над чем они работают. Но необходимо помнить, что это приводит к разрозненности организаций.

Включение людей со специализированными навыками в отдельную команду также лишает других разработчиков возможности приобрести эти востребованные навыки. Например, вам не нужно, чтобы каждый сотрудник учился на опытного iOS-разработчика. Но некоторым из них все равно может быть полезно овладеть определенными навыками в этой области, чтобы помогать с простыми вещами, позволяя специалистам решать действительно сложные задачи. Обмену опытом также может помочь создание сообществ практиков, например организация сообщества UI, охватывающего все ваши команды и позволяющего людям делиться идеями и проблемами со своими коллегами.

Я помню времена, когда все изменения в БД должны были выполняться центральным пулом администраторов БД (АБД). В результате разработчики плохо понимали, как работают базы данных, и чаще создавали ПО, неэффективно использующее БД. Кроме того, большая часть работы, поручаемая опытным АБД, состояла из выполнения тривиальных задач. Поскольку основная часть работы с БД была передана командам доставки, разработчики в целом стали лучше разбираться в базах данных и могли выполнять тривиальную работу самостоятельно, освобождая ценных АБД, чтобы те могли сосредоточиться на более сложных проблемах. Это плодотворно сказалось на использовании их навыков и опыта. Аналогичный сдвиг произошел в сфере операционной деятельности и тестирования, причем большая часть этой работы была передана командам.

Таким образом, обратный перевод работников из специализированных команд не повлияет на их способность выполнять свою работу. Скорее всего, это лишь увеличит пропускную способность, которую они должны использовать для решения сложных проблем, действительно требующих их внимания.

Хитрость в том, чтобы найти более эффективный способ привлечения ваших специалистов к работе. В идеале они должны быть интегрированы в команды. Иногда, однако, может оказаться недостаточно работы, чтобы оправдать их постоянное присутствие в конкретной команде, и в этом случае они вполне могут разделить свое рабочее время между несколькими командами. Другая модель заключается в том, чтобы создать специальную команду с этими навыками, чья явная задача — *помогать* другим командам. В «Топологиях команд» Скелтон и Пейс описывают эти команды как *команды поддержки*. Их задача — помогать другим командам, сосредоточенным на предоставлении новых функций, выполнять свою работу. Представьте себе эти команды как внутренних консультантов:

они могут прийти и провести запланированное время с потоковой командой, помогая ей стать более самодостаточной в определенной области или же выделяя некоторое время для выполнения особенно сложной части работы.

Таким образом, независимо от того, являются ли ваши специалисты штатными сотрудниками конкретной команды или работают над тем, чтобы другие могли выполнять ту же работу, вы можете устранить организационную разрозненность и одновременно помочь своим коллегам повысить квалификацию.

Обеспечение согласованности

Вторая проблема, часто приводимая в качестве причины создания специализированных фронтенд-команд, — это проблема согласованности. При наличии единой команды, ответственной за UI, у вас будут гарантии, что пользовательский интерфейс будет выглядеть и восприниматься единообразно. Это может касаться как простых вещей, таких как использование одних и тех же цветов и шрифтов, так и решения одних и тех же проблем интерфейса одинаковым способом — с использованием согласованного дизайна и языка, помогающего пользователям взаимодействовать с системой. Такая согласованность не только помогает довести продукт до определенной степени совершенства, но и гарантирует, что пользователям будет легче работать с новыми функциями после их внедрения.

Однако есть способы помочь обеспечить определенную согласованность действий между командами. Если вы используете модель с командой поддержки, когда специалисты проводят время с несколькими командами, они могут помочь обеспечить согласованность работы, выполняемой каждой командой. Также может помочь создание общих ресурсов, таких как активное руководство по стилю CSS или общие компоненты UI.

Например, команда Origami из Financial Times создает веб-компоненты в сотрудничестве с командой дизайнеров, которая инкапсулирует фирменный стиль, обеспечивая единообразный внешний вид для всех потоковых команд. Данный тип команды поддержки предоставляет две формы помощи: делится своим опытом в поставке уже созданных компонентов и помогает гарантировать, что UI обеспечивает соответствующий пользовательский опыт.

Однако стоит отметить, что драйвер согласованности не следует считать универсально правильным. Некоторые организации принимают, по-видимому, сознательные решения не требовать согласованности в своих UI, поскольку считают, что предпочтительнее обеспечить большую автономию команд. Amazon — одна из таких организаций. У более ранних версий их основного торгового сайта была большая степень несогласованности, поскольку виджеты применяли совершенно разные стили управления.

Это проявляется в еще большей степени, если посмотреть на веб-панель управления Amazon Web Services (AWS). У разных продуктов в AWS совершенно разные модели взаимодействия, что делает UI довольно запутанным. Однако это, по-видимому, стало логическим продолжением стремления Amazon сократить внутреннюю координацию между командами.

Повышенная автономия групп разработки определенных продуктов в AWS, вероятно, проявляется и в других аспектах, а не только в плане часто разрозненного пользовательского опыта. Бывает, что существует несколько различных способов решения одной и той же задачи (например, путем запуска рабочей нагрузки контейнера), при этом разные группы разработчиков внутри AWS часто накладываются друг на друга с помощью похожих, но несовместимых решений. Можно критиковать конечный результат, но AWS показала, что, имея эти высокоавтономные команды, ориентированные на продукт, она создала компанию, которая явно лидирует на рынке. Скорость доставки превосходит согласованность пользовательского опыта, по крайней мере в том, что касается AWS.

Решение технических проблем

Мы пережили несколько интересных эволюций, когда дело дошло до разработки пользовательских интерфейсов: от текстовых UI на основе терминалов с зеленым экраном до многофункциональных настольных приложений, сетевых, а теперь и мобильных. Во многих отношениях мы прошли полный цикл разработки, а затем еще немного, ведь наши клиентские приложения теперь создаются с такой сложностью и изощренностью, что они абсолютно не уступают по сложности богатым приложениям для настольных ПК, которые были основой разработки пользовательских интерфейсов в течение первого десятилетия XXI века.

В какой-то степени все новое — хорошо забытое старое. Мы часто все еще работаем с теми же элементами UI, что и 20 лет назад, — с кнопками, флажками, формами, полями со списком и т. п. В это пространство было добавлено еще несколько компонентов, но их гораздо меньше, чем вы думаете. Что изменилось в первую очередь, так это технология, которую мы используем для создания GUI.

Некоторые новые технологии в этой области, в частности одностраничные приложения, вызывают у нас проблемы, когда дело доходит до декомпозиции пользовательского интерфейса. Кроме того, более широкое разнообразие устройств, на которых мы ожидаем предоставления одного и того же пользовательского интерфейса, вызывает другие требующие решения проблемы.

По сути, пользователи хотят взаимодействовать с ПО как можно более беспрепятственно. Будь то через браузер на рабочем столе или через нативное или веб-мобильное приложение, результат один и тот же — пользователи взаимодействуют с нашим ПО через унифицированный UI. Их не должно волновать,

построен ли UI модульным или монолитным способом. Поэтому необходимо искать способы разделить функциональность пользовательского интерфейса и собрать все это вместе, одновременно решая проблемы, вызванные одностраничными приложениями, мобильными устройствами и многим другим. Эти вопросы будут занимать наше внимание до конца текущей главы.

Шаблон: монолитный интерфейс

Шаблон *монолитного интерфейса* описывает архитектуру, в которой все состояние и поведение пользовательского интерфейса определяются в самом пользовательском интерфейсе с вызовами резервных микросервисов для получения требуемых данных или выполнения операций. На рис. 14.3 показан такой пример. На экране требуется отображать информацию об альбоме и его списке композиций, поэтому UI делает запрос на извлечение этих данных из микросервиса `Альбом`. Мы также показываем информацию о самых последних специальных предложениях, запрашивая данные у микросервиса `Акции`. В этом примере наши микросервисы возвращают данные в формате JSON, которые UI использует для обновления отображаемой информации.

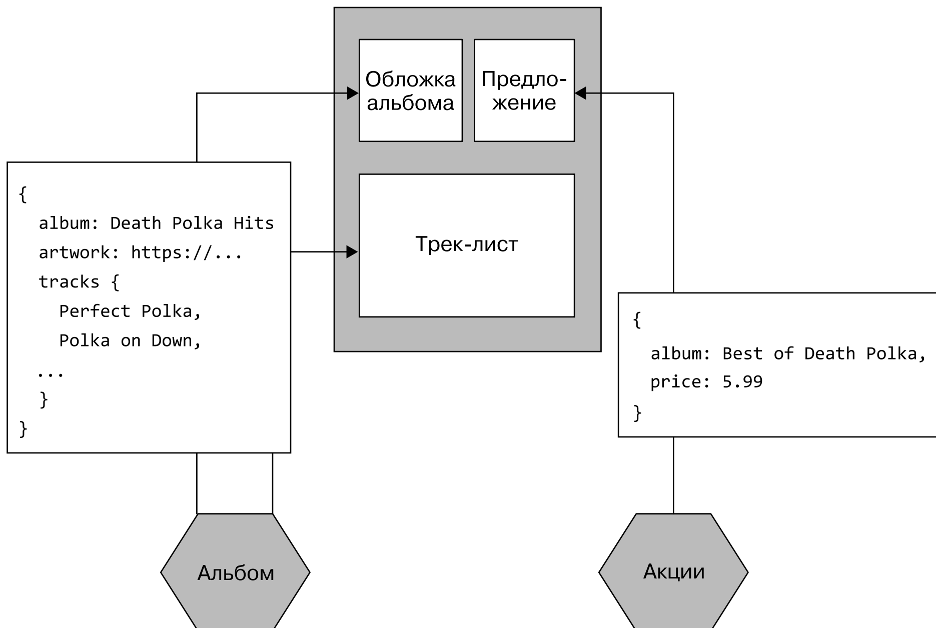


Рис. 14.3. Экран сведений об альбоме извлекает информацию из нижестоящих микросервисов для отображения пользовательского интерфейса

Эта модель стала наиболее распространенной для людей, создающих монолитные одностраничные приложения, часто со специальной фронтенд-командой. Требования наших микросервисов довольно просты: им просто нужно обмениваться информацией в форме, которая может быть легко интерпретирована пользовательским интерфейсом. В случае веб-интерфейса это означает, что нашим микросервисам, скорее всего, потребуется предоставлять данные в текстовом формате, наиболее вероятным выбором будет формат JSON. Затем пользовательскому интерфейсу понадобится создать различные компоненты, составляющие интерфейс, обрабатывающие синхронизацию состояния и т. п. с помощью серверной части. Применение двоичного протокола для обмена данными между сервисами было бы более сложным для веб-клиентов, но могло бы подойти для нативных мобильных устройств или «тучных» настольных приложений.

Когда использовать

У данного подхода есть некоторые недостатки. Во-первых, по своей природе монолитной сущности он может стать драйвером (или быть вызван) для выделенной фронтенд-команды. Наличие нескольких команд, разделяющих ответственность за этот монолитный интерфейс, может быть сложной задачей из-за множества источников разногласий. Во-вторых, у нас мало возможностей адаптировать ответы для различных типов устройств. При использовании веб-технологии можно изменить расположение экрана в соответствии с различными ограничениями устройства, но это не обязательно распространяется на изменение вызовов, выполняемых к поддерживающим микросервисам. Мой мобильный клиент может отображать только десять полей заказа, но, если микросервис возвращает все сто полей, мы в итоге извлекаем ненужные данные. Одно из решений данного подхода заключается в указании пользовательскому интерфейсу, какие поля нужно возвращать, когда он делает запрос, но это предполагает, что каждый поддерживающий микросервис способен реализовать эту форму взаимодействия. В разделе «GraphQL» в конце главы мы рассмотрим, как использование шаблона «Бэкенд для фронтенда» и GraphQL помогут в этом случае.

На самом деле такой шаблон работает лучше всего, когда *требуется*, чтобы вся реализация и поведение UI находились в одном развертываемом модуле. Для одной команды, разрабатывающей как интерфейс, так и все вспомогательные микросервисы, это может быть вполне приемлемо. Лично я думаю, что, если над вашим ПО работает более одной команды, вам следует бороться с этим желанием, поскольку оно может привести к тому, что вы скатитесь к многоуровневой архитектуре с соответствующими организационными разрозненностями. Однако, если не получается избежать многоуровневой архитектуры и соответствующей организационной структуры, вероятно, в конечном счете вы будете использовать именно эту модель.

Шаблон: микрофронтенды

Микрофронтенды — это организационная модель, при которой различные части интерфейса могут обрабатываться и развертываться независимо. Цитируя повсеместно рекомендуемую статью Кэма Джексона на эту тему[1], можно определить *микроифронтенды* как «архитектурный стиль, в котором независимо поставляемые фронтенд-приложения объединяются в единое целое».

Это становится важным шаблоном для потоковых команд, которые хотят самостоятельно предоставлять как бэкенд-микросервисы, так и поддерживающий UI. Там, где микросервисы обеспечивают независимую развертываемость для функциональности бэкенда, микрофронтенды — для фронтенда.

Концепция микрофронтендов приобрела популярность из-за проблем, создаваемых монолитными веб-интерфейсами с большим количеством кода JavaScript, типичными для одностраничных приложений. Применяя микрофронтенды, разные команды могут работать над различными частями интерфейса и вносить в них изменения. Возвращаясь к рис. 14.2, команды, отвечающие за сервис склада, за движение закупок и за профиль покупателя, могут преобразовывать функциональные возможности интерфейса, связанные с их потоком работы, независимо от других команд.

Реализация

Для веб-интерфейсов можно рассмотреть два ключевых метода декомпозиции, способных помочь реализации шаблона микрофронтенда. Разбиение на основе виджетов включает в себя объединение различных частей интерфейса в общий экран. С другой стороны, при декомпозиции на основе страниц интерфейс разделяется на независимые веб-страницы. Оба подхода заслуживают дальнейшего изучения, мы вскоре к этому вернемся.

Когда использовать

Шаблон микрофронтендов необходим, если вы хотите внедрить сквозные потоковые команды при попытке отойти от многоуровневой архитектуры. Я также допускаю, что это может быть полезно в ситуации, когда необходимо сохранить многоуровневую архитектуру, но функциональность внешнего интерфейса настолько велика, что требуется несколько выделенных фронтенд-команд.

В данном подходе есть одна ключевая проблема, и я не уверен, что ее можно решить. Иногда возможности, предлагаемые микросервисом, не вписываются в виджет или страницу. Конечно, можно разместить рекомендации в поле на

---

[1] *Jackson C.* Micro Frontends // martinfowler.com. 19 июня 2019 года. https://oreil.ly/U3K40.

странице нашего веб-сайта, но что, если нужно включить динамические рекомендации в другом месте? Когда выполняется поиск, я хочу, например, чтобы автоматически запускались новые рекомендации. Чем более сквозной будет форма взаимодействия, тем ниже вероятность, что эта модель подойдет, и тем выше, что мы вернемся к простым вызовам API.

АВТОНОМНЫЕ СИСТЕМЫ

Автономная система (self-contained system, SCS) — это архитектурный стиль, который возник, возможно, из-за отсутствия внимания к проблемам UI в первые годы существования микросервисов. SCS может состоять из нескольких механизмов (потенциально микросервисов), образующих при объединении одну SCS.

Согласно определению, автономная система должна соответствовать некоторым определенным критериям. Эти критерии, как мы видим, частично совпадают с теми же целями, которых мы пытаемся достичь с помощью микросервисов. Вы можете найти более подробную информацию об автономных системах на очень понятном веб-сайте SCS (https://scs-architecture.org), но вот некоторые основные моменты.

- Каждая SCS представляет собой автономное веб-приложение без общего UI.
- Каждая SCS принадлежит одной команде.
- По возможности следует использовать асинхронную связь.
- Никакой бизнес-код не может быть общим для систем SCS.

Подход SCS не прижился так, как микросервисы, и это не та концепция, с которой можно обычно столкнуться, несмотря на то что я согласен со многими изложенными в ней принципами. Мне особенно нравится тезис, что автономная система должна принадлежать одной команде. Я действительно задаюсь вопросом: отсутствие более широкого использования объясняет, почему некоторые аспекты подхода SCS кажутся чрезмерно узкими и предписывающими? Например, требование, чтобы каждая SCS была «автономным веб-приложением», подразумевает, что многие типы взаимодействия с пользователем никогда не могли считаться SCS. Означает ли это, что созданное мной собственное приложение iOS, использующее gRPC, может быть частью SCS, или нет?

Итак, находится ли подход SCS в конфликте с микросервисами? Не совсем. Я работал над многими микросервисами, которые, если рассматривать их изолированно, сами по себе соответствовали бы определению SCS. В подходе SCS есть несколько интересных идей, с которыми я согласен, и многие из них мы уже рассмотрели в этой книге. Я просто нахожу данный подход чрезмерно предписывающим, до такой степени, что кто-то заинтересованный в SCS может счесть его принятие невероятно сложным, поскольку это может потребовать массовых изменений во многих аспектах доставки ПО.

Меня беспокоит, что манифесты, подобные концепции SCS, могут направить нас по пути чрезмерного сосредоточения внимания на деятельности, а не на принципах и результатах. Можно было бы следовать каждой характеристике SCS и все равно потенциально упустить суть. Поразмыслив, я пришел к выводу, что подход SCS — это ориентированный на технологии подход к продвижению организационной концепции. Таким образом, я бы предпочел сосредоточиться на важности потоковых команд с ограниченной координацией, а технологии и архитектура пусть вытекают из этого.

Шаблон: декомпозиция на основе страниц

При декомпозиции на основе страниц UI разбивается на несколько веб-страниц. Каждый набор страниц может обслуживаться из разных микросервисов. На рис. 14.4 показан пример такой модели для MusicCorp. Запросы для страниц в папке `/albums/` направляются непосредственно в микросервис `Альбомы`, который обрабатывает обслуживание этих страниц, и у нас есть нечто подобное для папки `/artists/`. Для объединения этих страниц используется общая навигация. Микросервисы, в свою очередь, могут извлекать информацию, необходимую для создания таких страниц, например извлекать уровни запасов из микросервиса `Запасы`, чтобы показать в пользовательском интерфейсе, какие товары есть на складе.

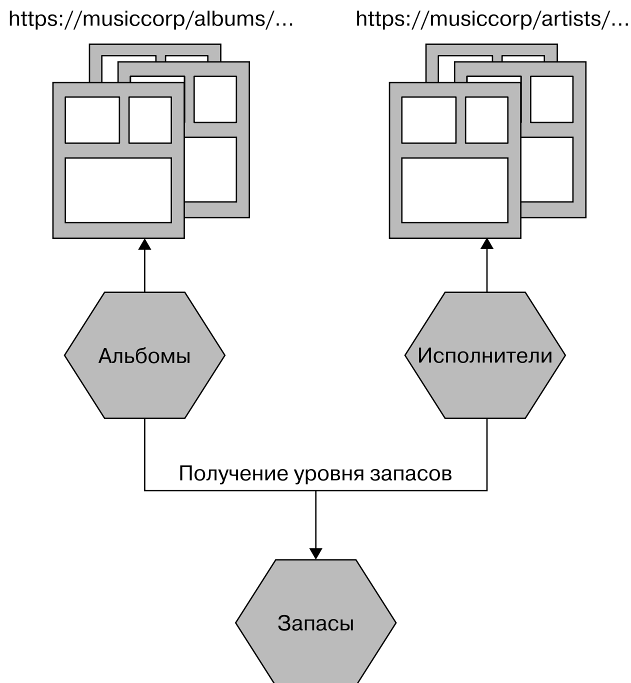


Рис. 14.4. Пользовательский интерфейс состоит из нескольких страниц, причем разные группы страниц обслуживаются из разных микросервисов

Применяя эту модель, команда, владеющая микросервисом `Альбомы`, сможет отобразить полный пользовательский интерфейс от начала до конца, что позволит ей легко понять, как изменения повлияют на пользователя.

ИНТЕРНЕТ

До появления одностраничных приложений мы пользовались веб-страницами. Взаимодействие с Интернетом основывалось на посещении URL-адресов и щелчках по ссылкам, приводящим к загрузке новых страниц в браузер. Наши браузеры были созданы для обеспечения навигации по этим страницам, используя закладки для пометки интересующих страниц и элементы управления «назад» и «вперед» для повторного просмотра ранее посещенных страниц. Возможно, вы все закатываете глаза и думаете: «Конечно, я знаю, как работает Интернет!» Однако этот стиль пользовательского интерфейса, похоже, вышел из моды. Его простота — это то, чего мне не хватает, когда я смотрю на текущие реализации веб-интерфейса пользователя. Мы многое потеряли, автоматически предполагая, что веб-интерфейс означает одностраничные приложения.

Что касается работы с различными типами клиентов, ничто не мешает странице адаптировать то, что она показывает, в зависимости от характера запрашивающего устройства. Концепции прогрессивного улучшения (или постепенной деградации) к настоящему времени должны быть хорошо поняты.

Простота декомпозиции на основе страниц с точки зрения технической реализации будет здесь действительно привлекательной. Вам не нужен какой-либо навороченный JavaScript, запущенный в браузере, и вам не нужно использовать проблемные элементы iFrame. Пользователь нажимает на ссылку, и запрашивается новая страница.

Где использовать

Полезная как для монолитного интерфейса, так и для микрофронтенда, декомпозиция на основе страниц была бы моим выбором по умолчанию для разбиения UI, если бы моим UI был веб-сайт. Веб-страница как единица декомпозиции стала настолько базовой концепцией Интернета в целом, что служит простым и очевидным методом дробления большого пользовательского веб-интерфейса.

Я думаю, проблема в том, что в стремлении использовать технологию одностраничных приложений эти UI становятся все более редкими, до такой степени, что пользовательский опыт, который, по моему мнению, лучше подходил бы для реализации веб-сайта, в конечном счете превращается в одностраничное приложение[1]. Вы, конечно, можете комбинировать декомпозицию на основе страниц с некоторыми другими рассмотренными шаблонами, например создать страницу, содержащую виджеты. Эту тему мы рассмотрим далее.

[1] Я слежу за тобой, Sydney Morning Herald!

Шаблон: декомпозиция на основе виджетов

При декомпозиции на основе виджетов экран в графическом интерфейсе содержит виджеты, которые можно изменять независимо. На рис. 14.5 показан пример интерфейса MusicCorp с двумя виджетами, обеспечивающими функциональность пользовательского интерфейса для корзины покупок и рекомендаций.

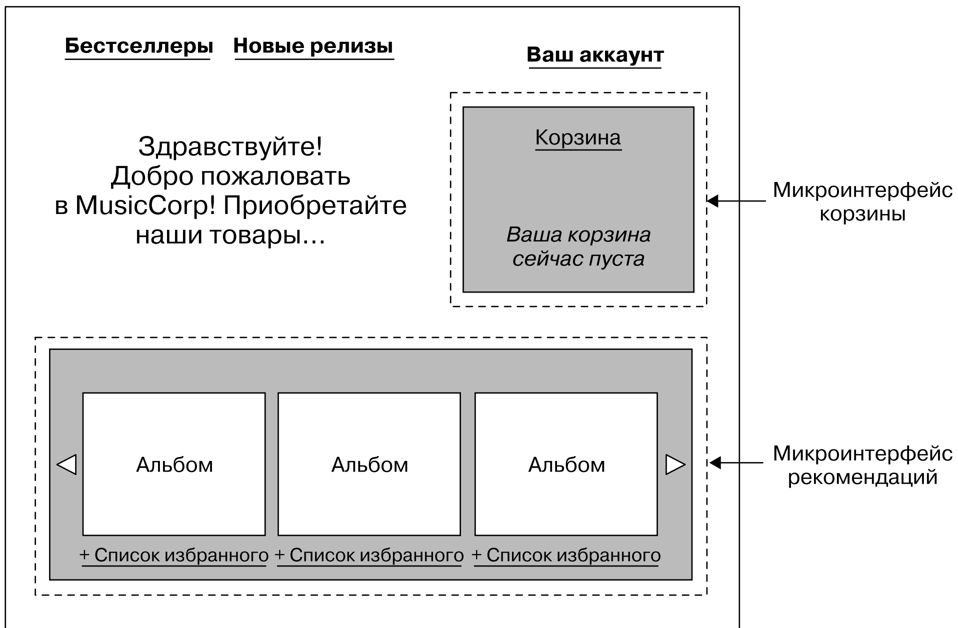


Рис. 14.5. Виджеты корзины и рекомендаций, используемые в MusicCorp

Виджет рекомендаций для MusicCorp возвращает карусель рекомендаций, которые можно циклически просматривать и фильтровать. Как показано на рис. 14.6, когда пользователь взаимодействует с виджетом рекомендаций, например переключается на следующий набор рекомендаций или добавляет элементы в свой список избранного — это может привести к вызову микросервисов серверной стороны, а в нашем случае — к вызову микросервисов `Рекомендации` и `Список избранного`. Это хорошо сочетается с командой, владеющей как этими вспомогательными микросервисами, так и самим компонентом.

В целом вам понадобится «контейнерное» приложение, определяющее такие вещи, как основная навигация для интерфейса, и то, какие виджеты необходимо включить. Если бы мы думали в терминах сквозных потоковых команд,

мы могли бы представить себе единую команду, предоставляющую виджет рекомендаций, а также отвечающую за поддержку микросервиса `Рекомендации`.

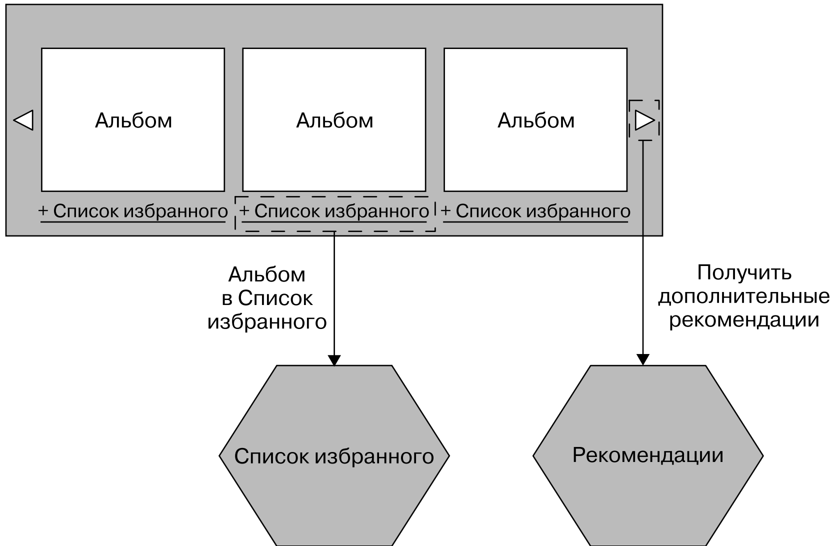


Рис. 14.6. Взаимодействие между микрофронтендом рекомендаций и вспомогательными микросервисами

Данный шаблон часто встречается в реальном мире. Например, пользовательский интерфейс Spotify активно использует его. Один виджет может содержать список воспроизведения, другой — информацию об исполнителе, а третий — информацию об исполнителях и других пользователях Spotify, на которых вы подписаны. Эти виджеты комбинируются по-разному в зависимости от ситуации.

Вам все еще нужен какой-то сборочный слой системы, чтобы соединить эти части вместе. Однако это может быть так же просто, как использование шаблонов на стороне сервера или клиента.

Реализация

Способ встраивания виджета в UI будет во многом зависеть от того, как создан ваш UI. На обыкновенном веб-сайте включение виджетов в виде фрагментов HTML с использованием шаблонов на стороне клиента или сервера может быть довольно простым, хотя и есть риск столкнуться с проблемами, если поведение виджетов более сложное. Например, если виджет рекомендаций содержит много функций JavaScript, как гарантировать, что они не будут конфликтовать

с поведением, загруженным в остальную часть веб-страницы? В идеале весь виджет стоило бы упаковать таким образом, чтобы он не нарушал другие аспекты пользовательского интерфейса.

Вопрос о том, как обеспечить автономную функциональность в UI, не нарушая другие функции, исторически был особенно проблематичным для одностраничных приложений. Отчасти это связано с тем, что концепция модульности, по-видимому, не была главной проблемой при создании поддерживающих SPA-фреймворков. Эти проблемы заслуживают более глубокого изучения.

Зависимости

Хотя iFrames были широко используемой техникой в прошлом, мы стараемся избегать применять их для объединения различных виджетов в одну веб-страницу. У iFrames есть множество проблем, связанных с размером и взаимодействием между различными частями интерфейса. Вместо этого виджеты обычно либо встраиваются в UI с помощью шаблонов на стороне сервера, либо динамически вставляются в браузер на стороне клиента. В обоих случаях проблема в том, что виджет запускается на одной странице браузера с другими частями интерфейса, а это означает, что вам нужно быть осторожным, чтобы разные виджеты не конфликтовали друг с другом.

Например, виджет рекомендаций может использовать React v16, в то время как виджет корзины — React v15. Конечно, это может быть полезно, поскольку такая ситуация помогает нам опробовать разные технологии (мы могли бы использовать разные фреймворки SPA для разных виджетов), но это также может помочь, когда дело доходит до обновления версий используемых фреймворков. Я общался с командами, у которых возникали проблемы при переходе между версиями Angular или React из-за различий в соглашениях, используемых в более новых версиях фреймворка. Обновление всего монолитного UI может быть сложной задачей, но если есть возможность делать это постепенно, обновляя элементы вашего интерфейса один за другим, вы можете разбить работу на части, а также снизить шанс появления новых проблем при обновлении.

Однако вы рискуете в конечном счете получить множественное дублирование зависимостей, что, в свою очередь, может привести к большому разрастанию размера загружаемой страницы. Например, я мог бы включить несколько разных версий фреймворка React и связанных с ним переходных зависимостей. Неудивительно, что многие веб-сайты теперь получили размер загрузки страницы, в несколько раз превышающий размер некоторых операционных систем. В качестве быстрого ненаучного исследования я проверил загрузку страницы веб-сайта CNN на момент написания книги, и она составила 7,9 Мбайт, что намного больше, чем, например, 5 Мбайт для дистрибутива Alpine Linux. 7,9 Мбайт — это на самом деле самый маленький размер загрузки некоторых встречающихся страниц.

Коммуникация между виджетами внутри страницы

Хотя наши виджеты можно создавать и развертывать независимо, все равно необходимо, чтобы они могли взаимодействовать друг с другом. Рассмотрим ситуацию с MusicCorp. Когда пользователь выбирает один из альбомов в чарте бестселлеров, нужно, чтобы другие части UI обновлялись на основе его выбора, как показано на рис. 14.7.

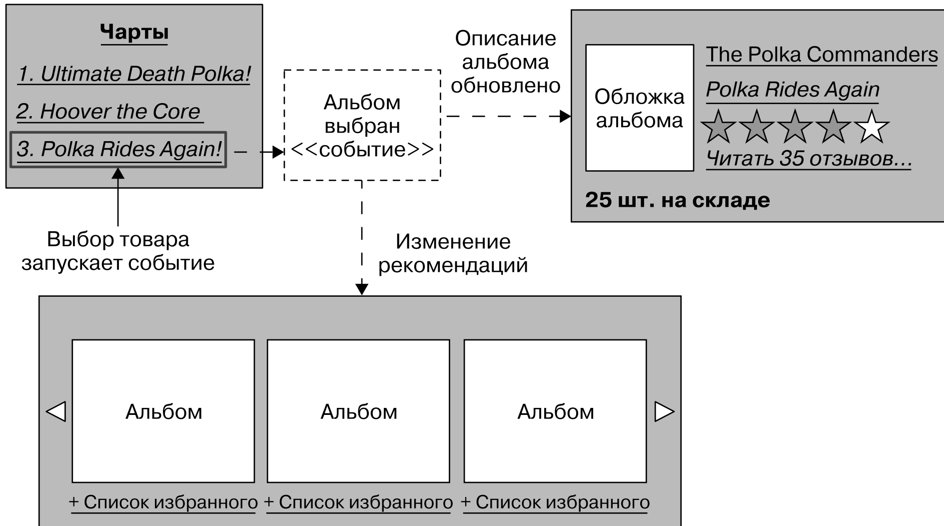


Рис. 14.7. Виджет чартов может выдавать события, которые прослушиваются другими частями пользовательского интерфейса

Способ, которым мы могли бы добиться этого, заключается в том, чтобы виджет чартов выдавал нестандартное событие. Браузеры уже поддерживают ряд событий, которые можно использовать для запуска такого поведения. Они позволяют реагировать на нажатие кнопок, прокрутку колесика мыши и т. п., и вы, вероятно, уже активно использовали такое управление событиями, если занимались разработкой интерфейсов JavaScript. Это простой шаг для создания ваших собственных нестандартных пользовательских событий.

Поэтому в нашем случае, когда элемент выбран в чарте, данный виджет вызывает пользовательское событие `Альбом выбран`. Виджеты рекомендаций и сведений об альбоме подписываются на событие и реагируют соответствующим образом, при этом рекомендации обновляются в зависимости от выбора, а сведения об альбоме загружаются. Это взаимодействие, конечно, должно быть нам уже знакомо, поскольку оно имитирует событийное взаимодействие между микросервисами, которое мы обсуждали в разделе «Шаблон: событийное взаимодействие» главы 4. Единственное реальное отличие заключается в том, что эти взаимосвязи с событиями происходят внутри браузера.

ВЕБ-КОМПОНЕНТЫ

На первый взгляд, стандарт веб-компонентов должен быть очевидным способом реализации этих виджетов. Он описывает, как можно создать компонент пользовательского интерфейса, способного изолировать его аспекты HTML, CSS и JavaScript. К сожалению, стандарту веб-компонентов, похоже, потребовалось много времени, чтобы утвердиться, и еще больше времени ушло на его надлежащую поддержку браузерами. Большая часть первоначальной работы вокруг них, похоже, застопорилась, что, очевидно, повлияло на их внедрение. Я еще не встречал организацию, которая, например, использует веб-компоненты, предоставляемые микросервисами.

Учитывая, что стандарт веб-компонентов в настоящее время поддерживается довольно хорошо, вполне возможно, что в будущем мы увидим, как они станут распространенным способом реализации изолированных виджетов или более крупных микрофронтендов, но после многих лет ожидания я сдержан в своих эмоциях.

Когда использовать

Данный шаблон позволяет нескольким потоковым командам легко вносить свой вклад в один и тот же UI. Это обеспечивает большую гибкость, чем декомпозиция на основе страниц, поскольку виджеты, поставляемые разными командами, могут одновременно сосуществовать в пользовательском интерфейсе. Это также создает возможность командам поддержки предоставлять повторно применяемые виджеты, которые могут использоваться потоковыми командами (примером этого подхода я поделился ранее, упоминая роль команды Origami из Financial Times).

Шаблон декомпозиции виджетов невероятно полезен, если вы разрабатываете богатый пользовательский веб-интерфейс, и я бы настоятельно рекомендовал прибегать к виджетам в любой ситуации, когда применяется фреймворк SPA и требуется разделить ответственность за внешний интерфейс, перейдя к подходу микрофронтендов. Методы и вспомогательные технологии, связанные с этой концепцией, заметно улучшились за последние несколько лет до такой степени, что при создании веб-интерфейса на основе SPA моим подходом по умолчанию было бы разбиение UI на микрофронтенды.

Мои основные опасения по поводу декомпозиции виджетов в контексте SPA связаны с работой, необходимой для настройки отдельного объединения компонентов, и с размером полезной нагрузки. Первая проблема, скорее всего, относится к единовременным затратам и просто связана с определением стиля упаковки для вашего существующего набора инструментов. Последний вопрос более проблематичен. Небольшое изменение зависимостей одного виджета может привести к включению в приложение целого ряда новых зависимостей, что резко увеличит размер страницы. Если вы создаете UI, в котором вес страницы вызывает беспокойство, я бы посоветовал ввести некоторые автоматические

проверки для предупреждения, если размер страницы превысит определенный допустимый порог.

С другой стороны, если виджеты более просты по своей природе и в основном представляют собой статические компоненты. Их очень легко включить в состав страницы с помощью чего-то столь же простого, как создание шаблонов на стороне клиента или сервера.

Ограничения

Прежде чем перейти к обсуждению нашего следующего шаблона, я хочу затронуть тему ограничений. Все чаще пользователи нашего ПО взаимодействуют с ним с самых разных устройств. Каждое из этих устройств накладывает различные ограничения, которые должно учитывать ПО. Например, в настольном веб-приложении предполагаются такие ограничения, как то, какой браузер используют посетители, или разрешение их экрана. Слабовидящие клиенты могут использовать наше ПО через программы чтения с экрана, а люди с ограниченной подвижностью могут с большей вероятностью использовать ввод с клавиатуры для навигации по экрану.

Поэтому, хотя наши основные сервисы (наше основное предложение) могут быть одинаковыми, нужен способ адаптировать их к различным ограничениям, существующим для каждого типа интерфейса, и ко всяческим потребностям пользователей. Если хотите, это может быть продиктовано исключительно финансовой точкой зрения — чем больше довольных клиентов, тем больше денег. Но есть и человеческие, этические соображения, которые выходят на первый план: когда мы игнорируем клиентов с особыми потребностями, мы лишаем их возможности воспользоваться нашими услугами. В некоторых контекстах невозможность навигации по пользовательскому интерфейсу из-за дизайнерских решений привела к судебным искам и штрафам. Например, в Великобритании, как и в ряде других стран, действует законодательство, обеспечивающее доступ к веб-сайтам для людей с ограниченными возможностями[1].

Мобильные устройства принесли с собой целый ряд новых ограничений. Способ взаимодействия мобильных приложений с сервером может оказать свое влияние. Речь идет не только о проблемах с пропускной способностью, где ограничения мобильных сетей играют свою роль. Различные виды взаимодействий могут привести к разрядке батареи устройства, что вызовет недовольство некоторых клиентов.

---

[1] Если вы хотите ознакомиться с законодательством, охватывающим этот вопрос в Великобритании, то это Закон о равенстве 2010 года, в частности раздел 20. В W3C также содержится хороший обзор руководящих принципов доступности (https://www.w3.org/TR/WCAG).

Характер взаимодействий также меняется в зависимости от устройства. На планшете я не могу запросто щелкнуть правой кнопкой мыши. На мобильном телефоне я, возможно, захочу спроектировать интерфейс для использования преимущественно большим пальцем одной руки. Можно было бы позволить людям взаимодействовать с сервисами с помощью SMS в местах, где пропускная способность канала связи ограничена, например на глобальном юге SMS широко используется в качестве интерфейса.

Более широкое обсуждение доступности UI выходит за рамки книги, но мы можем по крайней мере изучить конкретные проблемы, вызываемые различными типами клиентов, такими как мобильные устройства. Общим решением для удовлетворения различных потребностей клиентских устройств стало выполнение какой-либо фильтрации и агрегирования вызовов на стороне клиента. Невостребованные данные можно удалить и не отправлять на устройство, а несколько вызовов могут быть объединены в один.

Далее мы рассмотрим два полезных шаблона в этой области: центральный объединяющий шлюз и шаблон «Бэкенд для фронтенда». Мы также затронем тему использования GraphQL для помощи в адаптации ответов для различных типов интерфейсов.

Шаблон: центральный объединяющий шлюз

Объединяющий шлюз центрального назначения расположен между внешними UI и нижестоящими микросервисами и выполняет фильтрацию и объединение вызовов для всех UI. Без объединения пользовательскому интерфейсу, возможно, придется выполнять несколько вызовов для извлечения требуемой информации, часто отбрасывая извлеченные, но ненужные данные.

На рис. 14.8 показана такая ситуация. Мы хотим отобразить экран с информацией о последних заказах клиента. На экране должна воспроизводиться некоторая общая информация о клиенте, а затем перечисляться количество его заказов, отсортированных по дате, а также сводная информация, показывающая дату, статус каждого заказа и цену.

Мы делаем прямой вызов в микросервис `Покупатель`, возвращая полную информацию о клиенте, даже если нам нужно всего несколько полей. Затем мы извлекаем детали заказа из микросервиса `Заказ`. Мы могли бы несколько улучшить ситуацию, изменив микросервис `Покупатель` или `Заказ`, чтобы возвращать данные, точно соответствующие нашим критериям отбора, но это все равно потребует двух вызовов.

С помощью объединяющего шлюза можно вместо этого выполнить один вызов из UI к шлюзу. Затем шлюз выполняет все необходимые вызовы, объединяет результаты в единый ответ и отбрасывает данные, не востребованные пользовательским интерфейсом (рис. 14.9).

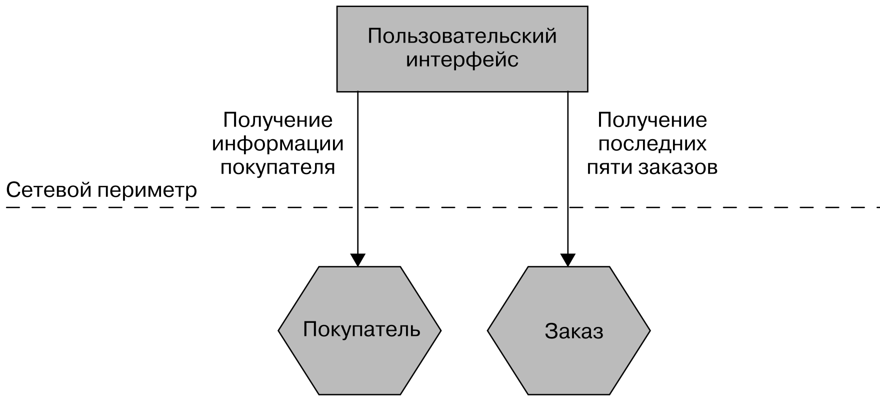


Рис. 14.8. Выполнение нескольких вызовов для получения информации для одного экрана

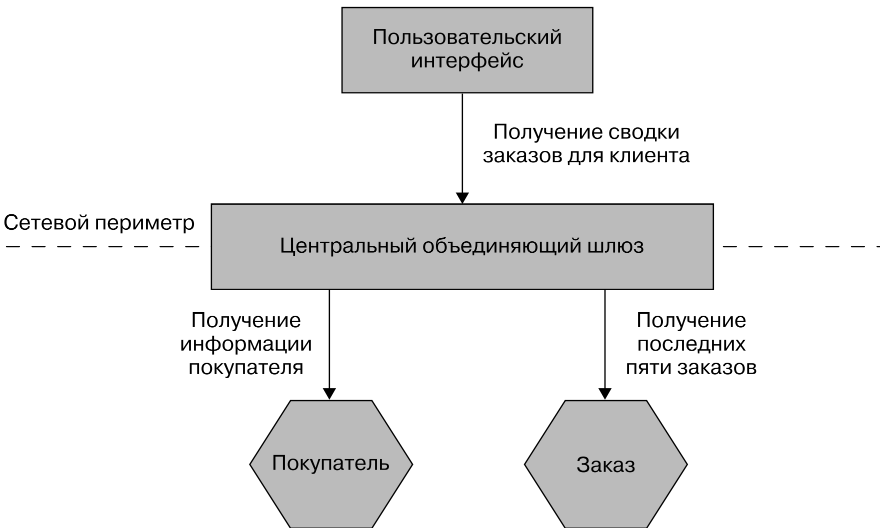


Рис. 14.9. Центральный шлюз на стороне сервера обрабатывает фильтрацию и объединение вызовов нижестоящих микросервисов

Такой шлюз также мог бы помочь в пакетных вызовах. Например, вместо того, чтобы искать десять идентификаторов заказов с помощью отдельных вызовов, я могу отправить один пакетный запрос объединяющему шлюзу, а он обработает все остальные.

По сути, наличие какого-либо объединяющего шлюза позволяет сократить количество вызовов, которые необходимо выполнить внешнему клиенту,

и уменьшить объем данных, которые требуется отправить обратно. Это может дать значительные преимущества в плане сокращения использования полосы пропускания и улучшения задержки приложения.

Владение

Поскольку все больше UI используют центральный шлюз и все большее количество микросервисов нуждаются в логике объединения вызовов и фильтрации для этих UI, шлюз становится потенциальным источником разногласий. Кто владеет шлюзом? Принадлежит ли он людям, создающим пользовательские интерфейсы, или людям, которые владеют микросервисами? Часто можно заметить, что центральный объединяющий шлюз делает так много, что в конечном счете им владеет специальная команда. Привет, изолированная многоуровневая архитектура!

По сути, характер агрегирования и фильтрации вызовов в значительной степени определяется требованиями внешних UI. Поэтому вполне логично, что шлюз должен принадлежать команде (командам), создающей UI. К сожалению, особенно в организациях, в которых есть специальная фронтенд-команда, эта команда может не обладать навыками для создания такого жизненно важного бэкенд-компонента.

Независимо от того, кто в итоге будет владельцем центрального шлюза, он может стать узким местом для доставки. Если нескольким командам необходимо внести изменения в шлюз, программирование в нем потребует координации между этими командами, что замедлит общий процесс разработки. Если им владеет одна команда, она может стать узким местом, когда дело дойдет до доставки. Вскоре мы увидим, как шаблон «Бэкенд для фронтенда» может помочь решить эти проблемы.

Различные типы пользовательских интерфейсов

Если удастся справиться с проблемами, связанными с владением, центральный объединяющий шлюз может все еще работать хорошо, пока мы не коснемся вопроса различных устройств и их потребностей. Как мы уже обсуждали, возможности мобильных устройств сильно различаются. Места на экране, доступного для отображения данных, значительно меньше. Открытие большого количества подключений к ресурсам на стороне сервера может привести к разряду аккумулятора и исчерпанию доступного трафика. Кроме того, характер взаимодействий, которые мы хотим обеспечить на мобильном устройстве, может кардинально различаться. Подумайте о типичном розничном магазине. В настольном приложении можно просматривать товары, выставленные на продажу, и заказывать их онлайн или резервировать в магазине. Однако на мобильном устройстве можно было бы разрешить сканировать штрихкоды

для сравнения цен или предоставлять контекстные предложения во время нахождения в магазине. По мере создания все большего числа мобильных приложений мы пришли к пониманию, что каждый человек использует их по-своему, и поэтому функциональность, которую мы должны предоставить, тоже будет различаться.

Таким образом, на практике наши мобильные устройства будут совершать разные и более редкие вызовы и отображать другие данные (и, вероятно, меньший их объем) данные, чем их настольные аналоги. Это означает, что нам необходимо добавить дополнительные функциональные возможности в бэкенд-API для поддержки различных типов UI. На рис. 14.10 показано, что веб-интерфейс и мобильный интерфейс MusicCorp используют один и тот же шлюз для экрана сводки о покупателе, но каждый клиент хочет получить определенный набор информации. Веб-интерфейс запрашивает дополнительную информацию о клиенте, а также содержит краткое описание товаров в каждом заказе. Это приводит нас к реализации двух различных вызовов объединения и фильтрации в бэкенд-шлюзе.

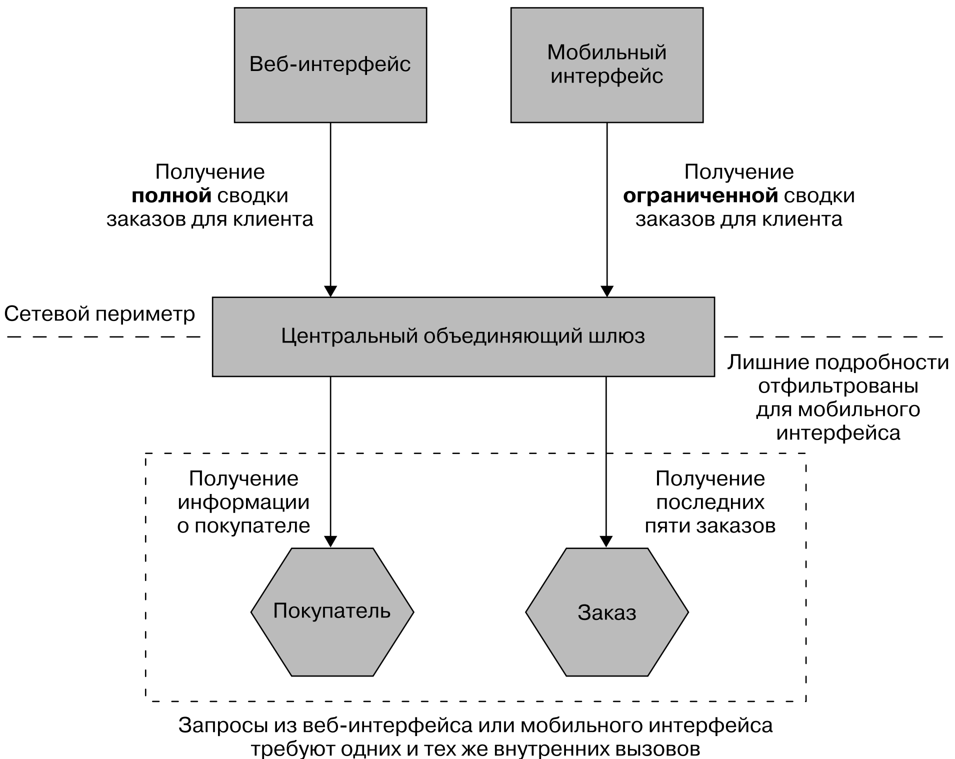


Рис. 14.10. Поддержка различных объединяющих вызовов для разных устройств

Это может привести к раздуванию шлюза, особенно если учесть различные нативные мобильные приложения, веб-сайты, ориентированные на клиента, внутренние интерфейсы администрирования и т. п. Конечно, у нас также есть проблема. Она заключается в том, что, хотя эти разные UI могут принадлежать разным командам, шлюз представляет собой единое целое — остаются старые проблемы, связанные с необходимостью работать над одним и тем же развернутым модулем нескольким командам. Единый агрегирующий бэкенд рискует стать узким местом, поскольку будет множество попыток внести изменения в один и тот же развертываемый артефакт.

Многочисленные проблемы

Существует целый ряд проблем, которые, возможно, потребуется решить на стороне сервера, когда речь заходит об обработке вызовов API. Помимо агрегирования и фильтрации вызовов, стоит подумать о более общих проблемах, таких как управление API-ключами, аутентификация пользователей или маршрутизация вызовов. Зачастую эти общие проблемы могут быть решены с помощью API-шлюзов, которые доступны во многих размерах и по разным ценам (некоторые из которых невероятно высоки!). В зависимости от сложности, которая вам требуется, может иметь смысл приобрести продукт (или лицензировать услугу), который будет решать некоторые из этих задач вместо вас. Вы действительно хотите самостоятельно управлять выдачей API-ключей, отслеживанием, ограничением скорости и т. д.? Обязательно рассмотрите готовые продукты в этой области, но будьте осторожны, пытаясь также использовать их для объединения и фильтрации вызовов, даже если они утверждают, что могут это сделать.

При настройке продукта, созданного кем-то другим, вам часто приходится работать в чужой среде. Инструментарий будет ограничен, потому что вы, возможно, будете использовать непривычный язык программирования и не известные вам методы разработки. Вместо того чтобы писать код Java, вам надо будет настраивать правила маршрутизации в каком-то странном DSL, специфичном для конкретного продукта (возможно, с использованием JSON). Такая работа может оставить неприятные впечатления, и вы будете использовать некоторые интеллектуальные возможности своей системы в стороннем продукте. Часто бывает так, что шаблон объединения вызовов на самом деле связан с некоторой функциональностью домена, которая может оправдать создание микросервиса как такового (что мы рассмотрим чуть позже, когда будем говорить о шаблонах BFF). Если это поведение заложено в конфигурации, специфичной для конкретного поставщика, перенос этой функциональности может быть более проблематичным, поскольку вам, скорее всего, придется изобретать ее заново.

Ситуация может стать еще хуже, если объединяющий шлюз станет настолько сложным, что для его владения и управления потребуется специальная команда.

В худшем случае более горизонтальное распределение обязанностей между командами может привести к ситуации, когда для развертывания какой-либо новой функциональности придется привлекать фронтенд-команду, команду объединения и команду (команды), владеющую микросервисом, чтобы каждая из них внесла свои изменения. Внезапно процесс разработки начинает идти гораздо медленнее.

Так что, если вы хотите применить выделенный API-шлюз — пожалуйста, но настоятельно рекомендую разместить вашу логику фильтрации и объединения в другом месте.

Когда использовать

Для отдельной команды, которая разрабатывает UI и бэкенд-микросервисы, было бы неплохо создать единый центральный объединяющий шлюз. Тем не менее похоже, что эта команда проделывает *огромную* работу — в таких ситуациях я обычно вижу большую степень согласованности между пользовательскими интерфейсами, что часто устраняет необходимость в этих точках объединения.

Если вы все-таки решите использовать единый центральный объединяющий шлюз, будьте осторожны и ограничьте функциональность, которую вы в него вкладываете. Я бы остерегался внедрять эту функциональность в более общий продукт, например API-шлюз, по изложенным ранее причинам.

Тем не менее концепция фильтрации и объединения вызовов на стороне бэкенда может быть действительно важной с точки зрения оптимизации взаимодействия пользователя с UI. Проблема в том, что при организации доставки с несколькими командами использование центрального шлюза может привести к необходимости выполнения большого объема взаимодействий между этими командами.

Итак, если все же требуется выполнить объединение и фильтрацию на стороне бэкенда, но необходимо устранить проблемы, связанные с моделью владения центральным шлюзом, что мы можем сделать? Вот тут пора обратиться к шаблону «Бэкенд для фронтенда» (Backend for Frontend, BFF).

Шаблон: бэкенд для фронтенда (BFF)

Основное различие между BFF и центральным объединяющим шлюзом заключается в том, что шаблон BFF одноцелевой по своей природе — он разработан для конкретного UI. Данный шаблон оказался очень успешным в решении различных проблем, связанных с пользовательскими интерфейсами, и я видел, как он хорошо работает в ряде организаций, включая Sound-Cloud[1] и REA. Как по-

[1] В статье BFF @ SoundCloud (https://oreil.ly/DdnzN) за авторством Lukasz Plotnicki отличный обзор того, как SoundCloud использует шаблон BFF.

казано на рис. 14.11, где мы возвращаемся к MusicCorp, веб- и мобильные торговые интерфейсы теперь получают свои собственные объединяющие бэкенды.

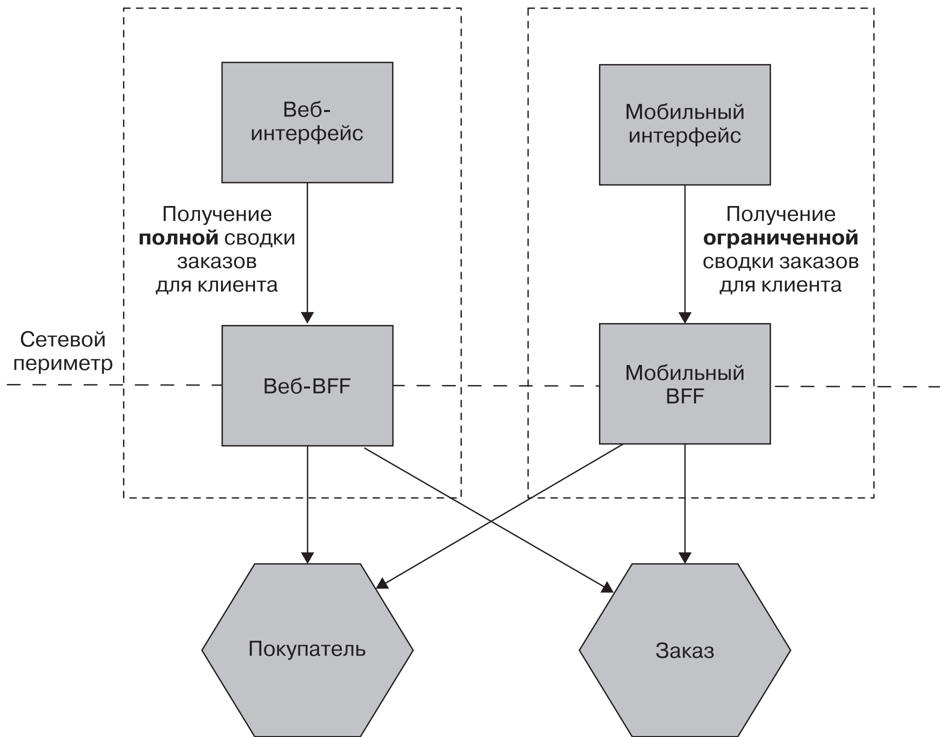


Рис. 14.11. У каждого пользовательского интерфейса есть свой собственный BFF

Из-за своей специфики BFF обходит стороной некоторые проблемы, связанные с центральным шлюзом. Поскольку мы не пытаемся быть всем для всех людей, BFF не становится узким местом для разработки, когда несколько команд пытаются разделить ответственность. Мы также меньше беспокоимся о связанности с пользовательским интерфейсом, поскольку ее уровень гораздо более приемлем. Я часто описываю использование BFF с UI так, как будто пользовательский интерфейс на самом деле разделен на две части. Одна часть находится на клиентском устройстве (веб-интерфейс или нативное мобильное приложение), а вторая расположена на стороне сервера.

BFF тесно связан с конкретным пользовательским опытом и обычно поддерживается той же командой, что и пользовательский интерфейс, что упрощает определение и адаптацию API в соответствии с требованиями UI, а также упрощает процесс подготовки выпуска как клиентских, так и серверных компонентов.

Сколько должно быть BFF

Если говорить о предоставлении одинакового (или схожего) пользовательского опыта на различных платформах, я видел два разных подхода. Модель, которую я предпочитаю (и которую вижу чаще всего), заключается в том, чтобы строго соблюдать наличие одного BFF для каждого отдельного типа клиентов — это используемая в REA модель, как показано на рис. 14.12. У каждого из приложений для Android и iOS, несмотря на схожую функциональность, был свой собственный BFF.

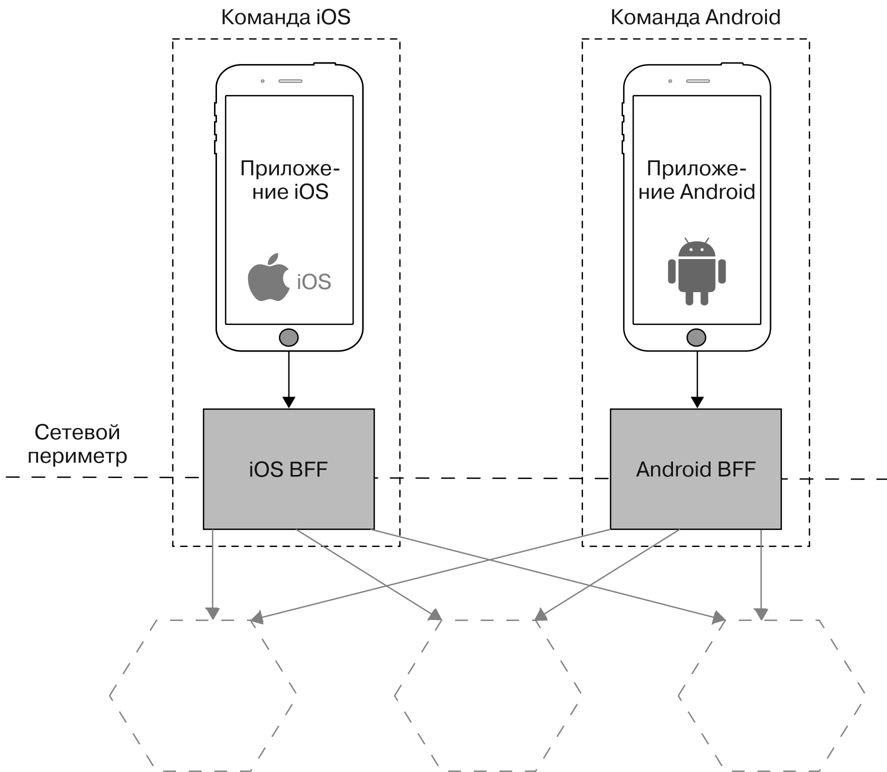


Рис. 14.12. У приложений REA для iOS и Android разные BFF

Вариантом является поиск возможностей использования одного и того же BFF для нескольких типов клиентов, хотя и для одного и того же типа UI. Приложение SoundCloud позволяет людям прослушивать контент на своих устройствах Android или iOS. SoundCloud использует единый BFF для обеих платформ, как показано на рис. 14.13.

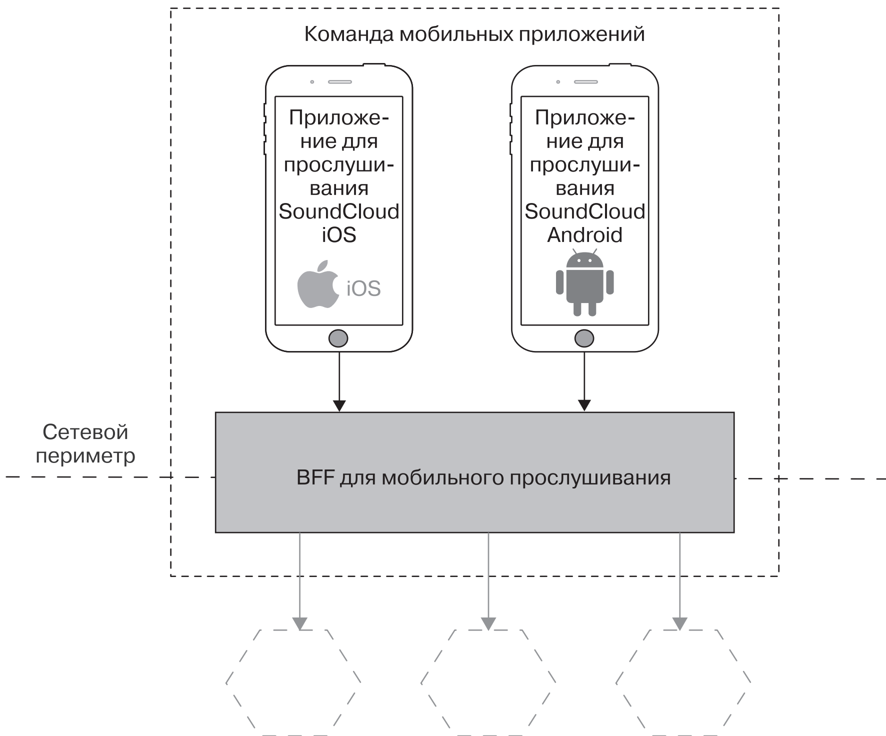


Рис. 14.13. В SoundCloud используют общий BFF для приложений iOS и Android

Во второй модели меня сильно беспокоит то, что чем больше типов клиентов используют один BFF, тем выше соблазн раздуть BFF за счет обработки нескольких задач. Однако здесь важно понимать, что, даже когда клиенты совместно используют BFF, он предназначен для одного класса UI, поэтому, в то время как нативные приложения SoundCloud для прослушивания контента для iOS и Android используют единый BFF, другие нативные приложения будут использовать иные BFF. Я более спокойно отношусь к использованию этой модели, если одна и та же команда владеет приложениями для Android и iOS, а также владеет BFF. Если эти приложения поддерживаются разными командами, я склонен рекомендовать более строгую модель. Таким образом, можно рассматривать свою организационную структуру как один из основных факторов, определяющих, какая модель представляет наибольший смысл (закон Конвея снова побеждает).

Стюарт Глидоу из REA предложил принцип «Один опыт — один BFF». Так что, если опыт взаимодействия с iOS и Android очень похож, тогда легче

оправдать наличие одного BFF[1]. Однако, если они сильно расходятся, то лучше поддерживать отдельные BFF. В случае с REA, хотя два опыта взаимодействия частично совпадали, они принадлежали отдельным командам, которые по-разному внедряли схожие функции. Иногда одна и та же функция может быть развернута по-разному на разных мобильных устройствах — то, что составляет нативный интерфейс для приложения Android, возможно, потребуется переработать, чтобы оно выглядело нативным на iOS.

Еще один урок из истории с REA (который мы уже много раз обсуждали) заключается в том, что ПО часто работает лучше всего, когда оно ориентировано на границы команды, и BFF не исключение. Наличие у SoundCloud единой мобильной команды делает наличие одного BFF разумным на первый взгляд, так же как и наличие у REA двух разных BFF для двух отдельных команд. Отмечу, что инженеры SoundCloud, с которыми я общался, предположили, что наличие одного BFF для приложений прослушивания контента на Android и iOS — это то, что им стоило бы пересмотреть. У них была единая мобильная команда, но фактически она представляла собой смесь специалистов по Android и iOS, и оказалось, что они в основном работают над одним или другим приложением. Такая схема работы подразумевает, что на самом деле они были двумя отдельными командами.

Часто движущей силой к меньшему количеству BFF становится желание повторно использовать функциональность на стороне сервера, чтобы избежать излишнего дублирования. Но есть и другие способы справиться с этим, о которых мы поговорим далее.

Повторное использование и BFF

Одна из проблем, связанных с наличием одного BFF для каждого UI, заключается в том, что в итоге вы можете столкнуться с большим объемом дублирования между BFF. Например, они могут в конечном счете выполнять одни и те же типы объединения, иметь идентичный или похожий код для взаимодействия с нижестоящими сервисами и т. д. Если вы захотите извлечь общую функциональность, то ее придется еще поискать. Такое дублирование может происходить в самих BFF, но оно также может быть заложено в различных клиентах. Из-за того что эти клиенты используют очень разные стеки технологий, затрудняется идентификация факта такого дублирования. Поскольку у организаций, как правило, общий технологический стек для серверных компонентов, наличие нескольких BFF с дублированием легче обнаружить и учесть.

[1] Стюарт, в свою очередь, поблагодарил Фила Кальсадо и Мустафу Сезгина за эту рекомендацию.

Некоторые люди реагируют на это, попыткой объединить BFF обратно, и в итоге они получают универсальный объединяющий шлюз. Мое беспокойство по поводу регресса к единому шлюзу заключается в том, что в конечном счете можно потерять больше, чем получить, тем более что существуют другие способы решения проблемы дублирования.

Как я уже говорил, я довольно спокойно отношусь к дублированию кода в микросервисах. Это означает, что если в границах одного микросервиса я обычно делаю все возможное для рефакторинга дублирования в подходящие абстракции, то с дублированием микросервисов я так не поступаю. Меня больше беспокоит, что извлечение общего кода может привести к сильной связанности между сервисами (тема, которую мы исследовали в разделе «DRY и опасности повторного использования кода в мире микросервисов» главы 5). Тем не менее, безусловно, есть случаи, когда это оправданно.

Когда приходит время извлечь общий код, чтобы обеспечить повторное использование кода BFF, есть два очевидных варианта. Первый часто дешевле, но более рискован и заключается в извлечении какой-либо общей библиотеки. Причина, по которой это может быть проблематичным решением, в том, что общие библиотеки окажутся основным источником связанности, особенно когда они применяются для создания клиентских библиотек вызова нижестоящих сервисов. Тем не менее бывают ситуации, в которых это кажется правильным, особенно когда абстрагируемый код представляется чисто внутренней проблемой сервиса.

Другой вариант — извлечь общие функции в новый микросервис. Это может хорошо работать, если извлекаемая функциональность относится к области бизнеса. Разновидностью данного подхода может быть передача обязанностей по агрегированию микросервисам ниже по потоку. Рассмотрим ситуацию, в которой требуется отобразить список товаров в клиентском списке избранного, а также информацию о том, есть ли эти товары на складе, и текущую цену, как показано в табл. 14.1.

Таблица 14.1. Отображение списка избранного для клиента MusicCorp

The Brakes, Give Blood	На складе!	$5,99
Blue Juice, Retrospectable	Нет в наличии	$7,50
Hot Chip, Why Make Sense?	Покупайте скорее! (Осталось 2)	$9,99

Микросервис `Покупатель` хранит информацию о списке избранного и идентификаторе каждого элемента, `Каталог` — название и цену каждого товара, а уровни запасов хранятся в микросервисе `Запасы`. Чтобы отобразить один и тот же элемент управления как в приложениях iOS, так и в Android, каждому

BFF потребуется выполнить те же три вызова поддерживающих микросервисов, как показано на рис. 14.14.

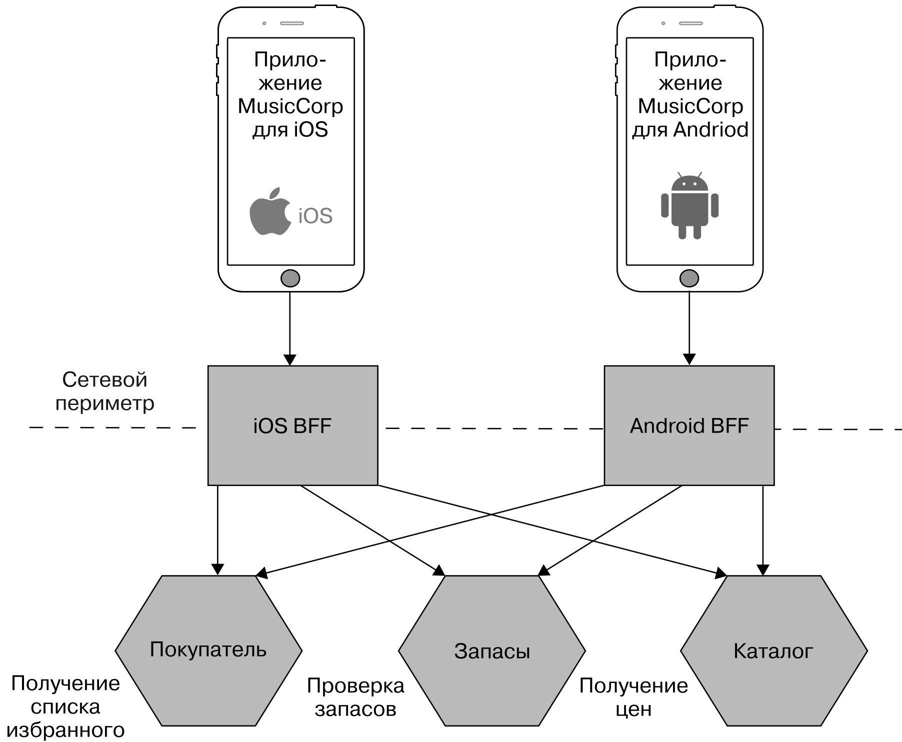


Рис. 14.14. Оба BFF выполняют одни и те же операции для отображения списка избранного

Одним из способов уменьшить дублирование функциональности здесь может служить извлечение этого общего поведения в новый микросервис. На рис. 14.15 показан новый выделенный микросервис `Список избранного`, который могут использовать Android и iOS-приложения.

Должен сказать, что один и тот же код, используемый в двух местах, не вызвал у меня моментального желания извлечь сервис таким образом. Но определенно стоило бы об этом задуматься, если бы транзакционные издержки создания нового сервиса были достаточно низкими или если бы код использовался более чем в паре мест. В данной ситуации, например, если бы мы также показывали списки избранного в нашем веб-интерфейсе, выделенный микросервис стал бы выглядеть еще более привлекательным. Я думаю, что старая пословица о создании абстракции, когда вы собираетесь реализовать что-то в третий раз, все еще кажется хорошим правилом, даже на уровне сервиса.

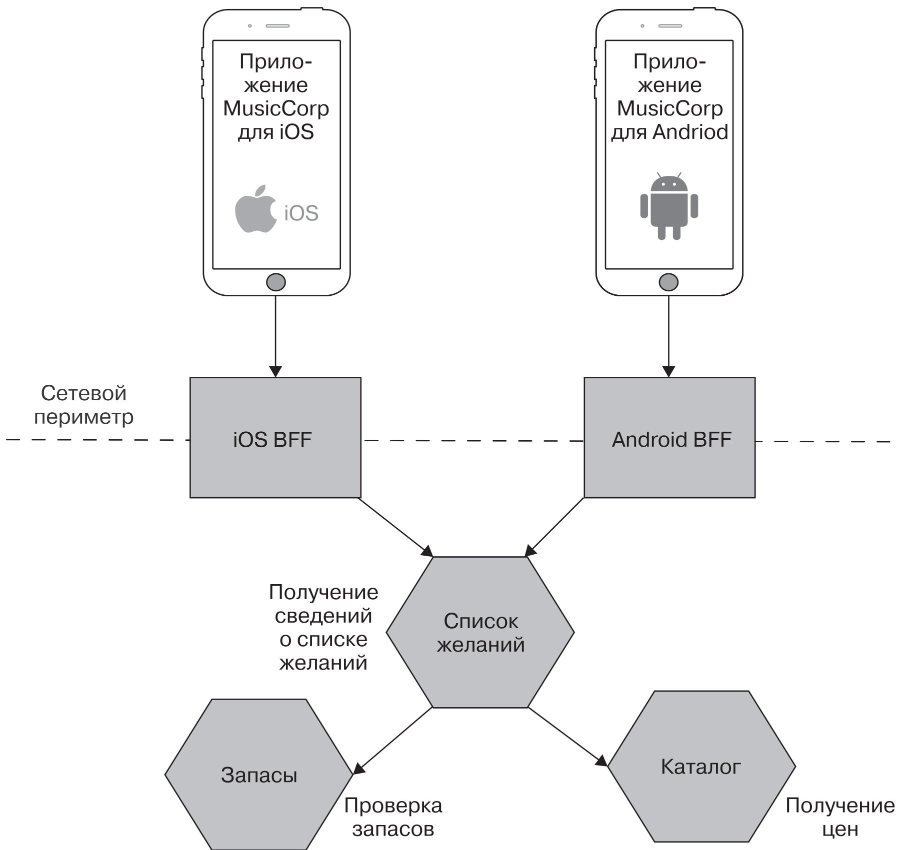


Рис. 14.15. Общая функциональность извлекается в микросервис Список избранного, что позволяет повторно использовать его в разных BFF

BFF для настольных веб-сайтов и не только

Можно считать, что BFF просто используются для решения проблем, связанных с мобильными устройствами. Веб-интерфейс для настольных компьютеров обычно предоставляется на более мощных устройствах с улучшенной связью, где затраты на выполнение нескольких вызовов сервисов будут приемлемы. Это позволит вашему веб-приложению совершать несколько вызовов непосредственно к нижестоящему сервису без необходимости в BFF.

Однако я видел ситуации, в которых BFF для веб-приложений тоже может быть полезным. Когда вы создаете большую часть веб-интерфейса на стороне сервера (например, используя шаблонизацию на стороне сервера), BFF станет

очевидным местом, где это можно сделать. Данный подход также несколько упрощает кэширование, поскольку вы можете поместить обратный прокси перед BFF, что позволит кэшировать результаты объединенных вызовов.

Я знаю, что по крайней мере одна организация использовала BFF для сторонних клиентов, которым нужно было совершать вызовы. Возвращаясь к извечному примеру с MusicCorp, можно было бы предоставить BFF, чтобы позволить третьей стороне извлекать информацию о выплате роялти или разрешить стриминг на ряд устройств с приставкой, как показано на рис. 14.16. На самом деле это уже не совсем BFF, поскольку на сторонних потребителях не отображается пользовательский интерфейс, но это пример того, как тот же шаблон используется в другом контексте, поэтому я решил, что им стоит поделиться.

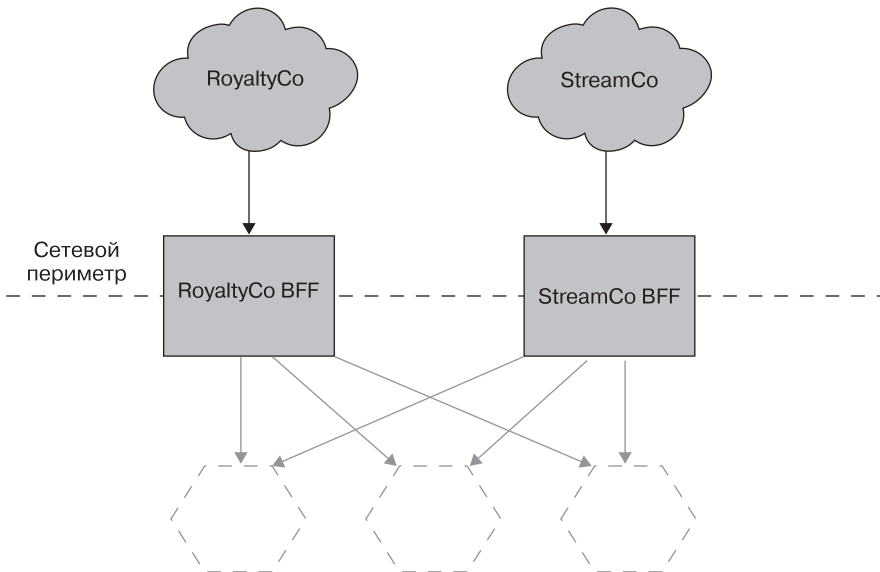


Рис. 14.16. Использование BFF для управления внешними API

Этот подход может быть особенно эффективным, поскольку третьи стороны часто не имеют возможности (или желания) использовать или изменять выполняемые вызовы API. При использовании централизованного бэкенда API вам, возможно, придется поддерживать старые версии API только для того, чтобы удовлетворить небольшую группу пользователей, которые не могут внести изменения. С BFF эта проблема существенно уменьшается. Это также ограничивает влияние критических изменений. Вы можете изменить API для Facebook таким образом, что будет нарушена совместимость с другими сторонами, но, поскольку они используют другой BFF, это изменение на них не повлияет.

Когда использовать

Для приложения, предоставляющего только веб-интерфейс, BFF будет иметь смысл, если на стороне сервера требуется значительный объем объединения. В противном случае я думаю, что некоторые из уже рассмотренных методов составления UI могут работать так же хорошо, не требуя дополнительного серверного компонента.

Однако, когда вам нужно предоставить определенную функциональность для мобильного UI или третьей стороны, я бы настоятельно рекомендовал использовать BFF для каждого типа клиентов с самого начала. Я мог бы пересмотреть свое решение, если стоимость развертывания дополнительных сервисов окажется высока, но разделение обязанностей, которое может обеспечить BFF, делает его довольно убедительным предложением в большинстве случаев. Значительное разделение между людьми, создающими UI, и нижестоящими сервисами еще сильнее склонило бы меня к использованию BFF по изложенным ранее причинам.

А теперь мы перейдем к вопросу о том, как реализовать BFF. Давайте посмотрим на GraphQL и ту роль, которую он может сыграть.

GraphQL

GraphQL — это язык запросов, позволяющий клиентам создавать запросы для доступа к данным или их преобразования. Как и SQL, GraphQL дает возможность динамически изменять эти запросы, позволяя клиенту точно определять, какую информацию он хочет получить обратно. Например, при стандартном вызове REST через HTTP при отправке запроса GET для ресурса `Заказ` вы получите обратно все поля для этого заказа. Но что, если в данной ситуации вам нужна была только общая сумма заказа? Конечно, вы можете просто проигнорировать другие поля или предоставить альтернативный ресурс (например, `Сводку заказов`), содержащий только требуемую информацию. С помощью GraphQL можно отправить запрос только на те поля, которые вам нужны, как показано в примере 14.1.

Пример 14.1. Пример запроса GraphQL, используемого для получения информации о заказе

```
{
    order(id: 123) {
        date
        total
        status
        delivery {
            company
            driver
            duedate
        }
    }
}
```

Здесь мы запросили заказ `123`, а также общую цену и статус заказа. Мы пошли дальше и запросили информацию о доставке этого заказа, чтобы получить информацию об имени водителя, который привезет нашу посылку, компании, в которой он работает, и ожидаемом времени прибытия посылки. При использовании обычного REST API, если информация о доставке не содержится внутри ресурса `Заказ`, нам придется выполнить дополнительный вызов для извлечения этой информации. Таким образом, GraphQL не только помогает запрашивать именно нужные поля, но и может сократить количество обходов. Запрос, подобный этому, требует, чтобы мы определили различные типы данных, к которым обращаемся, — явное определение типов является ключевой частью GraphQL.

Чтобы реализовать GraphQL, необходим *резолвер* для обработки запросов. Резолвер GraphQL находится на стороне сервера и преобразует запросы GraphQL в вызовы для фактического извлечения информации. Таким образом, при использовании микросервисной архитектуры нам понадобится резолвер, способный отображать запрос на заказ с `ID 123` в эквивалентный вызов микросервиса.

Таким образом, можно использовать GraphQL для реализации объединяющего шлюза или даже BFF. Преимущество этого языка запросов в том, что можно легко изменить желаемые объединение и фильтрацию, просто изменив запрос от клиента, — не нужно никаких изменений на стороне сервера, пока типы GraphQL поддерживают запрос, который мы хотим выполнить. Если имя водителя больше не требуется в примере запроса, мы можем просто исключить его из самого запроса, и оно больше не будет отправляться. С другой стороны, если бы мы хотели увидеть количество баллов, полученных за конкретный заказ, предполагая, что эта информация доступна в типе заказа, мы могли бы просто добавить это в запрос, и получили бы ее в ответе. Подобная реализация — значительное преимущество по сравнению с BFF, требующим, чтобы изменения в логике объединения также применялись к самому BFF.

Гибкость, которую GraphQL предоставляет клиентскому устройству для динамического изменения выполняемых запросов без изменений на стороне сервера, означает, что существует меньшая вероятность, что ваш сервер GraphQL станет общим, оспариваемым ресурсом, как в ситуации с универсальным объединяющим шлюзом. Тем не менее изменения на стороне сервера потребуются, если вам нужно предоставить новые типы или добавить поля к существующим. То есть вам все равно может понадобиться, чтобы несколько бэкенд-элементов GraphQL были выровнены по границам команд — таким образом, GraphQL становится способом реализации BFF.

Свои опасения по поводу GraphQL я подробно изложил в главе 5. И все же это удобное решение, позволяющее выполнять динамические запросы в соответствии с потребностями различных типов пользовательских интерфейсов.

Гибридный подход

Многие из вышеупомянутых вариантов не обязательно должны быть универсальными. Я встречал организацию, использующую подход декомпозиции на основе виджетов для создания веб-сайта, но применяющую BFF для работы с мобильными приложениями. Ключевой момент в том, что нам необходимо сохранить целостность базовых возможностей, предлагаемых нашим пользователям. Следует убедиться, что логика, связанная с заказом музыки или изменением данных клиента, живет внутри сервисов, обрабатывающих эти операции, и не распределяется по всей системе. Непросто избежать ловушки, когда в промежуточные слои закладывается слишком много поведения.

Резюме

Надеюсь, я показал, что декомпозиция функциональности не обязательно должна останавливаться на серверной части, и наличие выделенных фронтенд-команд не будет неизбежным. Я поделился несколькими различными способами создания UI, который может использовать вспомогательные микросервисы, обеспечивая при этом целенаправленную сквозную доставку.

В следующей главе мы перейдем от технической стороны вопроса к человеческой, когда более подробно рассмотрим взаимодействие микросервисов и организационных структур.

ГЛАВА 15

Организационные структуры

Хотя большая часть книги до этой страницы была посвящена техническим проблемам перехода к детализированной архитектуре, мы также рассмотрели взаимодействие между микросервисной архитектурой и организацией наших команд. В разделе «На пути к потоковым командам» главы 14 мы обсудили концепцию потоковых команд, несущих сквозную ответственность за предоставление функциональности, ориентированной на пользователя, и то, как микросервисы помогают воплотить такие структуры команд в реальность.

Теперь необходимо конкретизировать эти идеи и рассмотреть другие организационные моменты. Как мы увидим, если вы захотите получить максимальную отдачу от микросервисов, то игнорирование организационной структуры вашей компании — это риск!

Слабо связанные организации

На протяжении всей книги я приводил доводы в пользу слабо связанной архитектуры и утверждал, что работа с более автономными, слабо связанными, потоковыми командами, скорее всего, обеспечит наилучшие результаты. Переход на микросервисную архитектуру без изменения организационной структуры снизит полезность микросервисов — в конечном счете вы рискуете потратить немалую сумму денег за изменение архитектуры, не получив отдачи от своих инвестиций. В целом я писал о необходимости ослабления взаимодействия между командами, чтобы ускорить доставку, что, в свою очередь, позволит им принимать больше решений самостоятельно. Эти идеи мы рассмотрим более подробно в текущей главе и конкретизируем некоторые из этих организационных и поведенческих необходимых преобразований, но перед этим, я думаю, важно поделиться своим видением того, что такое слабо связанная организация.

Николь Форсгрен, Джез Хамбл и Джин Ким в книге «Ускоряйся!»[1] рассмотрели характеристики автономных, слабо связанных команд, чтобы лучше понять, какое поведение наиболее важно для достижения оптимальной производительности. По мнению авторов, ключевым моментом является возможность команды:

- вносить крупномасштабные изменения в проект своей системы без разрешения кого-либо за пределами команды;
- вносить крупномасштабные изменения в проект своей системы, не полагаясь на то, что другие команды будут вносить изменения в свои системы, и не создавая значительного фронта работ для других;
- завершать свою работу, не общаясь и не координируя свои действия с людьми, не входящими в их команду;
- развертывать и выпускать свой продукт или сервис по требованию, независимо от других сервисов, с которыми он взаимодействует;
- проводить большую часть своего тестирования по мере необходимости, не запрашивая интегрированной тестовой среды;
- выполнять развертывания в обычное рабочее время с минимальным временем простоя.

Потоковая команда, концепция, с которой мы впервые столкнулись в главе 1, соответствует этому видению слабо связанной организации. Если вы пытаетесь перейти к структуре с потоковыми командами, сверяйтесь со списком выше, чтобы убедиться, что вы движетесь в правильном направлении.

Некоторые из этих характеристик выглядят более техническими по своей природе. Например, возможность развертывания в обычное рабочее время может быть обеспечена с помощью архитектуры, поддерживающей развертывание без простоя. Но все это на самом деле требует изменения поведения. Чтобы команды могли ощутить всю ответственность за свои системы, требуется отказаться от централизованного контроля, в том числе за тем, как осуществляется принятие архитектурных решений (что мы рассмотрим в главе 16). По сути, достижение слабо связанных организационных структур требует *децентрализации* управления и контроля.

Большая часть данной главы посвящена обсуждению того, как мы заставляем все это работать, рассматривая размеры команды, типы моделей владения, роль платформы и многое другое. Существует множество изменений, которые вам стоит рассмотреть, чтобы продвинуть свою организацию в правильном направлении.

Однако прежде всего давайте подробнее поговорим о взаимосвязи между организацией и архитектурой.

---

[1] *Форсгрен Н., Хамбл Дж., Ким Дж.* Ускоряйся! Наука DevOps: Как создавать и масштабировать высокопроизводительные цифровые организации.

Закон Конвея

Наша отрасль молода и, похоже, постоянно изобретает себя заново. И все же несколько ключевых «законов» выдержали испытание временем. Закон Мура, например, гласит, что плотность транзисторов в интегральных схемах удваивается каждые два года, и он оказался сверхъестественно точным (хотя эта тенденция замедляется). Один закон, который, как я заметил, представляется почти универсально верным и гораздо более полезным в моей повседневной работе, — закон Конвея.

В статье Мелвина Конвея «Как комитеты создают новое?», опубликованной в журнале Datamation в апреле 1968 года, отмечалось:

> Любая организация, разрабатывающая систему (определяемую здесь более широко, чем просто информационные системы), неизбежно создаст проект, структура которого будет копией коммуникационной структуры организации.

Это утверждение часто цитируется в различных формах как закон Конвея. Эрик С. Рэймонд обобщил это явление в «Словаре нового хакера» (MIT Press), заявив: «Если у вас есть четыре группы, работающие над компилятором, вы получите четырехпроходной компилятор».

Закон Конвея показывает нам, что слабо связанная организация приводит к слабо связанной архитектуре (и наоборот), укрепляя идею о том, что проблематично получить преимущества слабо связанной микросервисной архитектуры без учета организационной структуры компании, создающей ПО.

Подтверждение

История гласит, что, когда Мелвин Конвей представил свою статью на эту тему в Harvard Business Review, журнал отклонил ее, заявив, что он не обосновал свою теорию. Я видел, как его теория доказывала сама себя в стольких разных ситуациях, что принял ее за истину. Но вы не должны верить мне на слово: с момента первоначального представления Конвея в этой сфере проделано много работы. Был проведен ряд исследований для изучения взаимосвязи структуры организаций и создаваемых ими систем.

Слабо и тесно связанные организации

В «Исследовании двойственности между архитектурами продукта и организации»[1] авторы рассматривают ряд различных программных систем, которые условно классифицируются как созданные либо «слабо связанными организа-

[1] *MacCormack A., Baldwin C., Rusnak J.* Exploring the Duality Between Product and Organizational Architectures: A Test of the Mirroring Hypothesis // Research Policy 41, № 8 (октябрь 2012): 1309–1324.

циями», либо «тесно связанными организациями». Что касается вторых, подумайте о фирмах, выпускающих коммерческие продукты, которые обычно объединены с четко согласованными видениями и целями, в то время как первые хорошо представлены распределенными сообществами с открытым исходным кодом.

В своем исследовании авторы сопоставили похожие пары продуктов от каждого типа организаций и обнаружили, что более слабо связанные организации на самом деле создавали более модульные, менее связанные системы, в то время как ПО более тесно связанной организации было менее модульным.

Windows Vista

Корпорация Microsoft провела эмпирическое исследование[1], в котором изучила, как ее собственная организационная структура влияет на качество конкретного программного продукта — Windows Vista. В частности, эксперты рассмотрели множество факторов, чтобы определить, насколько подвержен ошибкам тот или иной компонент в системе[2]. Проанализировав множество параметров, в том числе часто используемые показатели качества ПО, такие как сложность кода, они обнаружили, что индексы, связанные с организационными структурами (например, количество инженеров, работавших над фрагментом кода), оказались наиболее статистически значимыми показателями.

Итак, это еще один пример того, как структура организации влияет на характер создаваемой ею системы.

Netflix и Amazon

Вероятно, двумя образцами идеи, что структура организации и архитектура системы должны быть согласованы, стали Amazon и Netflix. С самого начала в Amazon понимали преимущества того, что команды владеют всем жизненным циклом управляемых систем. В компании хотели, чтобы рабочие коллективы владели системами, за которые они отвечали, и управляли ими на протяжении всего жизненного цикла. Однако в Amazon также знали, что небольшие команды могут работать быстрее, чем большие. Это привело к появлению печально известных *команд на две пиццы*. При этом подходе ни одна команда не должна быть настолько большой, чтобы ее нельзя было накормить двумя пиццами. Это, конечно, не совсем полезная единица измерения, ведь мы никогда не узнаем, едим ли мы пиццу на обед или ужин (или завтрак!) и насколько велика пицца, но общий смысл в том, что оптимальный размер команды — примерно 8–10 человек и эта команда должна быть ориентирована на клиента. Эта движущая сила для небольших коллективов, владеющих всем жизненным циклом своих сервисов,

---

[1] *Nagappan N., Murphy B., Basili V.* The Influence of Organizational Structure on Software Quality: An Empirical Case Study // ICSE ’08: Материалы 30-й Международной конференции по разработке ПО. — ACM, 2008.

[2] И мы все знаем, что в Windows Vista было весьма много багов!

стала основной причиной, по которой в компании Amazon разработали Amazon Web Services. Ведь необходимо было создать инструментарий, позволяющий рабочим группам стать самодостаточными.

В Netflix извлекли уроки из этого примера и с самого начала создавали свою структуру на основе небольших независимых команд, чтобы создаваемые ими сервисы также были независимы друг от друга. Это гарантировало, что архитектура системы будет оптимизирована с учетом скорости изменений. По сути, в Netflix разработали организационную структуру для той архитектуры системы, которую они хотели. Я также слышал, что это распространилось на планы рассадки сотрудников в Netflix — разработчики, чьи сервисы взаимодействуют друг с другом, сидят рядом. Идея в том, что вам понадобится чаще общаться с командами, которые пользуются вашими сервисами или сервисами которых пользуетесь вы сами.

Размер команды

Спросите любого разработчика, насколько большой должна быть команда, и хотя вы получите разные ответы, все согласятся, что в какой-то степени чем меньше, тем лучше. Если попросить их указать «идеальное» число, в основном можно получить ответы в диапазоне от пяти до десяти человек.

Я провел небольшое расследование, чтобы определить наилучший размер команды для разработки ПО. Я нашел множество статей, но многие из них содержат такие нюансы, что экстраполировать их выводы слишком сложно. Лучшее исследование, которое я нашел, — «Эмпирические данные о размере команды и производительности при разработке ПО»[1]. По крайней мере, оно позволило извлечь выгоду из большого массива данных, хотя и не обязательно репрезентативных для разработки ПО в целом. Выводы исследователей показали, что «как и ожидалось из литературы, производительность наихудшая в тех проектах, в которых средний размер команды — девять или более человек». Эта научная работа, по крайней мере, подтверждает мои собственные наблюдения.

Нам нравится работать в небольших группах, и нетрудно понять почему. Когда малочисленная группа людей сосредоточена на одних и тех же результатах, легче сохранять единство и координировать работу. У меня есть (непроверенная) гипотеза, что географическая разобщенность или большие различия в часовых поясах между членами команды вызовут проблемы, способные еще больше ограничить оптимальный размер коллектива, но эту мысль, вероятно, лучше изучить кому-то другому, а не мне.

Итак, маленькие команды — хорошо, большие — плохо. Довольно просто. А если еще и получается выполнять всю необходимую работу в одной команде,

---

[1] *Rodriguez D., Sicilia M. A., Garcia E., Harrison R.* Empirical Findings on Team Size and Productivity in Software Development // Journal of Systems and Software, 85, № 3 (2012). doi.org/10.1016/j.jss.2011.09.009.

тогда вообще отлично! Ваш мир прост, и вы, вероятно, могли бы пропустить большую часть оставшейся в главе информации. Но что, если у вас больше работы, чем времени на нее? Очевидной реакцией на это станет добавление людей. Но, как мы знаем, увеличение количества людей не обязательно поможет добиться большего.

Понимание закона Конвея

Неофициальные и эмпирические данные свидетельствуют о том, что организационная структура оказывает сильное влияние на характер (и качество) создаваемых нами систем. Мы также знаем, что нам нужны небольшие команды. Так как же это понимание поможет? Что ж, по сути, если мы хотим, чтобы слабо связанная архитектура позволяла легче вносить изменения, нам также необходима слабо связанная организация. Иными словами, причина, по которой часто требуется наличие более слабо связанной организации, в том, что мы хотим, чтобы разные части организации могли принимать решения и действовать быстрее и эффективнее, а архитектура слабо связанных систем очень помогает в этом.

В книге «Ускоряйся!» авторы обнаружили серьезную взаимосвязь между организациями, в которых была слабо связанная архитектура, и их способностью более эффективно использовать крупные команды доставки.

> Если мы достигаем слабо связанной хорошо инкапсулированной архитектуры с соответствующей организационной структурой, происходят две важные вещи. Во-первых, мы можем добиться более высокой производительности доставки, увеличив темп и стабильность при одновременном снижении выгорания и проблем развертывания. Во-вторых, можно существенно увеличить размер нашей инженерной организации и линейно — или лучше, чем линейно, — повысить производительность по мере ее увеличения.

В организационном плане сдвиг, который происходит уже некоторое время, особенно для компаний, работающих в больших масштабах, заключается в отходе от централизованных моделей командования и контроля. При централизованном принятии решений скорость реакции нашей организации значительно снижается. Это усугубляется по мере роста предприятия: чем больше оно становится, тем больше его централизованный характер снижает эффективность принятия решений и скорость реагирования.

Организации все больше признают, что если вы хотите масштабировать свою структуру, но при этом быстро продвигаться в разработке, вам необходимо более эффективно распределять ответственность, разрывая централизованное принятие решений и передавая решения в те части предприятия, которые могут работать с большей автономией.

Таким образом, хитрость заключается в создании крупных компаний из небольших автономных команд.

Маленькие команды, большая организация

> Добавление рабочей силы к запаздывающему программному проекту делает его еще более запаздывающим.
>
> *Фред Брукс (закон Брукса)*

В своем знаменитом эссе «Мифический человеко-месяц»[1] Фред Брукс пытается объяснить, почему использование «человеко-месяца» в качестве метода оценки проблематично. Данный метод загоняет нас в ловушку, и мы думаем, что при привлечении большего количества людей к решению проблемы проект будет двигаться быстрее. Теория звучит так: если часть работы займет у разработчика шесть месяцев, то, если добавить второго разработчика, это займет всего три месяца. Если добавить пять разработчиков, а всего теперь их шесть, то работа должна быть выполнена всего за один месяц! Конечно, ПО так не работает.

Чтобы вы могли привлечь больше людей (или команд) к решению проблемы и ускорить ее решение, работа должна быть разделена на задачи, которые можно выполнять в некоторой степени параллельно. Если один разработчик работает над проблемой, а другой в это время ждет, когда первый закончит, то такая работа не может делаться параллельно — только последовательно. Даже если она выполняется параллельно, часто возникает необходимость в координации между людьми, осуществляющими разные функции, что приводит к дополнительным издержкам. Чем более взаимосвязана работа, тем менее эффективно добавлять больше людей.

Если нет возможности разбить задачу на независимые подзадачи, нельзя просто бросить людей на ее решение. Хуже того, это, скорее всего, замедлит работу: добавление новых людей или новых команд имеет свою цену. Необходимо время, чтобы помочь этим людям выйти на нормальный уровень продуктивности, и часто разработчики, у которых и так слишком много проблем, требующих внимания, — это те же разработчики, которым потребуется тратить время, чтобы ввести новых людей в курс дела.

Самым затратным при поставке ПО является необходимость взаимодействия. Чем больше координации между командами, работающими над разными задачами, тем медленнее вы будете работать. В Amazon осознали это и построили свою структуру так, чтобы уменьшить необходимость взаимодействия между своими небольшими «командами на две пиццы». Фактически компания сознательно стремится минимизировать контакты между коллективами именно по этой причине и по возможности ограничить эту координацию теми областями, где

[1] *Брукс Ф. П.* Мифический человеко-месяц, или Как создаются программные системы. — Питер, 2021.

это абсолютно необходимо, — между командами, получившими общую границу между микросервисами. Из книги «Думай как Amazon»[1], написанной бывшим исполнительным директором Amazon Джоном Россманом:

> «Команда на две пиццы» автономна. Взаимодействие с другими командами ограничено, а когда оно происходит, то хорошо документируется, а интерфейсы четко определены. Она владеет всеми аспектами своих систем и несет за них ответственность. Одной из основных целей стало снижение накладных расходов на коммуникации в организациях, включая количество совещаний, координационных центров, планирование, тестирование или выпуски. Более независимые команды работают быстрее.

Очень важно понять, как команда вписывается в более крупную организацию. Книга «Топологии команд» определяет концепцию *команды API*, что в целом определяет, как эта рабочая группа взаимодействует с остальной частью организации, не только с точки зрения интерфейсов микросервисов, но и в отношении методов работы[2].

> Команда API должна точно учитывать удобство использования другими командами. Будет ли им легко и удобно взаимодействовать с нами, или процесс будет сложным и запутанным? Насколько просто будет новой команде освоить наш код и методы работы? Как мы реагируем на запросы и предложения других команд? Будет ли бэклог нашей команды и дорожная карта продукта легко видимыми и понятными для остальных команд?

Об автономии

> В какой бы отрасли вы ни работали, все зависит от ваших сотрудников, от того, насколько правильно они выполняют свои обязанности, насколько они уверены в себе, мотивированы, свободны и хотят реализовать свой потенциал.
>
> *Джон Тимпсон*

Наличие большого количества маленьких команд само по себе не поможет, если они просто станут более изолированными, но все еще будут зависеть от других коллективов в выполнении своих задач. Необходимо убедиться, что каждая из групп обладает достаточной автономией для выполнения работы, за которую она отвечает. Это означает, что следует предоставить командам широкие

[1] *Россман Дж.* Думай как Amazon. 50 и 1/2 идей для бизнеса. — Питер, 2020.

[2] *Skelton M., Pais M.* Team Topologies. — IT Revolution, 2019.

полномочия для принятия решений и инструменты, гарантирующие, что они смогут сделать как можно больше без необходимости постоянно координировать работу с другими командами. Таким образом, обеспечение автономии станет ключевым фактором.

Многие организации продемонстрировали преимущества создания автономных команд. Сохранение небольших организационных групп, позволяющее им создавать тесные связи и эффективно работать вместе без лишней бюрократии, помогло многим компаниям расти и масштабироваться более эффективно, чем это делали некоторые из их коллег. Фирма W. L. Gore and Associates[1] добилась большого успеха, убедившись, что ни одно из ее бизнес-подразделений никогда не насчитывает более 150 человек, чтобы удостовериться, что все знают друг друга. Чтобы эти небольшие подразделения могли работать, им необходимо предоставить полномочия и ответственность для работы в качестве автономных единиц.

Многие из этих организаций, по-видимому, опирались на работу антрополога Робина Данбара, изучавшего способность людей образовывать социальные группы. Его теория заключалась в том, что наши когнитивные способности накладывают ограничения на то, насколько эффективно мы можем поддерживать различные формы социальных отношений. Он подсчитал, что группа может вырасти до 150 человек, прежде чем ей придется отделиться, иначе она рухнет под собственным весом.

Timpson, весьма успешный британский ретейлер, разросся до огромных масштабов за счет расширения прав и возможностей своих сотрудников, сокращения потребности в централизованных функциях и предоставления местным магазинам возможности самостоятельно принимать решения, например, о том, сколько средств нужно вернуть недовольным покупателям. Ныне председатель правления компании Джон Тимпсон известен тем, что отменил внутренние правила и заменил их всего двумя:

- «выгляди подобающе»;
- «положи деньги в кассу».

Автономия работает и в меньших масштабах, и большинство современных компаний, с которыми я сотрудничаю, стремятся создать более автономные команды внутри своих организаций, часто пытаясь скопировать модели других организаций, такие как модель «команды на две пиццы» в Amazon или модель Spotify, которая популяризировала концепцию гильдий и филиалов[2]. Здесь, конечно, я должен предостеречь: во что бы то ни стало учитесь тому, что делают другие организации, но знайте, что бездумное копирование, без понимания, *почему* другая организация делает то, что она делает, практически никогда не приведет к желаемому результату.

---

[1] Известна разработкой водонепроницаемого материала Gore-Tex.

[2] Которую даже Spotify больше не использует.

Если все сделано правильно, автономия команд может расширить возможности людей, помочь им сделать шаг вперед и вырасти, а также быстрее выполнять работу. Когда рабочие группы владеют микросервисами и получают полный контроль над ними, они могут обладать большей автономией в рамках более крупной организации.

Концепция автономии начинает менять наше понимание собственности в архитектуре микросервисов. Рассмотрим этот вопрос более подробно.

Сильное или коллективное владение

В пункте «Определение владения» главы 7 мы обсудили различные типы владения и исследовали их последствия в контексте внесения изменений в код. Вкратце напомню, что мы рассмотрели две основные формы владения кодом.

Сильное владение

Команда владеет микросервисом и сама решает, какие изменения в него внести. Если внешняя команда хочет внести модификации, ей необходимо попросить команду-владельца сделать это от ее имени, либо может потребоваться отправить запрос на извлечение — тогда команда-владелец будет полностью решать, при каких обстоятельствах модель запроса на извлечение будет принята. Команда может владеть более чем одним микросервисом.

Коллективное владение

Любая команда может изменить любой микросервис. Необходима тщательная координация, чтобы гарантировать, что команды не помешают друг другу.

Давайте подробнее рассмотрим последствия применения этих моделей владения и то, как они помогут (или помешают) движению к повышению автономии команды.

Сильное владение

При сильном владении команда, владеющая микросервисом, сама принимает решения. На самом базовом уровне она полностью контролирует, какие изменения вносятся в код. В дальнейшем команда может принять решение о стандартах кодирования, идиомах программирования, когда развертывать ПО, какая технология используется для создания микросервиса, платформы развертывания и многого другого. Благодаря большей ответственности за изменения, происходящие в программном обеспечении, группы с сильным владением получат более высокую степень автономии со всеми вытекающими отсюда преимуществами.

Сильная ответственность в конечном счете сводится к оптимизации автономии этой команды. Возвращаясь к способу ведения дел в Amazon из книги «Думай как Amazon»:

> Когда речь заходит о знаменитых «командах на две пиццы» в Amazon, большинство людей упускают суть. Дело не в размере команды. Речь идет о ее автономии, подотчетности и предпринимательском мышлении. «Команда на две пиццы» — это про создание небольшой команды внутри организации, чтобы она могла работать независимо и гибко.

Модели сильного владения могут обеспечить большую локальную вариативность. Вы можете быть спокойны, например, по поводу того, что одна команда решила создать свой микросервис в функциональном стиле Java, поскольку это решение должно повлиять только на них. Конечно, такой разброс действительно нуждается в некотором смягчении, поскольку некоторые решения требуют определенной последовательности. Например, если все остальные коллективы используют API на основе REST через HTTP для своих конечных точек микросервиса, но вы решили использовать gRPC, это может вызвать некоторые сложности у других людей, желающих воспользоваться вашим микросервисом. С другой стороны, если бы эта конечная точка gRPC использовалась только внутри вашей команды, это не стало бы проблемой. Поэтому при принятии локальных решений, оказывающих влияние на другие команды, координация все еще может потребоваться. Когда и как привлечь более обширную часть организации, мы рассмотрим, когда будем изучать баланс между локальной и глобальной оптимизацией.

По сути, чем более сильную модель владения может принять команда, тем меньше требуется координации и, следовательно, тем более продуктивным может быть коллектив.

Как далеко заходит сильное владение

До этого этапа мы говорили в основном о таких вещах, как внесение изменений в код или выбор технологии. Но концепция владения может быть гораздо глубже. Некоторые организации используют модель, которую я называю *владением полным жизненным циклом*, когда отдельная команда разрабатывает проект, вносит изменения, развертывает микросервис, управляет им в эксплуатации и в конечном счете сворачивает микросервис, когда он больше не требуется.

Эта модель владения полным жизненным циклом еще больше увеличивает автономию команды, поскольку требования к внешней координации снижаются. Тикеты не запрашиваются у операционных команд для развертывания, никакие внешние стороны не подписывают изменения, и команда сама решает, какие изменения следует внести и когда отправить.

Для многих такая модель может показаться причудливой, поскольку у вас уже есть ряд существующих процедур, касающихся рабочего процесса. Возмож-

но, у специалистов вашей команды также нет нужных навыков, чтобы взять на себя полную ответственность, или вам могут потребоваться новые инструменты (например, механизмы самообслуживания при развертывании). Конечно, стоит отметить, что вы не получите полный жизненный цикл владения в одночасье, даже если считаете это желанной целью. Не удивляйтесь, если для этого могут потребоваться годы, особенно в более крупной компании. Многие аспекты владения полным жизненным циклом, скорее всего, потребуют культурных изменений и значительных корректировок с точки зрения ожиданий, поступивших от некоторых ваших сотрудников, таких как необходимость поддержки их ПО в нерабочее время. Но, поскольку вы можете возложить больше ответственности за аспекты микросервиса на свою команду, вы еще сильнее расширите свою автономию.

Я не хочу утверждать, что эта модель каким-либо образом необходима для использования микросервисов, — я твердо убежден, что сильное владение для организаций с несколькими командами будет наиболее разумной моделью для получения максимальной отдачи от микросервисов. Модель сильного владения, связанная с изменениями кода, станет хорошим началом — со временем вы сможете работать над переходом к полному владению жизненным циклом.

Коллективное владение

При модели коллективного владения в микросервис вносятся изменения любой из нескольких команд. Одно из главных преимуществ коллективного владения представлено возможностью перемещать людей туда, где они необходимы. Это может быть полезно, если узкое место в плане доставки вызвано нехваткой людей. Например, некоторые изменения требуют внесения определенных обновлений в микросервис `Оплата`, чтобы обеспечить автоматическое выставление ежемесячных счетов. Вы могли бы просто назначить дополнительных людей для реализации такой функции. Конечно, привлечение людей к решению проблемы не всегда ускоряет процесс работы, но при коллективной ответственности вы, безусловно, обладаете большей гибкостью в этом отношении.

Поскольку команды и люди все чаще переходят от микросервиса к микросервису, требуется более высокая степень согласованности в том, как все это происходит. Вы не можете позволить себе широкий выбор технологий или различные типы моделей развертывания, если ожидается, что разработчик будет работать над разными микросервисами каждую неделю. Чтобы добиться какой-либо степени эффективности от модели коллективного владения микросервисами, в конечном счете потребуется убедиться, что работа с одним микросервисом практически ничем не отличается от работы с любым другим.

По сути, это может подорвать одно из ключевых преимуществ микросервисов. Возвращаясь к цитате, которую мы использовали в начале книги, Джеймс Льюис сказал, что «приобретая микросервисы, вы покупаете себе новые возможности». При более коллективной модели владения вам, вероятно, придется *сократить*

возможности выбора, чтобы обеспечить более высокую степень согласованности в том, что делают команды и как внедряются микросервисы.

Коллективное владение требует высокой степени координации как между отдельными работниками, так и между командами, в которых они находятся. Эта более высокая степень координации приводит к повышению степени связанности на организационном уровне. Возвращаясь к статье Маккормака и др., упомянутой в начале главы, получим следующее наблюдение:

> В организациях с сильной связанностью... даже если это не явный управленческий выбор, проект естественным образом также становится более тесно связанным.

Более тесная координация может привести к большей организационной связанности, что, в свою очередь, приводит к сильной связанности систем. Микросервисы работают лучше всего, когда они могут полностью реализовать концепцию независимого развертывания, а сильно связанная архитектура противоположна нашим стремлениям.

Если у вас небольшое количество разработчиков и, возможно, только одна команда, модель коллективного владения может отлично подойти. Но по мере увеличения числа разработчиков тщательная координация, необходимая для обеспечения коллективного владения, в конечном счете станет существенным негативным фактором с точки зрения получения преимуществ от внедрения микросервисных архитектур.

Уровень команд или уровень организации

Концепции сильной и коллективной ответственности могут применяться на разных уровнях организации. Нужно, чтобы люди в команде были на одной волне и могли эффективно взаимодействовать друг с другом, поэтому надо обеспечить высокую степень коллективной ответственности. В качестве примера это может проявляться в том, что все члены команды могут напрямую вносить изменения в кодовую базу. Многопрофильная команда, сосредоточенная на комплексное предоставление ПО, ориентированного на клиента, должна быть очень хороша в коллективном владении. На уровне организации, если вы хотите, чтобы рабочие группы обладали высокой степенью автономии, важно, чтобы у них также была сильная модель владения.

Баланс моделей

В конечном счете чем больше вы склоняетесь к коллективному владению, тем важнее становится последовательность в том, как все делается. Чем сильнее ваша организация стремится к сильному владению, тем больше допустима локальная оптимизация, как показано на рис. 15.1. Этот баланс не нужно фиксировать. Скорее всего, вы будете менять его в разное время и с учетом разных

факторов. Например, предоставить командам полную свободу в выборе языка программирования, но при этом потребовать от них развертывания на одной облачной платформе.

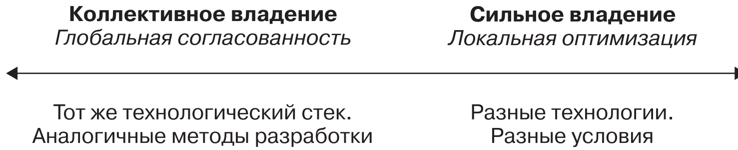


Рис. 15.1. Баланс между глобальной согласованностью и локальной оптимизацией

Однако, по сути, при модели коллективного владения вы почти всегда будете вынуждены придерживаться левого конца этого спектра, то есть требовать более высокой степени глобальной согласованности. По моему опыту, организации, получающие максимальную отдачу от микросервисов, постоянно пытаются найти способы сместить баланс правее. Тем не менее для лучших организаций это является не чем-то неизменным, а скорее тем, что постоянно переоценивается.

Реальность такова, что вы не сможете выполнить какое-либо балансирование, если не будете в той или иной степени знать, что происходит в вашей компании. Даже если вы возлагаете большую ответственность на сами команды, все равно может быть полезно создать в вашей структуре доставки функцию, выполняющую эту балансирующую деятельность.

Команды поддержки

В последний раз мы рассматривали команды поддержки в подразделе «Обмен специалистами» главы 14 в контексте пользовательских интерфейсов, но у этих команд есть более широкое применение. Как описано в книге «Топологии команд», это команды, работающие для поддержки потоковых команд. Везде, где владеющие микросервисами сквозные потоковые команды сосредотачиваются на предоставлении функциональности, ориентированной на пользователя, им требуется помощь других, чтобы выполнять свою работу. При обсуждении UI мы говорили об идее создания команды поддержки, способной помочь другим коллективам в формировании эффективного и последовательного пользовательского опыта. Как показано на рис. 15.2, можно представить это как работу команд поддержки по сопровождению потоковых команд в некотором сквозном аспекте.

Но команды поддержки могут быть разных форм и размеров.

Изучим рис. 15.3. Каждая команда решила выбрать свой язык программирования. Если рассматривать все эти решения по отдельности, кажется, что они

имеют смысл: каждая команда выбрала тот язык программирования, который ей больше всего нравится. Но как насчет организации в целом? Вы хотите поддерживать несколько различных языков программирования? Как это усложняет распределение ролей между командами и как это влияет на наем новых сотрудников, если уж на то пошло?

Рис. 15.2. Команды поддержки сопровождают несколько потоковых команд

Вы можете решить, что на самом деле вам не нужна такая масштабная локальная оптимизация, но вам необходимо знать о различных вариантах выбора и иметь возможность обсуждать эти изменения, если вы хотите получить определенную степень контроля. Обычно именно в этом контексте я вижу очень небольшую группу поддержки, работающую в разных командах и помогающую объединить людей, чтобы эти обсуждения проходили должным образом.

Именно в таких сквозных командах поддержки должны работать архитекторы, по крайней мере какое-то время. Старомодный архитектор говорил бы людям, что делать. В новой, современной, децентрализованной организации архитекторы изучают ландшафт, выявляют тенденции, помогают объединять людей и выступают в качестве рупора, помогающего другим командам выполнять работу. Они не представляют собой единицу управления, они стали еще одной вспомогательной функцией (часто с новым названием: я видел такие термины, как *«главный инженер»*, используемые для людей, играющих роль, как мне кажется, архитектора). Мы подробнее рассмотрим роль архитекторов в главе 16.

Команда поддержки должна помочь выявить проблемы, которые лучше решать вне команды. Рассмотрим сценарий, в котором командам было трудно создавать БД с тестовыми данными. Каждая команда работала над решением

данного вопроса по-разному, но проблема никогда не была настолько важной, чтобы какая-либо команда могла исправить ее должным образом. Но затем мы просматриваем несколько команд и замечаем, что многие из них могли бы извлечь выгоду из правильного решения, и внезапно становится очевидным, что задачу необходимо решать.

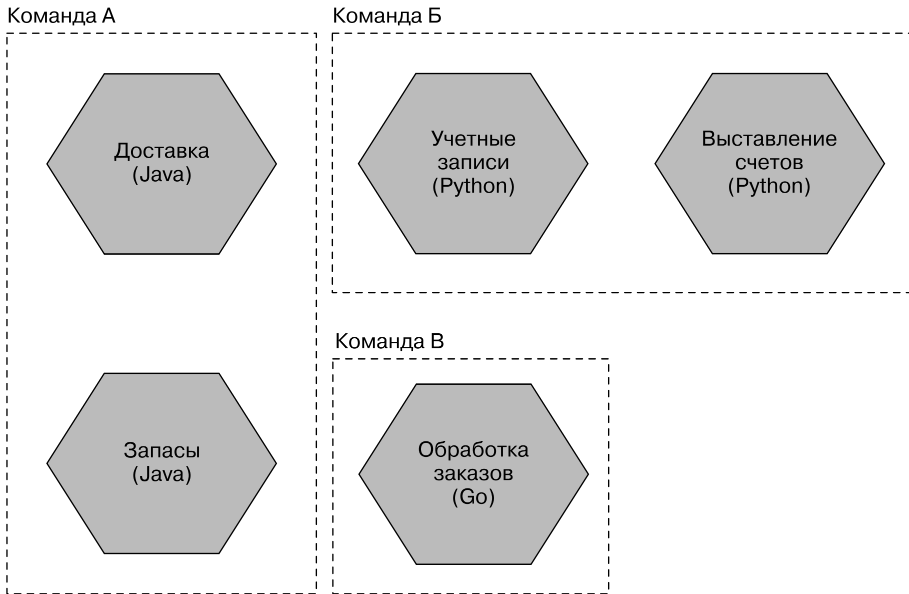


Рис. 15.3. Каждая команда выбрала свой язык программирования

Профессиональные сообщества

Профессиональное сообщество (community of practice, CoP) — это межсекторальная группа, способствующая обмену опытом и получению знаний между коллегами. Профсообщества — это фантастический способ создать организацию, в которой люди могут постоянно учиться и расти. В одной замечательной книге на эту тему Эмили Веббер[1] пишет:

> Профессиональные сообщества создают правильную среду для социального, эмпирического обучения и реализации комплексной учебной программы, что приводит к ускоренному обучению участников... Это может способствовать формированию культуры обучения, в которой люди ищут лучшие способы выполнять задачи, а не только используют существующие модели.

[1] *Webber E.* Building Successful Communities of Practice. — Blurb, 2016.

Я полагаю, что в некоторых случаях одна и та же группа может быть *и* CoP, *и* командой поддержки, но, по моему собственному опыту, это редкость. Однако здесь есть определенное наложение. Как команды поддержки, так и профсообщества дают представление о том, что происходит в разных командах вашей организации. Эта информация поможет понять, нужно ли вам изменить баланс между глобальной и локальной оптимизацией или выявить необходимость в дополнительной централизованной помощи, но разница здесь заключается в ответственности и возможностях группы.

Члены команды поддержки часто работают на полную ставку как часть команды, или же у них значительная часть рабочего времени отведена для такой цели. Таким образом, у них больше возможностей для воплощения изменений в жизнь — для реальной работы с другими командами и оказания им помощи. Профессиональные сообщества больше ориентированы на обучение: люди в группе обычно участвуют в обсуждениях не более нескольких часов в неделю, и состав такой группы часто меняется.

CoP и команды поддержки, конечно, могут очень эффективно работать вместе. Часто CoP способно предоставить ценную информацию, помогающую команде поддержки лучше понять ситуацию. Рассмотрим Kubernetes CoP, которое делится своим опытом о проблематичной работе над кластером разработки своей компании с командой платформы, управляющей кластером. Что касается команд платформы, то это тема, которую стоит рассмотреть более подробно.

Платформа

Возвращаясь к нашим слабо связанным потоковым командам, ожидается, что они будут проводить собственное тестирование в изолированных средах, управлять развертываниями таким образом, чтобы их можно было выполнять в течение дня, и вносить изменения в свою системную архитектуру, когда это необходимо. Все это, похоже, возлагает все больше ответственности и работы на эти команды. Здесь может помочь общая концепция команд поддержки, но в конечном счете потоковым командам нужен набор инструментов самообслуживания, позволяющий им выполнять свою работу, — это платформа.

На самом деле без платформы вам может быть трудно изменить организацию. В статье «Конвергенция к Kubernetes»[1] технический директор RVU Пол Динглз делится опытом работы над сайтом сравнения цен Uswitch, который переходит от прямого использования низкоуровневых сервисов AWS к более абстрактной платформе, основанной на Kubernetes. Идея заключалась в том, что эта платформа позволяла потоковым командам RVU больше сосредоточиться

[1] *Ingles P.* Convergence to Kubernetes // Medium, 18 июня 2018 года, https://oreil.ly/Ho7kY.

на предоставлении новых функций и тратить меньше времени на управление инфраструктурой. Как выразился Пол:

> Мы изменили нашу организацию не потому, что хотели использовать Kubernetes, — мы использовали Kubernetes, потому что хотели изменить нашу организацию.

Платформа, способная реализовать общие функции, такие как возможность управления желаемым состоянием микросервисов, агрегирование логов и авторизация и аутентификация между микросервисами, может значительно повысить производительность и позволить командам брать на себя больше ответственности без необходимости резко увеличивать объем выполняемой ими работы. Фактически платформа должна предоставлять командам больше пропускной способности, чтобы они могли сосредоточиться на предоставлении функций.

Команда платформы

Платформе нужен кто-то, кто будет ее запускать и управлять ею. Эти технологические стеки могут быть достаточно сложными, и будут требовать специальных знаний. Однако меня беспокоит то, что иногда команды платформ слишком легко упускают из виду цели своего существования.

У команды платформы есть пользователи точно так же, как и у любой другой команды. Пользователями команды платформы будут другие разработчики. Ваша задача, если вы входите в состав команды платформы, состоит в том, чтобы облегчить им жизнь (это, конечно, работа любой команды поддержки). Это означает, что создаваемая вами платформа должна соответствовать потребностям команд, использующих ее. Вам также нужно работать с командами не только для того, чтобы помочь им пользоваться платформой должным образом, но и чтобы учитывать их отзывы и требования для ее улучшения.

В прошлом я предпочитал называть такую команду чем-то вроде службы доставки или поддержки доставки, чтобы лучше сформулировать ее цель. На самом деле задача команды платформы заключается *не* в создании платформы, а в том, чтобы упростить разработку и доставку функциональности. Создание платформы — это всего лишь один из способов, с помощью которого члены команды платформы могут достичь своих целей. Я действительно переживаю, что, называя себя командой платформы, они будут рассматривать все проблемы как вещи, которые могут и должны быть решены платформой, вместо того чтобы более широко думать о других способах облегчить жизнь разработчикам.

Как и любая хорошая команда поддержки, команда платформы должна в какой-то степени работать почти как внутренний консультант. Если вы работаете в команде платформы, вам нужно выяснить, с какими проблемами сталкиваются люди, чтобы помочь им с решением. Но, поскольку в конечном счете вы создаете платформу, вам также необходимо провести большую работу

по разработке продукта. На самом деле подход к разработке продукта при создании вашей платформы — замечательная идея, которая может стать отличным способом помочь вырастить новых владельцев продуктов.

Мощеная дорога

Концепция, которая стала популярной при разработке ПО, — это принцип мощеной дороги. Ее смысл в том, что вы сообщаете четкие требования по выполнению задач, а затем предоставляете механизмы, с помощью которых это можно легко сделать. Например, вам может потребоваться убедиться, что все микросервисы взаимодействуют через MTLS. Затем вы могли бы предоставить общий фреймворк или платформу развертывания, которая автоматически обеспечивала бы MTLS для микросервисов, работающих на нем. Платформа может стать отличным способом доставки по этой мощеной дороге.

Ключевой момент концепции мощеной дороги в том, что ее использование не является обязательным, она просто обеспечивает более легкий способ добраться до места назначения. Таким образом, если бы команда хотела обеспечить связь своих микросервисов через MTLS без использования общей платформы, ей пришлось бы найти какой-то другой способ, но это все равно было бы допустимо. Аналогия в том, что, хотя нужно, чтобы все добирались до одного и того же пункта назначения, люди вольны сами выбирать свой путь. И есть надежда, что мощеная дорога — это самый простой способ добраться туда, куда вам нужно.

Концепция мощеной дороги направлена на то, чтобы упростить работу с общими случаями, оставляя при этом место для исключений, когда это оправданно.

Если представлять платформу как мощеную дорогу, то, считая ее необязательной, вы стимулируете команду платформы к тому, чтобы сделать платформу простой в использовании. Вот снова Пол Инглз, об оценке эффективности команды платформы[1]:

> Мы устанавливали [цели и ключевые результаты (OKR)] по количеству команд, которые хотели бы внедрить платформу, по количеству приложений, использующих сервис автоматического масштабирования платформы, по доле приложений, переключенных на сервис динамических учетных данных платформы, и т. д. Некоторые из этих показателей мы отслеживали в течение более длительных периодов времени, а другие были полезны для определения прогресса в течение квартала, а затем мы отказывались от них в пользу чего-то другого.
>
> Мы никогда не требовали обязательного использования платформы, поэтому установление ключевых результатов по количеству подключенных команд заставило нас сосредоточиться на решении проблем, которые способствовали бы принятию платформы. Мы также ищем естественные показатели прогресса: доля трафика, обслуживаемого платформой, и доля доходов, получаемых через сервисы платформы, могут служить хорошими примерами.

---

[1] Organizational Evolution for Accelerating Delivery of Comparison Services at Uswitch // Team Topologies. 24 июня 2020 года. https://oreil.ly/zoyvv.

Когда вы ставите на пути людей барьеры, кажущиеся необоснованными и капризными, они найдут способы обойти их, чтобы выполнить работу. Так что в целом я считаю, что гораздо эффективнее объяснять, почему что-то должно быть сделано определенным образом, а затем облегчать выполнение этих задач, чем пытаться сделать невозможным выполнение того, что вам не нравится.

Переход к более автономным потоковым командам не отменяет необходимости иметь четкое техническое видение или четко представлять определенные вещи, которые должны делать все команды. Если существуют конкретные ограничения (например, быть независимым от поставщика облачных услуг) или требования, которым должны подчиняться все команды (все PII должны быть зашифрованы в состоянии покоя с использованием определенных алгоритмов), то о них все равно необходимо четко сообщить, а также объяснить причины их возникновения. В таком случае платформа может сыграть определенную роль в упрощении выполнения этих задач. Используя платформу, вы идете по мощеной дороге — в конечном счете вы будете делать все правильно, не затрачивая особых усилий.

С другой стороны, я вижу, что некоторые организации пытаются управлять всем с помощью платформы. Вместо того чтобы четко сформулировать, что нужно сделать и почему, они просто говорят использовать ту или иную платформу. Проблема такого подхода в том, что, если платформа не проста в использовании или не подходит для конкретного случая использования, люди найдут способы обойти саму платформу. Когда команды работают за пределами платформы, у них нет четкого представления, какие ограничения важны для организации, и они заметят, что неосознанно делают «неправильные» вещи.

Общие микросервисы

Как я уже говорил, я большой сторонник сильных моделей владения микросервисами. Как правило, один микросервис должен принадлежать одной команде. Несмотря на это, я по-прежнему считаю обычным делом, когда микросервисы принадлежат нескольким командам. Почему так? И что вы можете (или должны) с этим сделать? Важно понять причины, по которым микросервисы принадлежат нескольким командам, тем более что, возможно, мы сможем найти несколько привлекательных альтернативных моделей, способных решить основные проблемы людей.

Слишком трудно разделить

Очевидно, что одна из причин, по которой вы можете столкнуться с микросервисом, принадлежащим более чем одной команде, заключается в слишком высокой стоимости разделения микросервиса на части. Это обычное явление в больших монолитных системах. Если это главная проблема, с которой вы сталкиваетесь,

то я надеюсь, что некоторые советы, приведенные в главе 3, окажутся полезными. Вы также могли бы рассмотреть возможность объединения команд, чтобы более тесно увязать их с самой архитектурой.

FinanceCo, финтех-компания, с которой мы познакомились в разделе «На пути к потоковым командам» главы 14, в основном использует сильную модель владения с высокой степенью автономии. Тем не менее у компании все еще есть существующая монолитная система, которая медленно разделялась на части. Это монолитное приложение, по сути, было общим для нескольких команд, и увеличение затрат на работу в этой общей кодовой базе было очевидным.

Сквозные изменения

Многое из того, что мы до сих пор обсуждали относительно взаимодействия организационной структуры и архитектуры, направлено на уменьшение необходимости координации между командами. Большая часть этого заключается в том, чтобы попытаться максимально сократить сквозные преобразования. Однако мы должны признать, что некоторые подобные изменения могут быть неизбежны.

FinanceCo столкнулась с одной из таких проблем. Когда этот процесс начинался, одна учетная запись была привязана к одному пользователю. По мере того как компания росла и привлекала все больше бизнес-пользователей (ранее она была больше ориентирована на рядовых потребителей), это стало ограничением. Компания хотела перейти к модели, в которой одна учетная запись в FinanceCo могла бы обслуживать несколько пользователей. Это было фундаментальное изменение, поскольку до этого момента в системе предполагалось, что «одна учетная запись = один пользователь».

Для того чтобы это изменение произошло, была сформирована отдельная команда. Однако огромный объем работы включал преобразования микросервисов, уже принадлежащих другим командам. Получалось, что работа этой команды частично сводилась к внесению изменений и отправке запросов на извлечение или обращению к другим командам с просьбой внести изменения. Координация изменений была очень болезненной, поскольку для поддержки новой функциональности требовалось модифицировать значительное количество микросервисов.

Проводя реструктуризацию команды и архитектуры, чтобы исключить один набор сквозных изменений, мы фактически подвергаем себя другому набору сквозных изменений, способных оказать более значительное влияние. Так было в случае с FinanceCo — реорганизация, которая потребовалась бы для снижения стоимости многопользовательской функции, привела бы к увеличению затрат на внесение других, более распространенных по системе изменений. В FinanceCo понимали, что это конкретное преобразование будет очень проблематичным, но это был настолько исключительный тип изменений, что эти трудности были приемлемы.

Узкие места в доставке

Одна из ключевых причин, по которой люди переходят к коллективному владению, когда микросервисы распределяются между командами, заключается в том, чтобы избежать узких мест в доставке. Что делать, если накопилось большое количество изменений, которые необходимо внести в один сервис? Давайте вернемся к MusicCorp и представим, что мы предоставляем клиенту возможность видеть жанр композиции в наших продуктах, а также добавляем совершенно новый тип ассортимента — виртуальные музыкальные рингтоны для мобильного телефона. Команде веб-сайта необходимо внести изменения, чтобы отобразить информацию о жанре, а команда мобильных приложений работает над тем, чтобы пользователи могли выбирать, просматривать и покупать рингтоны. Обе модификации необходимо внести в микросервис `Каталог`, но, к сожалению, половина команды застряла на диагностике сбоя в эксплуатируемой системе, а вторая половина получила пищевое отравление после недавней вылазки команды к появившемуся из переулка фудтраку.

Есть несколько вариантов избежать необходимости совместного использования микросервиса `Каталог` веб-сайтом и командами мобильной разработки. Первый — это просто ждать. Команды веб-сайта и мобильных приложений переключаются на что-то другое. Проблема это или нет, зависит от того, насколько важна функция или насколько длительной может быть задержка.

Вместо этого можно было бы добавить людей в команду, отвечающую за `Каталог`, чтобы помочь им быстрее выполнять свою работу. Чем более стандартизированы технологический стек и программные идиомы, используемые в вашей системе, тем легче другим людям преобразовывать ваши сервисы. Обратная сторона, конечно, как мы обсуждали ранее, заключается в том, что стандартизация, как правило, снижает способность команды принимать правильные решения и может привести к различного рода неэффективности.

Другим вариантом, который позволил бы избежать совместного использования микросервиса `Каталог`, может быть разделение каталога на отдельный общий музыкальный каталог и каталог рингтонов. Если изменения, вносимые в код поддержки рингтонов, довольно незначительны и вероятность того, что эта область будет активно развиваться в будущем, также достаточно мала, это вполне может быть преждевременным. С другой стороны, если в течение 10 недель будут накапливаться функции, связанные с рингтонами, разделение сервиса может иметь смысл и владение будет отдано мобильной команде.

Однако есть пара других моделей, требующих изучения. Сейчас мы рассмотрим, что можно сделать с точки зрения «подключаемости» совместно используемого микросервиса, позволяя другим командам либо вносить свой код через библиотеки, либо расширять его с помощью общей платформы. Однако сначала необходимо изучить возможность привнести в нашу компанию некоторые идеи из мира разработки с открытым исходным кодом.

Решение с открытым исходным кодом внутри компании

Многие организации решили внедрить ту или иную форму работы с открытым исходным кодом внутри компании как для того, чтобы помочь решить проблему общих кодовых баз, так и для того, чтобы людям вне команды было проще вносить изменения в микросервис, который они, возможно, используют.

При обычном использовании решений с открытым исходным кодом небольшая группа людей считается доверенными коммиттерами. Они хранители кода. Если вы хотите внести изменения в проект с открытым исходным кодом, либо вы просите кого-то из коммиттеров сделать это за вас, либо вносите изменения сами и отправляете им запрос на извлечение. Доверенные коммиттеры по-прежнему отвечают за кодовую базу — они являются владельцами.

Внутри организации данный шаблон тоже может хорошо работать. Возможно, люди, которые изначально работали над сервисом, больше не работают в одной команде: например, они разбросаны по всей организации. Если у них все еще есть права на фиксацию кода, вы можете найти их и попросить о помощи, возможно объединившись с ними. При наличии подходящих инструментов вы можете отправить им запрос на извлечение.

Роль доверенных коммиттеров

Мы по-прежнему хотим, чтобы сервисы были осмысленными. Хотим, чтобы код был достойного качества, а сам микросервис демонстрировал некоторую согласованность в том, как он собран. Также необходимо убедиться, что преобразования, вносимые сейчас, не сделают запланированные изменения намного сложнее, чем они должны быть. Поэтому нужно использовать шаблоны обычного проекта с открытым исходным кодом. Это означает отделение доверенных коммиттеров (основная команда) от ненадежных коммиттеров (люди из-за пределов команды, отправляющие изменения).

У основной команды владельцев должен быть какой-то способ проверки и утверждения изменений. Им необходимо убедиться, что изменения идиоматически согласованы, то есть соответствуют общим рекомендациям по программированию остальной части кодовой базы. Поэтому людям, проводящим проверку, придется потратить время на работу с отправителями изменений, чтобы подтвердить уровень качества каждого вносимого изменения.

Столкнувшись с обоими вариантами поведения, я могу сказать одно: любой способ требует времени. Рассматривая возможность предоставления ненадежным коммиттерам права вносить изменения в кодовую базу, нужно решить, стоят ли затраты на работу гейткипера таких хлопот: может ли основная команда заниматься более полезным делом в тот период времени, который тратится?

Завершенность

Чем менее стабильным или завершенным является сервис, тем сложнее разрешить людям, не входящим в основную команду, вносить исправления. Пока не создана конструктивная основа сервиса, команда может не знать, что значит «хорошо», и поэтому ей будет трудно понять, как выглядит хорошее представление. На этом этапе сам сервис претерпевает значительные изменения.

Большинство проектов с открытым исходным кодом, как правило, не принимают заявки от более широкой группы ненадежных коммиттеров до тех пор, пока не будет готово ядро первой версии. Следование аналогичной модели для вашей собственной организации имеет смысл. Если сервис довольно зрелый и редко меняется, например наш сервис корзины, то настало время открыть его для других участников.

Инструментарий

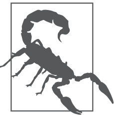
Чтобы наилучшим образом поддерживать внутреннюю модель с открытым исходным кодом, вам понадобится некоторый инструментарий. Важно использовать распределенный инструмент управления версиями, позволяющий пользователям отправлять запросы на извлечение (или что-то подобное). В зависимости от размера организации вам также могут понадобиться инструменты, позволяющие обсуждать и изменять запросы на исправление. Это может означать или не означать полноценную систему проверки кода, но возможность комментировать исправления очень полезна. Наконец, вам нужно сделать так, чтобы коммиттеру было очень легко собрать и развернуть ваше ПО и сделать его доступным для других. Обычно это предполагает наличие четко определенных конвейеров сборки и развертывания и централизованных хранилищ артефактов. Чем более стандартизирован ваш технологический стек, тем проще работникам из других команд вносить изменения и предоставлять исправления для микросервиса.

Подключаемые модульные микросервисы

В FinanceCo я столкнулся с интересной проблемой, связанной с одним микросервисом, который становился узким местом для многих команд. Для каждой страны у FinanceCo были выделенные команды, концентрирующие свои усилия на функциональности, специфичной для конкретной страны. В этом был большой смысл, поскольку у каждой страны были особые требования и проблемы. Но это создавало трудности для центрального сервиса, который нуждался в обновлении определенных функций для каждой страны. Команда, владеющая данным микросервисом, была перегружена отправленными ей запросами на извлечение.

Она отлично справлялась с быстрой обработкой этих запросов и фактически это стало основной частью обязанностей этой команды, но структурно ситуация не была достаточно устойчивой.

Это пример того, как команда, имеющая большое количество запросов на извлечение, может служить признаком множества различных потенциальных проблем. Принимаются ли всерьез запросы на извлечение от других команд? Или эти запросы становятся признаком того, что микросервис потенциально должен сменить владельца?



Если команде поступает много входящих запросов на извлечение, это может быть признаком того, что у вас существует микросервис, который совместно используется несколькими командами.

Изменение владения

Иногда правильнее всего изменить владельца микросервиса. Рассмотрим это на примере MusicCorp. Команде по взаимодействию с клиентами приходится отправлять большое количество запросов на извлечение, связанных с микросервисом `Рекомендации`, в отдел маркетинга и рекламных акций. Это связано с тем, что происходит ряд изменений в управлении информацией о клиентах, а также с тем, что нам нужно по-разному применять эти рекомендации.

В данной ситуации команде по взаимодействию с клиентами имеет смысл просто взять на себя управление микросервисом `Рекомендации`. Однако в примере с FinanceCo такой возможности не существовало. Проблема заключалась в том, что запросы на извлечение исходили от *нескольких* разных команд. Так что же еще можно было сделать?

Запустить несколько вариаций

Один из рассматриваемых вариантов заключался в том, что каждая команда, отвечающая за определенную страну, будет запускать свою собственную версию общего микросервиса. Таким образом, команда США запустила бы свою версию, команда Сингапура — свою и т. д.

Конечно, проблема с таким подходом заключается в дублировании кода. Совместно используемый микросервис реализовал набор стандартных моделей поведения и общих правил, но также требовалось, чтобы некоторые из этих функциональных возможностей менялись для каждой страны. Мы не хотели дублировать общую функциональность. Идея заключалась в том, чтобы команда, которая в настоящее время управляла общим микросервисом, вместо этого предоставила фреймворк, который на самом деле состоял из существующего микросервиса, но только с общей функциональностью. Каждая команда, работающая в конкретной стране, может запустить свой собственный экземпляр

скелета микросервиса, подключив к нему свои собственные пользовательские функции, как показано на рис. 15.4.

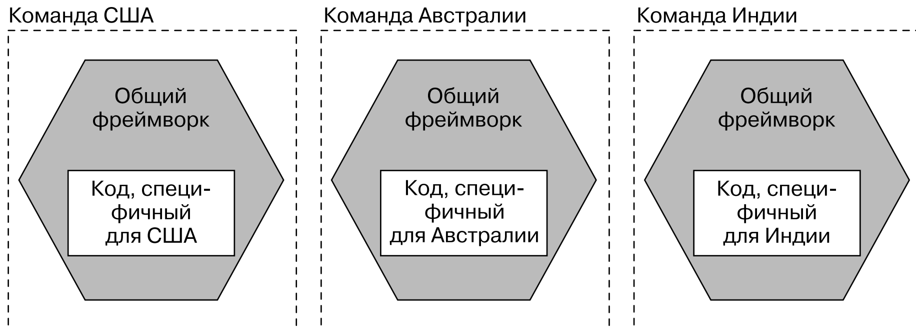


Рис. 15.4. Общая структура могла бы позволить различным командам управлять несколькими вариантами одного и того же микросервиса

Здесь важно отметить, что, хотя мы можем использовать общую функциональность в данном примере для каждого варианта микросервиса, специфичного для конкретной страны, она не может быть обновлена во всех вариантах микросервиса одновременно, без масштабного поэтапного релиза. Основная команда, управляющая фреймворком, может сделать новую версию доступной, но каждая команда должна будет использовать последнюю версию общего кода и повторно развернуть ее. В такой ситуации в FinanceCo не возражали против подобного ограничения.

Стоит подчеркнуть, что эта конкретная ситуация встречалась довольно редко и я сталкивался с ней всего один или два раза. Изначально я сосредоточился на поиске способов либо разделить обязанности этого центрального общего микросервиса, либо переназначить права собственности. Меня беспокоило, что создание внутренних фреймворков может быть сопряжено с большими трудностями. Слишком легко раздуть фреймворк или ограничить процесс разработки команд, которые его используют. Такая проблема не появляется в первый же день. При создании внутренней структуры все начинается с благих намерений. Хотя в ситуации с FinanceCo чувствовалось, что это правильный путь развития, я бы предостерег от слишком поспешного принятия такого подхода, если у вас есть другие варианты в запасе.

Сторонний вклад через библиотеки

Разновидностью этого подхода может быть предоставление каждой командой, отвечающей за конкретную страну, библиотеки с функциональностью, специфичной для данной страны, и последующая упаковка этих библиотек в один общий микросервис, как показано на рис. 15.5.

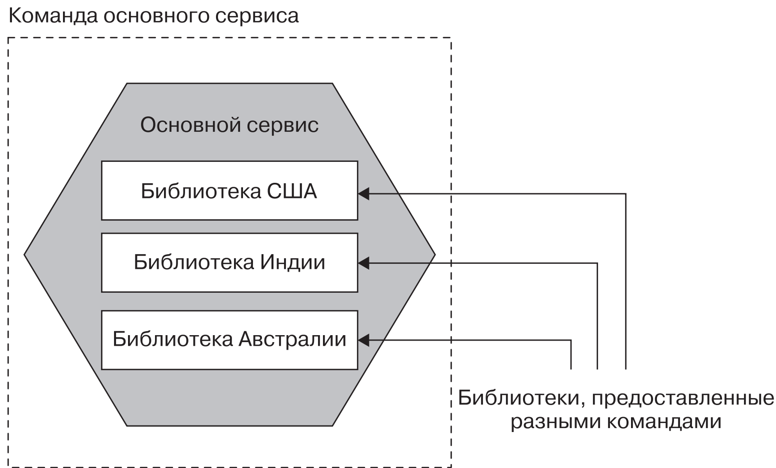


Рис. 15.5. Команды предоставляют библиотеки со своим пользовательским поведением в центральный микросервис

Идея здесь заключается в том, что, если американской команде необходимо реализовать специфичную для США логику, она вносит изменения в библиотеку, которая затем включается как часть сборки центрального микросервиса.

Такой подход уменьшает необходимость запуска дополнительных микросервисов. Нам не нужно запускать по одному сервису для *каждой* страны, можно запустить один центральный микросервис, обрабатывающий пользовательские функции для всех стран. Проблема заключается в том, что команды стран не отвечают за принятие решения о том, когда их индивидуальные функции будут запущены в эксплуатацию. Они могут внести изменение и запросить развертывание, но планировать его придется уже центральной команде.

Кроме того, возможно, что ошибка в одной из этих библиотек, специфичных для конкретной страны, может вызвать проблему в эксплуатации, а отвечать за это придется центральной команде. В результате это может усложнить устранение неполадок в эксплуатации.

Тем не менее данный вариант стоит рассмотреть, если он поможет отказаться от централизованного микросервиса, находящегося в коллективной собственности, особенно когда нет возможности оправдать запуск нескольких вариантов одного и того же микросервиса.

Обзоры изменений

При использовании внутреннего подхода с открытым исходным кодом концепция проверки становится ключевым принципом: изменение должно быть

рассмотрено, прежде чем его можно будет принять. Но даже при работе внутри команды над кодовой базой, где у вас есть прямые разрешения на фиксацию кода, все равно полезно проверять свои изменения.

Я большой поклонник того, чтобы мои изменения рецензировались. И я всегда чувствовал, что мой код только выигрывает от второй пары глаз. Наиболее предпочтительной формой анализа я считаю немедленный обзор, который можно получить в рамках парного программирования. Вы и другой разработчик пишете код вместе и обсуждаете изменения друг с другом. В таком случае код будет просматриваться до проведения фиксации.

Не стоит верить мне на слово. Сошлюсь на неоднократно упомянутую книгу «Ускоряйся!»:

> Мы обнаружили, что одобрение только изменений с высокой степенью риска не коррелировало с производительностью доставки ПО. Команды, сообщившие об отсутствии процесса одобрения или использовавшие коллегиальную оценку, добились более высоких показателей доставки ПО. Наконец, команды, которым требовалось одобрение внешнего органа, добились более низкой производительности.

Здесь мы видим различие между коллегиальной оценкой и внешней проверкой изменений. Коллегиальная проверка изменений выполняется кем-то, кто, по всей вероятности, входит в ту же команду, что и вы, и работает над той же кодовой базой. Очевидно, что они отлично разбираются в том, какое изменение хорошее, а также, возможно, проведут анализ быстрее (подробнее об этом — в ближайшее время). Внешний обзор не такой тщательный. Поскольку им занимается человек, не имеющий отношения к вашей команде, он, скорее всего, будет оценивать изменения по списку критериев, которые могут иметь или не иметь смысл, и, поскольку он находится в отдельной команде, он может получить к вашим изменениям с задержкой. Как отмечают авторы «Ускоряйся!»:

> Каковы шансы на то, что посторонний человек, не очень хорошо знакомый с внутренними компонентами системы, сможет просмотреть десятки тысяч строк кода, измененных потенциально сотнями инженеров, и точно определить влияние на сложную производственную систему?

Итак, в целом мы хотим использовать коллегиальные проверки изменений и избежать необходимости внешних проверок кода.

Синхронные и асинхронные обзоры кода

При парном программировании проверка кода происходит естественным образом во время написания кода. На самом деле это нечто большее. При парной работе есть ведущий (человек за клавиатурой) и навигатор (который

действует как вторая пара глаз). Оба участника находятся в постоянном диалоге: проверка и внесение изменений происходят одновременно. Обзор становится неявным постоянным аспектом парных отношений. Это означает, что, как только замечена ошибка, она немедленно исправляется.

Если вы не работаете в паре, идеальным вариантом может стать проверка в очень краткие сроки после написания кода. Тогда нужно, чтобы сам анализ был как можно более синхронным. У вас должна быть возможность напрямую обсуждать с рецензентом любые возникающие у него вопросы, совместно согласовывать дальнейшие действия, вносить изменения и двигаться дальше.

Чем быстрее вы получите обратную связь по изменению кода, тем скорее сможете просмотреть и оценить ее, попросить разъяснений, при необходимости продолжить обсуждение проблемы и в конечном счете внести любые необходимые изменения. Чем больше времени проходит между отправкой изменения кода на проверку и фактическим ее проведением, тем дольше и сложнее все становится.

Если вы отправляете код на проверку и не получаете отзыв через несколько дней, вы, скорее всего, перейдете к другой работе. Чтобы обработать поступившую обратную связь, вам нужно будет переключиться и возобновить выполняемую ранее работу. Вы можете согласиться с изменениями рецензента (если требуется), внести их и повторно отправить на утверждение. В худшем случае продолжится дальнейшее обсуждение поднятых вопросов. Этот асинхронный обмен сообщениями между отправителем и рецензентом может добавить несколько *дней* к процессу внесения изменений.

Делайте обзоры кода быстро!

Если вы хотите проводить обзоры кода и не занимаетесь парным программированием, сделайте обзор как можно быстрее после отправки изменений и просматривайте отзывы как можно более синхронно, в идеале — один на один с рецензентом.

Групповое программирование

Групповое или ансамблевое программирование (оно же моб-программирование) иногда обсуждается как способ выполнения анализа кода по умолчанию. При групповом программировании большая группа людей (возможно, вся команда) работает вместе над изменением. Речь идет в первую очередь о совместной работе над проблемой и получении информации от большого числа людей.

Из команд, с которыми я общался, применяющих такой вид программирования, большинство используют его лишь изредка для решения конкретных сложных задач или важных изменений, но большая часть разработки выпол-

няется и вне коллектива. Таким образом, хотя занятия ансамблевым программированием, вероятно, обеспечат достаточный синхронный анализ изменений, вносимых во время самой работы команды, вам все равно понадобится способ обеспечить обзор изменений, внесенных вне группы.

Некоторые из вас могут возразить, что вам просто нужно провести обзор изменений с высоким риском и поэтому достаточно проводить обзор в составе группы только таких изменений. Стоит отметить, что авторы «Ускоряйся!» неожиданно *не* обнаружили корреляции между производительностью доставки ПО и рассмотрением исключительно изменений с высоким риском, по сравнению с положительной корреляцией, когда все изменения проходят коллективную оценку. Так что, если вы действительно хотите участвовать в групповом программировании, вперед! Но вы, возможно, захотите изучить возможность пересмотра и других изменений, внесенных за пределами группы.

Лично у меня есть некоторые серьезные сомнения по поводу некоторых аспектов данного способа программирования. Вы заметите, что ваша команда на самом деле представляет собой разнородную группу, и дисбаланс сил в команде может еще больше подорвать цель коллективного решения проблем. Не всем комфортно работать в группе. Некоторые люди будут процветать в такой среде, в то время как другие будут чувствовать себя совершенно неспособными внести свой вклад.

Когда я поднимал этот вопрос с некоторыми сторонниками группового программирования, я получил множество ответов, большая часть из которых сводятся к убеждению, что, если вы создадите правильную среду для работы, любой сможет «выйти из своей скорлупы и внести свой вклад». Давайте просто скажем, что после таких разговоров я так сильно закатил глаза, что чуть не ослеп. Справедливости ради те же самые опасения могут быть высказаны и по поводу парного программирования!

Хотя я не сомневаюсь, что большинство сторонников ансамблевого программирования будут достаточно осведомленными или внимательными, важно помнить, что создание инклюзивного рабочего пространства отчасти связано с пониманием того, как создать среду, в которой все члены команды могут вносить полноценный вклад безопасным и удобным способом. И не обманывайте себя тем, что каждый фактически вносит свой вклад только потому, что вы собрали всех в одной комнате. Если вы хотите получить несколько конкретных советов по коллективному программированию, то я бы посоветовал прочитать самостоятельно изданное краткое «Руководство по ансамблевому программированию» Маарет Пюхяярви[1].

[1] *Pyhäjärvi M.* Ensemble Programming Guidebook (самоиздание, 2015–2020). https://ensembleprogramming.xyz.

Осиротевший сервис

Так что же насчет сервисов, которые больше не поддерживаются активно? По мере того как мы переходим к более детализированным архитектурам, сами микросервисы становятся меньше. Одним из преимуществ небольших микросервисов, как мы уже обсуждали, стало их упрощение. Более простые микросервисы с меньшей функциональностью некоторое время не нуждаются в замене. Рассмотрим микросервис `Корзина покупок`, который предоставляет некоторые довольно скромные возможности: добавить в корзину, удалить из корзины и т. д. Вполне возможно, что данный микросервис не придется менять в течение нескольких месяцев после первичного написания, даже если активная разработка все еще продолжается. Так что же происходит? Кому принадлежит этот микросервис?

Если структуры вашей команды согласованы с ограниченными контекстами вашей организации, то даже у сервисов, которые нечасто меняются, все равно есть фактический владелец. Представьте себе команду, ориентированную на контекст потребительских веб-продаж. Она может обрабатывать пользовательский веб-интерфейс и микросервисы `Корзина покупок` и `Рекомендации`. Даже если сервис корзины не менялся месяцами, естественно, что эта команда внесет изменения, если потребуется. Одно из преимуществ микросервисов, конечно, заключается в том, что, если команде нужно модифицировать микросервис, добавив новую функцию, и команда не находит нужную функцию по своему вкусу, ее переписывание не должно занять слишком много времени.

Тем не менее когда ваш подход подразумевает использование нескольких языков и технологических стеков, проблемы с внесением изменений в осиротевший сервис могут усугубиться, если ваша команда не знакома с технологическим стеком.

Тематическое исследование: realestate.com.au

Для первого издания книги я потратил некоторое время на общение с компанией realestate.com.au (REA) об использовании микросервисов, и многое из того, что я узнал, очень помогло в плане обмена реальными примерами микросервисов в действии. Я также нашел взаимодействие организационной структуры и архитектуры REA особенно увлекательным. Текущий обзор организационной структуры компании основан на наших обсуждениях еще 2014 года.

Я уверен, что сегодня REA выглядит совершенно по-другому. Обзор же представляет собой моментальный снимок, момент времени. Я не утверждаю, что это лучший способ структурирования организации, — просто это то, что лучше всего работало для REA в тот период времени. Учиться у других организаций — разумно, а бездумно копировать то, что они делают, — глупо.

Как и сегодня, основной бизнес REA в сфере недвижимости охватывал различные аспекты. В 2014 году REA была разделена на отраслевые бизнес-приложения (lines of business, LOB). Например, одно направление занималось жилой недвижимостью в Австралии, другое — коммерческой недвижимостью, а еще одно отделение занималось одним из зарубежных предприятий REA. С этими направлениями бизнеса были связаны команды (или отряды) по доставке ИТ-решений. Только на некоторых направлениях работало по одному отряду, в то время как у самого крупного бизнес-направления их было четыре. Таким образом, для отделения жилой недвижимости существовало несколько команд, участвовавших в создании веб-сайта и услуг по размещению объявлений, чтобы люди могли просматривать недвижимость. Люди время от времени менялись между этими командами, но, как правило, оставались в рамках одной сферы деятельности в течение длительных периодов времени, гарантируя, что члены команды смогут повысить свою осведомленность в данной области бизнеса. Это, в свою очередь, способствовало коммуникации между различными заинтересованными сторонами бизнеса и командой, предоставляющей для них функции.

Ожидалось, что каждая команда внутри направления деятельности будет владеть всем жизненным циклом каждого созданного ею сервиса, включая сборку, тестирование, выпуск, поддержку и даже вывод из эксплуатации. Основная команда службы доставки выполняла работу по предоставлению консультаций, указаний и инструментария подразделениям в LOB, помогая этим подразделениям выполнять работу более эффективно. Используя нашу новую терминологию, основная команда службы доставки играла роль команды поддержки. Сильная культура автоматизации была ключевой, и в компании REA активно использовали AWS, чтобы команды были более автономными. Рисунок 15.6 иллюстрирует, как все это работало.

Не только организация доставки была привязана к тому, как функционировал бизнес. Эта модель распространилась и на архитектуру. Одним из таких примеров были методы интеграции. В рамках НД все сервисы могли свободно общаться друг с другом любым способом, который они считали нужным, по решению отрядов, выступающих в качестве их хранителей. Но связь между НД должна была осуществляться асинхронным пакетным способом, что было одним из немногих незыблемых правил очень маленькой архитектурной команды. Эта детализированная коммуникация соответствовала той, которая существовала между различными частями бизнеса. Настаивая на пакетном режиме, каждое НД получило большую свободу в своих действиях и способах управления. Компания могла позволить себе сворачивать свои сервисы по желанию, зная, что до тех пор, пока она сможет обеспечить пакетную интеграцию с другими подразделениями бизнеса и заинтересованными сторонами, никому не будет до этого дела.

Такая структура обеспечивала значительную автономию не только среди команд, но и среди различных отделений бизнеса, а способность вносить

изменения помогла компании добиться значительного успеха на местном рынке. Эта более автономная структура также помогла фирме вырасти с нескольких сервисов в 2010 году до сотен к 2014 году, что дало возможность быстрее внедрять изменения.

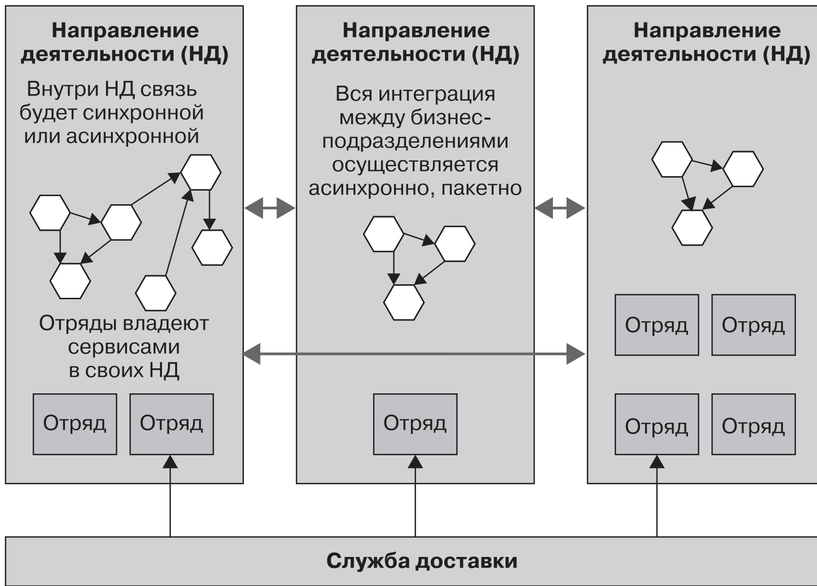


Рис. 15.6. Обзор организационной и командной структуры компании realestate.com.au и ее соответствие архитектуре

Те организации, которые достаточно адаптивны, чтобы изменять не только архитектуру своей системы, но и свою организационную структуру, могут извлечь из этого примера огромные преимущества с точки зрения повышения автономии команд и более быстрого вывода на рынок новых функций. REA — всего лишь одна из многих организаций, которые осознали, что системная архитектура не существует в вакууме.

Географическое распределение

Размещенным вместе командам синхронная коммуникация покажется очень простой, тем более что они обычно находятся в одном и том же месте в одно и то же время. Если ваша команда распределена, синхронная связь, скорее всего, более сложная, но она все равно достижима, если члены команды находятся в одинаковых или близких часовых поясах. При общении с людьми,

находящимися в разных часовых поясах, стоимость координации может резко возрасти. Однажды я занимал должность архитектора, помогая командам поддержки, базирующимся в Индии, Великобритании, Бразилии и США, в их вечер пятницы, а для меня утро субботы. Организовать встречу между мной и руководителями различных команд было невероятно сложно. Это означало, что мы проводили эти встречи нечасто (обычно ежемесячно) и нам приходилось следить за тем, чтобы во время этих сессий обсуждались только самые важные вопросы, поскольку часто более половины участников подключались в нерабочее время.

Вне этих сессий у нас поддерживалась асинхронная коммуникация, в основном по электронной почте, по другим, менее критическим по времени вопросам. Но по причине моего нахождения в Австралии задержка в этой форме общения была значительной. Я просыпался в понедельник утром, довольно спокойно начиная неделю, поскольку большая часть мира еще не проснулась. Это давало мне время обработать электронные письма, поступавшие от команд из Великобритании, Бразилии и США в их пятничный день, а для меня субботнее утро.

Вспоминается один проект, над которым я работал. В нем владение одним микросервисом было разделено между двумя географическими точками. В конце концов каждое подразделение начало специализироваться на выполняемой работе. Это позволило ему стать владельцем части кодовой базы, в рамках которой могла бы снизиться стоимость изменений. Затем команды получили более детальное представление о том, как эти две части взаимосвязаны. По сути, пути коммуникации, ставшие возможными в рамках организационной структуры, соответствовали детализированному API, сформировавшему границу между двумя половинами кодовой базы.

Итак, к чему это приводит нас при рассмотрении возможности развития нашего собственного проекта сервиса? Что ж, я бы предположил, что географические границы между людьми, вовлеченными в разработку, должны быть важным фактором при определении границ как команды, так и ПО. Гораздо легче сформировать единую команду, когда ее члены находятся в одном месте. Если совместное размещение невозможно и вы хотите сформировать распределенную команду, то нахождение членов команды в одних и тех же или близких часовых поясах поможет общению внутри данной команды, поскольку уменьшит потребность в асинхронном общении.

Возможно, ваша организация решит, что она хочет увеличить количество людей, работающих над вашим проектом, открыв офис в другой стране. На этом этапе вам следует серьезно подумать о том, какие части вашей системы можно переместить. Возможно, именно это определит, какую функциональность стоит отделить следующей.

На текущем этапе также стоит отметить, что, по крайней мере основываясь на наблюдениях авторов отчета «Исследование двойственности между архитектурами продукта и организации», на который я ссылался ранее, если организация, создающая систему, слабо связана (например, она состоит из географически

распределенных команд), то создаваемые системы, как правило, будут более модульными и, следовательно, менее связанными. Тенденцию, когда одна команда, владеющая множеством сервисов, склоняется к более тесной интеграции очень трудно поддерживать в более распределенной организации.

Закон Конвея наоборот

До сих пор мы говорили о том, как организация влияет на проект системы. Но как насчет обратного? А именно, может ли системный проект изменить организацию? Хотя я не смог найти доказательств в поддержку того, что закон Конвея работает в обратном направлении, как ни странно, я это видел.

Вероятно, лучшим примером послужит клиент, с которым я работал много лет назад. В те дни, когда технология Веб только зарождалась, а Интернет воспринимался как нечто, что доставлялось на дискете AOL по почте, эта компания была крупной типографией со скромным веб-сайтом. У них был веб-сайт, потому что так было нужно, но, по большому счету, он был не особо важен для работы. Когда создавалась оригинальная система, было принято довольно произвольное техническое решение относительно того, как она должна работать.

Система наполнялась контентом несколькими способами, но большая его часть поступала от третьих лиц, размещавших рекламу для просмотра широкой публикой. Существовала система ввода, позволявшая за деньги создавать контент, центральная система, которая брала эти данные и дополняла их различными способами, и система вывода, создававшая конечный веб-сайт для всеобщего обозрения.

Были ли первоначальные проектные решения правильными в то время — пусть решают историки, но за много лет компания сильно изменилась, и я, и многие мои коллеги начали задаваться вопросом, соответствует ли проект системы нынешнему состоянию компании. Ее физический печатный бизнес значительно сократился, и в отчетах и, следовательно, в деловых операциях организации теперь доминировало ее присутствие в Интернете.

В то время мы увидели организацию, тесно связанную с этой трехкомпонентной системой. Три канала или подразделения в ИТ-части бизнеса соответствовали каждой части системы: входной, основной и выходной. В рамках этих каналов существовали отдельные команды доставки. В то время я не понимал, что эти организационные структуры не предшествовали созданию системы, а фактически выросли вокруг нее. По мере того как печатная часть бизнеса сокращалась, а цифровая — росла, системный проект непреднамеренно проложил путь к росту организации.

В конце концов мы поняли, что, какими бы ни были недостатки системной разработки, нам придется внести изменения в организационную структуру, чтобы добиться сдвига. Сейчас компания сильно изменилась, но это происходило в течение многих лет.

Люди

> Неважно, как это выглядит на первый взгляд,
> но проблема всегда в людях.
>
> *Джерри Вайнберг, Второй закон консалтинга*

Мы должны признать, что в микросервисной среде разработчикам сложнее думать о написании кода. Они должны быть более осведомлены о последствиях таких вещей, как вызовы через границы сети или последствия сбоев. Мы также говорили о способности микросервисов облегчать опробование новых технологий, от хранилищ данных до языков. Но если вы переходите из мира монолитной системы, где большинству ваших инженеров приходилось использовать только один язык и они совершенно не обращали внимания на операционные проблемы, то переход в мир микросервисов для них может оказаться резким пробуждением.

Аналогичным образом навязывание полномочий командам для повышения автономии может быть чревато. Люди, которые в прошлом перекладывали работу на кого-то другого, привыкли, что есть на кого свалить вину, и могут чувствовать себя некомфортно, будучи полностью ответственными за свою работу. Вы даже можете столкнуться с тем, что ваши разработчики не будут носить пейджеры для получения оповещений систем, которые они поддерживают! Эти изменения могут происходить постепенно, и на начальном этапе имеет смысл переложить ответственность на тех людей, которые больше всего этого хотят.

Хотя книга в основном посвящена технологиям, люди — это не просто побочный продукт. Это те, кто создал то, что у вас есть сейчас, и создадут то, что будет в будущем. Планирование того, как все должно быть сделано, без учета мнения ваших сотрудников и их навыков, скорее всего, приведет к плохому результату.

У каждой организации своя собственная установка динамики вокруг данной темы. Оцените стремление ваших сотрудников к переменам. Не давите на них! Возможно, вы способны выделить отдельную команду для фронтальной поддержки или развертывания на короткий период, давая своим разработчикам время приспособиться к новым методам работы. Однако вам придется признать, что организации нужны разные люди, чтобы все это работало. Скорее всего, вам потребуется изменить способ найма — иногда проще показать возможности новой системы путем привлечения новых людей со стороны, у которых уже есть желаемый опыт работы. Грамотно подобранные новые сотрудники могут наглядно показать другим, что это возможно.

Каким бы ни был ваш подход, вам нужно четко формулировать обязанности своих сотрудников в мире микросервисов, а также доходчиво объяснить, почему эти обязанности важны для вас. Это поможет вам увидеть, какие у вас пробелы в знаниях и как их устранить. Для многих это будет довольно непростой путь. Просто помните, что без людей в команде любое желаемое изменение может быть обречено с самого начала.

Резюме

Закон Конвея подчеркивает опасность попыток навязать системный проект, который не соответствует организации. Это указывает нам, по крайней мере в сфере микросервисов, на модель, в которой сильное владение микросервисами является нормой. Совместное использование микросервисов или попытки практиковать коллективное владение в крупном масштабе организации часто могут привести к тому, что мы подорвем преимущества микросервисов.

Когда организация и архитектура не согласованы, мы получаем точки напряжения, о которых говорилось на протяжении всей главы. Осознав связь между этими двумя понятиями, мы гарантируем, что создаваемая система подходит организации, для которой мы ее создаем.

Если вы хотите продолжить изучение этой темы, в дополнение к ранее упомянутым топологиям команд я также настоятельно рекомендую доклад Джеймса Льюиса «Масштабирование, микросервисы и поток»[1], из которого я почерпнул много идей, способствовавших формированию этой главы. Его стоит посмотреть, если вы заинтересованы в более глубоком изучении некоторых изложенных в данной главе идей.

В следующей главе мы более подробно рассмотрим тему, которую я уже затрагивал, — роль архитектора.

---

[1] *Lewis J.* Scale, Microservices and Flow // YOW! Конференции, 10 февраля 2020 года, видео на YouTube, 51:03. https://oreil.ly/ON81J.

ГЛАВА 16

Эволюционный архитектор

Мы уже узнали, что микросервисы предоставляют большой выбор и, соответственно, множество решений для принятия. Например, сколько различных технологий необходимо применять, должны ли мы позволить разным командам использовать разные идиомы программирования и стоит ли разделить или объединить микросервис? Как мы будем принимать эти решения? С очень быстрыми темпами изменений и более гибкой средой, которую допускают эти архитектуры, роль архитектора также должна измениться. В этой главе я достаточно объективно взгляну на то, в чем заключается роль архитектора, и, надеюсь, начну последний штурм башни из слоновой кости.

Что в имени твоем?

> Вы продолжаете употреблять это слово. Я не думаю, что оно означает то, что вы думаете.
>
> *Иниго Монтойя, из к/ф «Принцессы-невесты»*

У архитекторов важная работа. Они отвечают за то, чтобы система представляла единое техническое видение, помогающее поставлять ПО, необходимое клиентам. В одних ситуациях им, возможно, придется работать только с одной командой, и в этом случае роль архитектора и технического руководителя часто совпадает. В других ситуациях они могут определять видение всей программы работы, координировать работу с несколькими командами по всему миру или, возможно, даже с целой организацией. На каком бы уровне ни работали архитекторы, их роль сложно определить, и, несмотря на то что она часто становится очевидным продвижением по карьерной лестнице для разработчиков в корпоративных организациях, это также должность, подвергающаяся большей критике, чем практически любая другая в нашей области. Как никто другой архитекторы оказывают непосредственное влияние на качество создаваемых

систем, на условия труда своих коллег и на способность организации реагировать на изменения. И все же их роль кажется очень плохо понятной. Почему?

Наша отрасль — молодая. Иногда мы, кажется, забываем, что создаем программы, работающие на том, что мы называем компьютерами, всего 75 лет или около того. Наша профессия не вписывается в красивую аккуратную рамку, понятную обществу в целом. Мы не похожи на электриков, сантехников, врачей или инженеров. Сколько раз вы рассказывали кому-нибудь на вечеринке, чем занимаетесь, только для того, чтобы разговор прекратился? Мир в целом с трудом понимает, что такое разработка ПО. Как я неоднократно подчеркивал на протяжении всей книги, мы часто, кажется, и сами этого не понимаем.

Поэтому мы заимствуем понятия у других профессий. Мы называем себя инженерами или архитекторами ПО. Но мы не архитекторы и не инженеры в том смысле, в каком общество понимает эти профессии. Архитекторы и инженеры обладают строгостью и дисциплиной, о которых мы можем только мечтать, и их важность в обществе хорошо понимается. Я помню разговор со своим другом за день до того, как он стал квалифицированным архитектором. «Завтра, — сказал он, — если я в пабе дам тебе совет, как что-то построить, а он окажется неправильным, меня привлекут к ответственности. На меня могут подать в суд, так как в глазах закона я теперь квалифицированный архитектор и должен нести ответственность за свои ошибки». Важность этих профессий для общества означает, что люди должны соответствовать требованиям к своей квалификации. В Великобритании, например, необходимо минимум семилетнее обучение, прежде чем вы сможете называться архитектором. Но эти профессии также основаны на совокупности знаний, уходящих корнями в тысячелетия. А профессия архитектора ПО? Про нее нельзя сказать то же самое. Отчасти поэтому я считаю многие формы сертификации в сфере ИТ бесполезными, поскольку мы так мало знаем о том, что такое «хорошо».

Я говорю это не для того, чтобы принизить термин «программная инженерия»[1], введенный в обиход еще в 1960-х годах Маргарет Гамильтон, но он был столь же вдохновляющим, сколь и относящимся к текущей реальности. Этот термин возник как призыв к повышению качества создаваемого ПО и в знак признания того факта, что программные проекты часто терпели неудачу, но все чаще использовались в жизненно важных областях, связанных с выполнением особых миссий и обеспечением безопасности. С тех пор была ситуация значительно улучшилась, но, отработав 20 лет в этой отрасли, я считаю, что нам еще многому предстоит научиться, чтобы делать работу хорошо (или по крайней мере получше).

Часть из нас хочет признания, поэтому мы заимствуем названия из других известных профессий. Но если мы позаимствуем *методы работы* у этих профессий, не понимая, что за ними стоит, или не принимая во внимание, чем разработка ПО отличается, скажем, от гражданского инженерного проектирования, то это

[1] По нескольким причинам, не последней из которых стало то, что у меня есть степень в данной области...

может вылиться в проблемы. Ничто из этого не должно восприниматься как аргумент в пользу того, что мы не должны стремиться к большей строгости в своей работе: мы не можем просто заимствовать откуда-то идеи и предполагать, что они сработают. Сфера очень молода, и проблема в том, что у нас гораздо меньше абсолютных ценностей, с которыми согласны специалисты отрасли в целом.

Общепринятое понимание термина «архитектор» в ИТ-отрасли вносит свою путаницу: считается, что архитектор составляет проект, а работники должны правильно его интерпретировать и реализовать; архитектор одновременно и художник, и инженер, который контролирует создание того, что является пока только идеей, не считаясь при этом с другими точками зрения, принимая во внимание лишь редкие возражения инженера-строителя относительно законов физики. В нашей отрасли такой взгляд на фигуру архитектора приводит к некоторым ужасным практикам, когда архитекторы создают диаграмму за диаграммой, страницу за страницей документации с целью построения совершенной системы, не принимая во внимание принципиально непознаваемое будущее и совершенно не понимая, насколько трудно будет реализовать их планы, будут ли они на самом деле работать и будут ли они меняться с получением нами новых знаний.

Но архитекторы строительной среды работают не в такой, как архитекторы ПО, сфере. Их ограничения различны, конечный продукт различается. Стоимость изменений при проектировании намного выше, чем при разработке ПО. Удалить бетон не то же самое, что изменить код. И даже инфраструктура, на которой запускается код, может стать более гибкой, чем раньше, благодаря виртуализации. Здания достаточно прочны после постройки. Да, их можно менять, расширять или сносить, но связанные с этим затраты очень высоки. Программное обеспечение же будет постоянно меняться в соответствии с новыми потребностями.

Итак, если архитектура ПО отличается от архитектуры строительной среды, возможно, нам следует немного прояснить, что такое архитектура ПО на самом деле.

Что такое архитектура ПО

Одно из самых известных определений архитектуры ПО пришло по электронной почте от Ральфа Джонсона: «Архитектура — это самое важное. Что бы это ни было»[1]. Значит ли это, что архитектор делает что-то важное? А вся остальная выполняемая работа при этом неважна? Проблема данной цитаты в том, что она

---

[1] Взято из переписки в рассылке экстремального программирования, которой Мартин Фаулер затем поделился в своей статье «Кому нужен архитектор?» (https://oreil.ly/6C0cI).

часто используется изолированно, без какого-либо понимания контекста. Во-первых, ясно, что он говорит с точки зрения разработчика ПО. Далее он говорит:

> Поэтому более точным определением будет следующее: «В большинстве успешных программных проектов у опытных разработчиков проекта складывается общее понимание проекта системы. Это общее понимание называется “архитектурой”. Оно включает в себя то, как система разделена на компоненты и как они взаимодействуют через интерфейсы. Эти компоненты обычно состоят из более мелких звеньев, но архитектура включает в себя только те части и интерфейсы, которые понятны всем разработчикам».
>
> Это было бы лучшим определением, так как оно проясняет, что архитектура — это социальная конструкция (ну, ПО тоже, но архитектура еще более важна), потому что это зависит не только от ПО, но и от того, какая часть ПО считается важной в результате достижения группового консенсуса.

Здесь Ральф использует термин «компоненты» в его самом общем смысле. В контексте книги можно думать о компонентах как о микросервисах и, возможно, о модулях внутри этих микросервисов.

Архитектура ПО — это форма системы. Она возникает по замыслу или случайно. Мы принимаем ряд определенных решений и в конечном счете получаем результаты. Не думая о проекте с точки зрения архитектуры, мы тем не менее получаем архитектуру. Она иногда может быть тем, что возникает, пока мы заняты составлением других планов.

Преданный своему делу архитектор — это человек, который видит и понимает всю эту систему в целом, понимает силы, действующие на нее. Он должен обеспечить понятное и соответствующее цели видение архитектуры, удовлетворяющее потребностям системы и ее пользователей, а также людей, работающих над ней. Рассмотрение только одного аспекта, например логического, но не физического, ограничивает эффективность архитектора. Если вы согласны с тем, что архитектура — это понимание системы, то сужение круга волнующих вас вопросов ограничивает и вашу способность рассуждать и вносить изменения.

Архитектура может быть невидимой для людей, работающих с ней. Она может быть настолько незаметной, словно ее не существует. Быть чем-то, что направляет и помогает достичь необходимого результата. Она может быть удушающей и подавляющей. Она способна восхищать, не давая о себе знать, и выбивать из вас дух без какого-либо злого умысла. Так что, независимо от того, является ли архитектура «самой важной частью» или нет, она, безусловно, *важна*.

Еще одна емкая цитата, часто используемая для определения архитектуры ПО, взята из той же статьи, где Мартин разделяет взгляды Ральфа: «Таким образом, вы можете определить архитектуру как *что-то, что люди воспринимают как трудноизменяемое*». Идея Мартина о том, что архитектура — это нечто трудноизменяемое, имеет определенный смысл и возвращает нас к концепции архитектуры в построенной среде. Там, где что-то изменить сложнее,

архитектура требует более тщательного обдумывания, чтобы действительно убедиться, что движение происходит в правильном направлении. Но есть проблема с тем, чтобы взять простое определение сложной идеи и использовать его в качестве рабочего понятия — если бы это утверждение полностью соответствовало вашим представлениям об архитектуре ПО, вы бы многое упустили. Да, большая часть архитектуры ПО связана с размышлениями о вещах, которые трудно изменить, но это также и создание пространства, позволяющего модифицировать проект.

Делая изменения возможными

Возвращаясь к теме строительства, а не программных систем: архитектор Мис ван дер Роэ, возможно, сделал больше для создания того, что мы сейчас называем современным небоскребом, чем любой другой архитектор, — его знаменитое здание Сигрем-билдинг стало основой для последующих подобных сооружений. Сигрем-билдинг отличалось от многих строений, созданных раньше. Внешние стены здания неструктурны — они окружают стальной наружный каркас. Основные коммуникации и узлы здания — лифты, лестницы, вентиляция, водоснабжение и канализация, а также электрическая система — проходят через бетонную сердцевину. Посмотрите, как сегодня строится современная высотка: именно эта сердцевина возводится первой, а на вершине часто можно увидеть гигантский кран. Ни один этаж Сигрем-билдинг не получил внутренних перегородок — это означает, что предусматривается полная гибкость в плане использования площадей. Можно изменить конфигурацию пространства по своему усмотрению, проложив электрическую проводку и кондиционирование воздуха в разные части любого этажа через подвесные потолки и воздуховоды в самом полу.

Интересно отметить, что Сигрем-билдинг разработано с использованием процесса, в ходе которого проект здания менялся по мере строительства. Где мы видели эту идею раньше?

Смысл данного проекта — создать что-то, что Мис ван дер Роэ назвал универсальным пространством: большой однопролетный объем, который можно было бы реконфигурировать в соответствии с различными потребностями. Целевое назначение зданий меняется, поэтому идея состояла в том, чтобы создать пространство, максимально гибкое с точки зрения того, как его можно использовать. Таким образом, Мис ван дер Роэ не только должен был сосредоточиться на фундаментальной эстетике здания, найдя площади для базовых коммуникаций, которые было бы трудно или даже невозможно изменить позже, но он также должен был обеспечить возможность использовать здание иначе, чем первоначально предполагалось. Вскоре мы рассмотрим, как перенести подобный подход на микросервисную архитектуру.

Эволюционное видение для архитектора

Требования к архитекторам ПО меняются быстрее, чем к людям, проектирующим и возводящим здания. Так же меняются инструменты и методы, имеющиеся в нашем распоряжении. То, что мы создаем, не становится фиксированными точками во времени. После запуска в эксплуатацию наше ПО будет продолжать развиваться по мере изменения способов его использования. Необходимо признать, что как только ПО попадет в руки пользователей, нам придется реагировать и адаптироваться, а не ожидать, что получилось создать неизменный артефакт. Таким образом, архитекторы ПО должны отказаться от создания идеального конечного продукта и сосредоточиться на помощи в создании *основы* (фреймворка), в которой системы могут появляться и развиваться по мере появления новых знаний.

Хотя я и планировал не проводить сравнений с другими профессиями и проектами, есть аналогия, которая мне нравится, когда речь заходит о роли ИТ-архитектора, и которая, как мне кажется, отлично сюда вписывается. Эрик Дерненбург из компании Thoughtworks впервые поделился со мной мыслью, что необходимо думать об архитекторе скорее как о градостроителе, чем как об архитекторе-строителе. Роль градостроителя должна быть знакома любому из вас, кто играл в SimCity или Cities: Skylines. Градостроитель изучает множество источников информации, а затем пытается оптимизировать планировку города, чтобы она наилучшим образом соответствовала потребностям современных жителей, в то же время принимая во внимание будущее использование. Однако интересно то, как градостроители влияют на развитие города. Они не говорят: «Постройте это конкретное здание там», вместо этого они определяют зоны, позволяющие принимать решения на местном уровне в рамках определенных ограничений. Так, как и в SimCity, можно обозначить часть вашего города как промышленную зону, а другую часть — как жилую. Затем уже другие люди решают, какие здания будут построены, но есть ограничения: если вы хотите построить завод, он должен располагаться в промышленной зоне. Вместо того чтобы слишком много беспокоиться о происходящем в одной зоне, градостроитель будет уделять гораздо больше времени тому, как люди и коммунальные службы перемещаются из одной зоны в другую.

Множество людей сравнивало город с живым организмом. Город меняется с течением времени. Он эволюционирует под действием его обитателей или внешних сил. Градостроитель делает все возможное, чтобы предвидеть эти изменения, но признает, что пытаться напрямую контролировать все аспекты происходящего бесполезно. Таким образом, наши архитекторы, словно градостроители, должны задавать направление в общих чертах и лишь в ограниченных случаях заниматься конкретными деталями реализации. Они должны убедиться, что система соответствует текущим требованиям, но также имеет задел на будущее.

Сравнение с программным обеспечением должно быть очевидным. Поскольку пользователи работают на нашем ПО, необходимо реагировать и меняться. Невозможно предвидеть все, и поэтому вместо того, чтобы предугадывать всевозможные события, мы должны планировать свою работу так, чтобы допускать изменения, до последнего избегая стремления чрезмерно конкретизировать каждый момент. Наш город — система — должен быть хорошим, счастливым местом для всех, кто им пользуется.

Определение границ системы

В очередной раз затрагивая метафору архитектора как градостроителя, каковы наши зоны? Это границы микросервисов или, возможно, обобщенные группы микросервисов. Как архитекторам, нам нужно гораздо меньше беспокоиться о происходящем *внутри* зоны и больше о том, что происходит *между* зонами. Это означает, что необходимо потратить время на размышления о взаимодействии микросервисов друг с другом и убедиться, что есть возможность должным образом отслеживать общее состояние нашей системы. С точки зрения архитектуры именно так мы создаем наше собственное *универсальное пространство* — определяя некоторые конкретные границы, показываем нашим коллегам, создающим систему, те области, где изменения могут быть внесены более свободно, не нарушая какой-либо фундаментальный аспект архитектуры системы.

Рассмотрим простой пример. На рис. 16.1 показано, как микросервис Рекомендации получает доступ к информации из микросервисов Акции и Продажи. Мы уже всесторонне рассмотрели возможность преобразовать функции, скрытые внутри этих трех микросервисов, не беспокоясь о поломке всей системы, — можно по желанию внести изменения в сервис Продажи или Акции, пока в сервисе Рекомендации содержится информация о том, как он будет взаимодействовать с этими нижестоящими микросервисами.

Мы можем создать пространство для изменений и на более масштабных уровнях. На рис. 16.2 показано, что микросервисы с рис. 16.1 фактически существуют в маркетинговой зоне, которая связана с ответственностью конкретной команды. Мы определили ожидаемое поведение с точки зрения того, как маркетинговая функциональность взаимодействует с более крупной системой. Внутри маркетинговой зоны можно вносить любые желаемые преобразования при условии сохранения совместимости с остальной системой. Возвращаясь к идее трудноизменяемых вещей, организационные структуры часто попадают в эту категорию, и поэтому существующие структуры команд могут помочь определить эти зоны. Координировать изменения внутри команды между микросервисами, принадлежащими данной команде, будет проще, чем изменять виды коммуникаций, доступные другим командам.

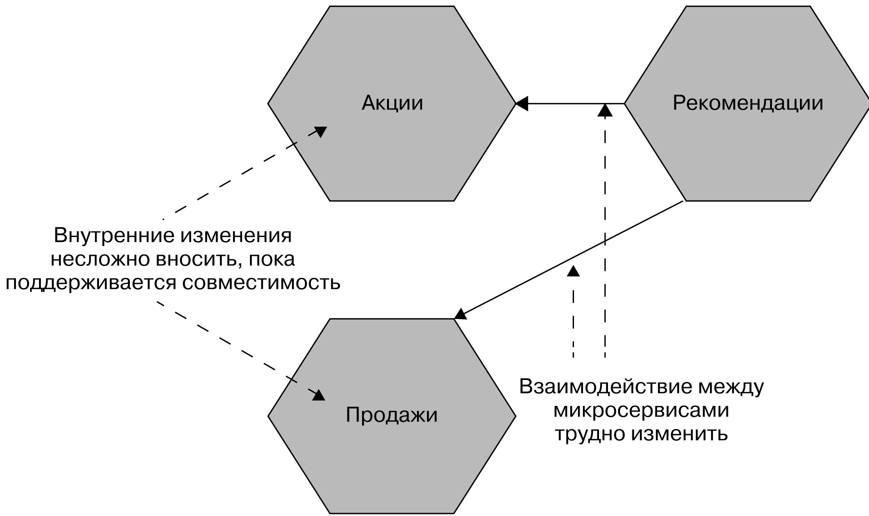


Рис. 16.1. Изменения внутри границ микросервиса легко вносить, если взаимодействие между микросервисами не меняется

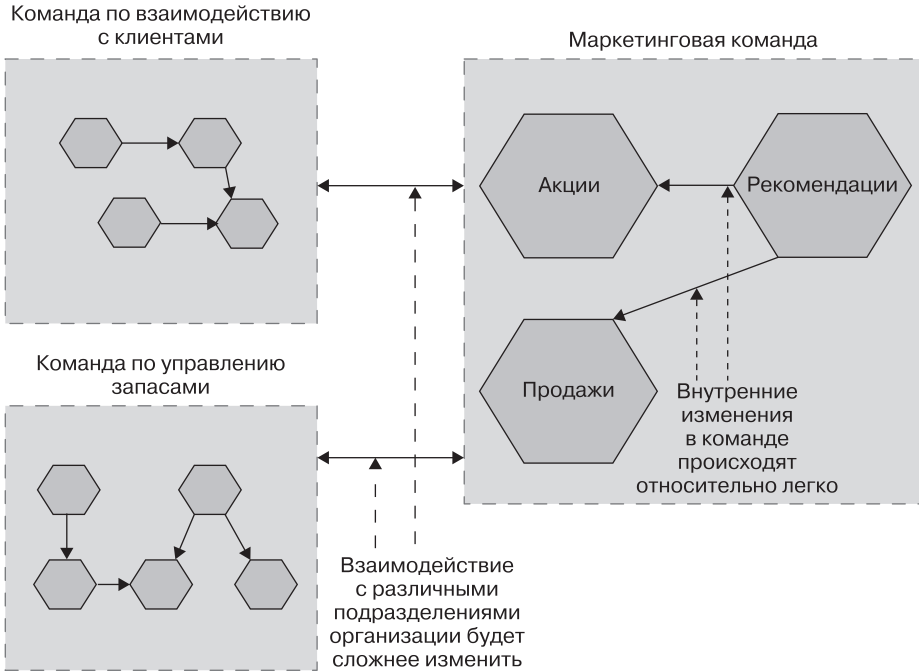


Рис. 16.2. Изменения внутри зоны вносить легче, чем изменения между зонами

Это хорошо перекликается с концепцией команды API, которую мы обсуждали в разделе «Маленькие команды, большая организация» главы 15. Архитектор способен облегчить создание команды API, убедившись, что микросервисы команды и методы работы соответствуют более масштабной организации.

Определяя области, в которых эти преобразования могут быть внесены без ущерба для системы в целом, мы облегчаем жизнь разработчикам, а также фокусируем свое внимание на более сложных частях системы. Помните концепцию скрытия информации, рассмотренную в главе 2? Как мы уже выяснили, скрытие информации внутри границ микросервиса значительно упрощает создание стабильного интерфейса для потребителей. Когда микросервис модифицируется, легче убедиться, что совместимость с внешними потребителями не нарушена. Здесь можно определить архитектуру, обеспечивающую скрытие информации на уровне команды, а не только на уровне микросервиса. Это дает нам еще один уровень скрытия информации и создает большее безопасное пространство. В нем команда способна вносить локальные изменения, не нарушая иные части системы.

В пределах каждого микросервиса или более крупной зоны можно принять то, что команда, владеющая данной зоной, выберет другой технологический стек или хранилище данных. Конечно, здесь могут возникнуть и другие проблемы. Ваша склонность позволить командам выбирать инструмент для работы может омрачаться тем, что становится сложнее нанимать людей или перемещать их между командами, если у вас есть десять различных поддерживаемых технологических стеков. Аналогичным образом, если каждая команда выберет отличное от других хранилище данных, вам может не хватить опыта для масштабного запуска любого из них. Netflix, например, в основном стандартизировал Cassandra как технологию хранения данных. Хотя это, вероятно, не лучшим образом подходит для всех случаев, Netflix считает, что ценность, полученная за счет создания инструментария и экспертных знаний на основе Cassandra, важнее, чем необходимость поддерживать и масштабировать множество других платформ, даже если они лучше подходят для определенных задач. Netflix — это экстремальный пример, где масштаб, вероятно, стал самым сильным преобладающим фактором, но вы поняли идею.

Однако между микросервисами все может запутаться. Если один сервис решит предоставить REST через HTTP, другой использует gRPC, а третий — Java RMI, то интеграция может стать кошмаром, поскольку потребляющие микросервисы должны понимать и поддерживать несколько стилей обмена. Вот почему я стараюсь придерживаться принципа, согласно которому мы должны «беспокоиться о том, что происходит между блоками, и быть либеральными в том, что происходит внутри».

Таким образом, успешная архитектура в такой же степени позволяет вносить изменения в соответствии с потребностями наших пользователей, как и все остальное. Но люди часто забывают одну вещь: наша система предназначена не только для пользователей, но и для людей, которые сами создают ПО. Успешная архитектура также помогает создать приятную среду для выполнения нашей работы.

Социальная конструкция

Ни один план не выдерживает контакта с врагом.

Гельмут фон Мольтке
(сильно перефразированный)

Итак, вы подумали о видении, об ограничениях и о целях. Вам кажется, что вы понимаете, что будет трудно изменить, и те области, где вам нужно сделать изменения возможными. Что теперь? Что ж, архитектура — это то, что уже происходит, а не то, что, по вашему мнению, должно произойти. В этом заключается разница между видением и реальностью. Архитекторам сферы строительства необходимо работать с людьми, возводящими здания, чтобы помочь им понять, в чем заключается видение, но также и изменить план, когда реальность бросает вызов этому видению. Вполне вероятно, что то, что вы считаете возможным, в корне таковым не является. Если архитектор в какой-то степени не связан с разработчиками системы, то он не сможет помочь донести видение до людей, выполняющих работу, и не поймет, где это видение больше не соответствует цели. Строительная бригада может столкнуться на месте с трудностями, которые не были предусмотрены, или, возможно, нехватка поставок приведет к переосмыслению проекта.

Архитектура — это то, что происходит, а не то, что планируется. Если как архитектор вы отстраняете себя от процесса воплощения этого видения в жизнь, то вы не архитектор, а мечтатель. Архитектура, которая появится, не обязательно может иметь отношение к тому, что вы хотите. Это произойдет с вами или без вас. Внедрение архитектуры требует работы многих людей и множества решений, больших и малых.

Как выразился Грейди Буч[1]:

> В начале процесса создания ПО архитектура программно-интенсивной системы представляет собой изложение видения. В конце архитектура каждой такой системы становится отражением миллиардов маленьких и больших, преднамеренных и случайных проектных решений, принятых на этом пути.

Это означает, что, даже если у вас есть преданный своему делу человек, который в конечном счете отвечает за архитектуру, существует много людей, ответственных за воплощение этого видения на практике. Внедрение успешной архитектуры — это командная работа. Вернемся к предыдущей цитате Ральфа Джонсона, «архитектура — это социальная конструкция». Отличным примером может служить компания Comcast, которая поделилась своим опытом децентра-

[1] Grady Booch (@Grady_Booch), Twitter. 4 сентября 2020 года, 5:12. https://oreil.ly/ZgPRZ.

лизации принятия решений с помощью гильдии архитекторов[1]. Учитывая свой масштаб, в Comcast решили использовать опыт отраслевых руководящих групп, где ключевой принцип — коллективное принятие решений.

> В компании Comcast мы поняли, что эта проблема очень похожа на то, как работают органы по разработке открытых стандартов: согласование технических подходов несколькими автономными группами. Мы разработали внутреннюю гильдию архитекторов, явно смоделированную по образцу Инженерного совета Интернета (Internet Engineering Task Force, IETF), определяющего многие важные интернет-протоколы.
>
> *Джон Мур, главный архитектор ПО Comcast Cable*

Подход компании Comcast отличается определенной формальностью, которая может показаться неуместной некоторым организациям, но, похоже, он хорошо работает для самой Comcast, учитывая ее размер и распространение.

Обитаемость

Еще одно понятие, пришедшее из строительной среды и получившее резонанс в области разработки ПО, — *обитаемость*. Впервые я узнал об этом термине от Фрэнка Бушмана. Он объяснил, что архитектор несет ответственность за то, чтобы создаваемая им среда была приятной для работы. Если архитектура — это каркас системы, описывающий, как трудноизменяемые элементы сочетаются друг с другом, то также бывают случаи, когда потребуется ввести ограничения. Однако, если все сделать неправильно, работа в системе может стать затруднительной и подверженной ошибкам.

Как объясняет Ричард Гэбриэл, автор книги Patterns of Software[2]:

> Обитаемость — это характеристика исходного кода, позволяющая программистам, приходящим к уже давно кем-то написанному коду, понимать его конструкцию и намерения и изменять его удобно и уверенно.

Однако современная экосистема разработки ПО состоит не только из кода, она распространяется и на применяемые технологии, и на используемые методы работы. Слишком часто я видел, как разработчики проклинают технологию, которую им навязывают, ведь обычно ее выбирают люди, которым никогда не приходилось ею пользоваться. Чем больше сотрудников будут участвовать в процессе эволюции вашей архитектуры и выбора используемых инструментов

---

[1] *Moore J.* Architecture with 800 of My Closest Friends: The Evolution of Comcast's Architecture Guild // InfoQ. 14 мая 2019 года. https://oreil.ly/aIvbi.

[2] *Gabriel R.* Patterns of Software: Tales from the Software Community. — Oxford University Press, 1996.

и технологий, тем легче вам будет гарантировать, что конечным результатом станет пригодная для жизни среда, в которой люди, создающие систему, будут чувствовать себя счастливыми и продуктивными в своей работе.

Если мы хотим, чтобы создаваемые системы были пригодны для работы наших разработчиков, то архитекторы и другие лица, принимающие решения, должны понимать влияние своих решений. Как минимум это означает проводить время с командой, а в идеале — писать код с командой. Для тех из вас, кто практикует парное программирование, становится простым делом присоединиться к команде на короткий период времени в качестве одного из членов пары. Участие в упражнениях по групповому программированию также может принести значительную пользу, хотя архитектор, принимающий участие в такой групповой деятельности, должен осознавать, как его присутствие может изменить динамику ансамбля.

В идеале вы должны работать над стандартными задачами, чтобы действительно понять, на что похожа «обычная» работа. Я не могу не подчеркнуть, насколько важно для архитектора действительно проводить время с командами, создающими систему! Это значительно эффективнее, чем позвонить или просто просмотреть их коды. Что касается периодичности, то во многом она зависит от размера команды (команд), с которой вы работаете. Но главное, что это должно быть обычным делом. Например, если вы работаете с четырьмя командами, возможно, вы будете проводить полдня с каждой из них каждые четыре недели, работая с ними над их задачами по доставке, чтобы повысить осведомленность и улучшить коммуникацию с этими командами.

Принципиальный подход

> Правила предназначены для послушания глупцов
> и руководства мудрецов.
>
> *Приписывается Дугласу Бейдеру*

Принятие решений при проектировании систем сводится к компромиссам, а микросервисные архитектуры дают множество возможностей для компромиссов! Выбирая хранилище данных, выберем ли мы платформу, с которой у нас меньше опыта работы, но которая обеспечивает лучшее масштабирование? Нормально ли поддерживать два разных технологических стека в системе? А как насчет трех? Некоторые решения могут быть приняты на месте на базе имеющейся информации, и это будет простейшим решением. Но как насчет тех решений, которые, возможно, придется принимать на основе неполной информации?

Здесь может помочь кадрирование, и отличный способ помочь разделить на этапы процесс принятия решений — это определить набор принципов и практик, которыми мы руководствуемся, исходя из поставленных целей. Давайте рассмотрим каждый из этих аспектов кадрирования по очереди.

Стратегические цели

Роль архитектора и так достаточно сложна, так что, к счастью, ему обычно не приходится определять стратегические цели! Они должны указывать на то, куда движется ваша компания, и на то, как, по ее мнению, лучше всего сделать своих клиентов счастливыми. Это будут цели высокого уровня, которые вообще не обязаны включать в себя технологическую сторону. Они могут быть определены на уровне компании или подразделения. Это могут быть такие задачи, как «Расширяйтесь в Юго-Восточную Азию, чтобы открыть новые рынки» или «Позвольте клиенту достичь как можно больших результатов, используя самообслуживание». Ключ в том, что они определяют, куда движется ваша организация, поэтому необходимо убедиться, что технология соответствует этому.

Если вы стали лицом, определяющим техническое видение компании, скорее всего, вам нужно будет уделять больше времени нетехническим подразделениям вашей организации (или бизнесу, как их часто называют). Что лежит в основе бизнес-подхода? И как оно меняется?

Принципы

Принципы — это установленные правила для того, чтобы привести текущую работу в соответствие с какой-то более крупной целью, и иногда они меняются. Например, если в качестве одной из стратегических целей организации было принято сокращение времени вывода на рынок новых функций, можно определить принцип, согласно которому команды доставки получают полный контроль на протяжении всего жизненного цикла своего ПО, чтобы отправлять его по готовности и независимо от любой другой команды. Если другая цель заключается в том, что ваша организация стремится активно расширять свои предложения в других странах, можно принять решение внедрить принцип, согласно которому вся система должна быть переносимой, чтобы ее можно было развернуть локально для обеспечения конфиденциальности данных.

Вероятно, вам не захочется, чтобы таких принципов было много. Около десяти принципов — это хорошее количество, достаточно маленькое, чтобы люди запомнили их или поместили на небольших плакатах. Чем больше у вас принципов, тем выше вероятность, что они пересекаются или противоречат друг другу.

«Двенадцать факторов» Героку (http://www.12factor.net) — это набор принципов проектирования, структурированных на основе цели, помогающей создавать

приложения, которые хорошо работают на платформе Heroku. Эти принципы могут также иметь смысл в других контекстах. Некоторые из них фактически представляют собой ограничения, основанные на поведении, которое должно демонстрировать ваше приложение, чтобы работать на Heroku. Ограничение — это действительно то, что очень трудно (или практически невозможно) изменить, в то время как принципы — это то, что было решено выбрать. Вы можете решить явно выделить принципы на фоне ограничений. Такой подход поможет указать те вещи, которые действительно невозможно изменить. Лично я думаю, что есть некая польза в хранении их в одном списке, — время от времени можно одобрять сложные ограничения и проверять, действительно ли они непоколебимы!

Методы

Методы — это то, как мы обеспечиваем выполнение своих принципов. Они представляют собой набор подробных практических рекомендаций по решению задач. Методы часто будут зависеть от конкретной технологии и должны быть достаточно низкоуровневыми, чтобы любой разработчик мог их понять. Они могут включать в себя рекомендации по программированию, например: «Все данные лога должны собираться централизованно» или «HTTP/REST принят стандартным стилем интеграции». В силу своей технической природы методы, как правило, меняются чаще, чем принципы.

Как и в случае с принципами, иногда методы отражают ограничения внутри организации. Например, если было решено выбрать Azure в качестве облачной платформы, это необходимо будет отразить в ваших методах.

Методы должны лежать в основе ваших принципов. Принцип, согласно которому группы доставки контролируют полный жизненный цикл своих систем, может означать, что у вас есть метод, согласно которому все микросервисы развертываются в изолированных учетных записях AWS, обеспечивая самообслуживание ресурсов и изоляцию от других команд.

Сочетание принципов и методов

Принципы одного человека — это методы другого. Например, можно назвать использование HTTP/REST принципом, а не методом. И это будет в порядке вещей. Ключевым моментом становится то, что есть ценность в наличии всеобъемлющих идей, определяющих развитие системы, и в наличии достаточного количества деталей, чтобы люди знали, как реализовать эти идеи. Для достаточно небольшой группы, возможно, одной команды, сочетание принципов и методов может быть приемлемым. Однако в более крупных организациях, где технологии и методы работы могут быть разнообразными, вам может потребоваться

определенные наборы методов для разных команд при условии, что все они соответствуют общему набору принципов. Например, у команды .NET может быть один набор методов, а у команды Java — другой. Принципы, однако, остаются одинаковыми для обеих.

Пример из жизни

Мой давний коллега, Эван Боттчер, разработал схему, показанную на рис. 16.3, в ходе работы с одним из своих клиентов. На рисунке изображено взаимодействие целей, принципов и практик в очень четком формате. В течение пары лет методы, расположенные справа, будут меняться довольно регулярно, в то время как принципы останутся довольно статичными. Подобную диаграмму можно красиво распечатать на одном листе бумаги и поделиться ею, а каждая идея достаточно проста, чтобы ее запомнил среднестатистический разработчик. Конечно, за каждым пунктом здесь стоит больше подробностей, но возможность сформулировать это в краткой форме очень полезна.

Рис. 16.3. Практический пример принципов и методов

Имеет смысл хранить документацию, поддерживающую некоторые из этих элементов, и еще лучше получить в свое распоряжение рабочий код, показывающий, как эти методы могут быть реализованы. В подразделе «Платформа» главы 15 мы рассмотрели, как создание общего набора инструментов облегчает разработчикам выполнение своих обязанностей, — в идеале платформа должна максимально упростить следование этим методам, и по мере изменения методов она должна меняться соответствующим образом.

Руководство эволюционной архитектурой

Итак, если наша архитектура не статична, а постоянно меняется и эволюционирует, как нам убедиться, что она растет и меняется как задумано, а не просто мутирует в какой-то неуправляемый гигантский сгусток проблем, трудностей и взаимных обвинений? В книге «Эволюционная архитектура»[1] авторы описывают функции приспособленности для сбора информации об относительной приспособленности архитектуры, чтобы помочь архитекторам решить, нужно ли им предпринимать действия. Из книги:

> Эволюционные вычисления включают в себя ряд механизмов, позволяющих решению постепенно появляться с помощью небольших изменений в каждом поколении ПО. При каждом создании решения инженер оценивает текущее состояние: будет ли оно ближе к конечной цели или дальше от нее? Например, при использовании генетического алгоритма для оптимизации конструкции крыла функция приспособленности оценивает сопротивление ветру, вес, воздушный поток и другие характеристики, желательные для хорошей конструкции крыла. Архитекторы определяют функцию приспособленности, чтобы объяснить, что лучше, и помочь понять, когда цель достигнута. В программном обеспечении функции приспособленности проверяют, чтобы разработчики сохраняли важные архитектурные характеристики.

Идея функции приспособленности заключается в том, что она используется для понимания текущего состояния некоторого важного свойства, так что если это свойство изменяется за пределами некоторых допустимых границ, то изменение необходимо изучить. Как правило, функции приспособленности используются для обеспечения построения архитектуры в соответствии с установленными принципами и ограничениями.

Если взять пример из книги «Эволюционная архитектура», рассмотрим требование, согласно которому ответ от определенного сервиса должен быть получен за 100 миллисекунд или меньше. Можно было бы реализовать функцию приспособленности для сбора данных о производительности из этого сервиса

[1] *Форд Н., Парсонс Р., Куа П.* Эволюционная архитектура. Поддержка непрерывных изменений. — Питер, 2019.

либо в среде тестирования производительности, либо из реальной запущенной системы, чтобы убедиться, что фактическое поведение системы соответствует требованиям. В книге «Эволюционная архитектура» гораздо более подробно рассматривается эта тема, и я настоятельно рекомендую ее, если вы хотите изучить эту концепцию глубже.

Функции приспособленности для архитектуры могут быть самых разных форм. Основная концепция, однако, заключается в том, что вы собираете реальные данные, чтобы понять, соответствует ли ваша архитектура этим критериям. Это может быть связано с производительностью системы, связанностью кода, временем цикла или множеством других аспектов. Эти функции приспособленности выступают в качестве еще одного источника информации, помогающего архитектору понять, где ему, возможно, потребуется принять участие. Обратите внимание, однако, что для меня фитнес-функции работают лучше всего в сочетании с тесным сотрудничеством с людьми, создающими систему. Функции приспособленности должны быть полезным способом помочь вам понять, движется ли архитектура в верном направлении, но они не заменяют необходимость реального общения с людьми на местах. На самом деле я бы предположил, что определение правильных функций потребует тесного сотрудничества.

Архитектура в потоковой организации

В главе 15 мы рассмотрели, как современные организации, предоставляющие ПО, переходят к более потокоориентированной модели, в которой автономные независимые команды сосредоточены на сквозной доставке функциональности, а их приоритеты определяются продуктом. Мы также говорили о сквозных командах — *вспомогательных командах*, сопровождающих потоковые команды. Как архитектор вписывается в этот мир? Что ж, иногда задачи потоковой команды достаточно сложны, чтобы потребовать специального архитектора (и здесь мы часто видим размывание границ между традиционными ролями технического руководителя и архитектора). Однако во многих случаях архитекторов просят работать в нескольких командах.

Многие обязанности архитектора можно рассматривать как обязанности поддержки: четкое донесение технического видения, понимание проблем по мере их возникновения и помощь в соответствующей адаптации технического видения. Архитектор помогает объединить людей, следя за общей картиной и помогая командам понять, как их деятельность вписывается в большее целое. Это прекрасно соответствует идее о том, что архитектор относится к команде поддержки, как показано на рис. 16.4. Такая команда поддержки могла бы состоять из разных специалистов, возможно, из людей, работающих в команде на полную ставку, и других, которые время от времени приходят на помощь.

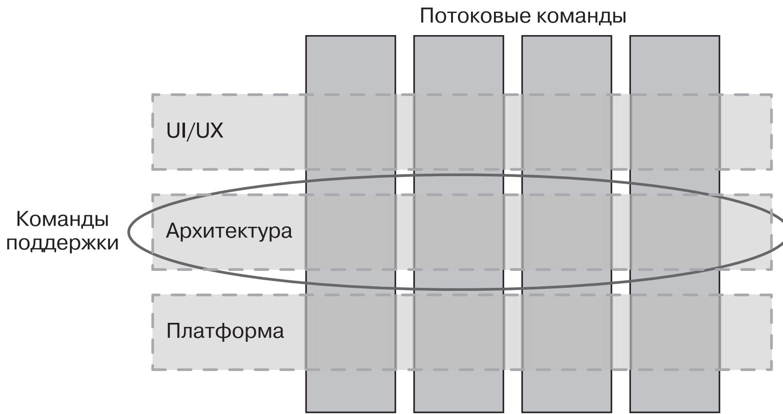


Рис. 16.4. Архитектура функционирует как команда поддержки

Модель, которую я очень люблю, заключается в том, чтобы в этой команде было небольшое количество преданных своему делу архитекторов (возможно, во многих случаях всего один или два человека), но чтобы со временем эта команда пополнялась технологами из каждой группы доставки — как минимум техническими руководителями каждой команды. Архитектор отвечает за то, чтобы группа работала. Такой подход позволяет распределить работу и гарантирует более высокий уровень заинтересованности. Эта модель также обеспечивает свободный поток информации из команд в группу, и в результате принятие решений становится гораздо более разумным и обоснованным.

Иногда группа может принимать решения, с которыми архитектор не согласен. Что должен делать архитектор в таком случае? Поскольку я бывал в таком положении раньше, могу сказать вам, что это одна из самых сложных ситуаций, с которыми приходится сталкиваться. Часто я придерживаюсь подхода, согласно которому мне стоит следовать групповому решению. Я считаю, что сделал все возможное, чтобы убедить людей, но в конечном счете оказался недостаточно убедителен. Группа часто намного мудрее отдельного человека, и мне не раз доказывали, что я ошибался! И представьте, насколько группу деморализует то, что ей была предоставлена возможность принятия решения, а в конечном счете ее мнение проигнорировали. Но иногда я брал верх над группой. Но когда и почему? Где эта граница?

Подумайте о том, как научить ребенка ездить на велосипеде. Вы не можете сделать это за него. Вы наблюдаете, как ребенок раскачивается при езде, но если вы будете вмешиваться каждый раз, когда кажется, что он вот-вот упадет, то он никогда не научится, и в любом случае падения происходят гораздо реже, чем вы думаете! Но если вы видите, что он собирается свернуть на проезжую часть или в ближайший утиный пруд, тогда вам придется вмешаться. Конечно, я часто

оказывался не прав в таких ситуациях — я позволял команде сделать что-то, что я считал неправильным, и то, что они делали, срабатывало! Точно так же, будучи архитектором, вы должны четко понимать, когда, образно говоря, ваша команда направляется в утиный пруд. Вы также должны отдавать себе отчет в том, что, даже если вы знаете, что правы, и отвергаете решение команды, это может подорвать вашу позицию, а также заставить команду почувствовать, что у них нет права голоса. Иногда лучше согласиться с решением, с которым вы не согласны. Знать, когда это делать, а когда — нет, непросто, но иногда жизненно важно.

Интересные моменты, о которых мы вскоре поговорим, возникают, когда архитектору также приходится участвовать в деятельности по управлению. Это может вызвать некоторую путаницу в отношении роли любой сквозной архитектурной команды. Что происходит, когда одна команда отклоняется от технической стратегии? Нормально ли это? Возможно, это разумное исключение, но оно также может вызвать более фундаментальные проблемы. Краткосрочное решение, принятое во имя целесообразности, может поставить под угрозу вносимые более масштабные изменения. Представьте, что архитектурная группа пытается помочь организации отказаться от использования централизованных данных из-за возникающих сопутствующих проблем со связью и эксплуатацией, но одна из команд решает просто добавить некоторые новые данные в общую БД, поскольку на нее оказывается давление с требованиями быстрой доставки. Что произойдет?

По моему опыту, все это сводится к хорошей, четкой коммуникации и пониманию ответственности. Если бы я увидел, что владелец продукта принимает решения, способные, по моему мнению, подорвать какую-то сквозную деятельность, над которой я работал, я бы пошел и поговорил с ним. Возможно, ответ заключается в том, что краткосрочное решение станет правильным (и, возможно, в конечном счете это своего рода технический долг, который мы сознательно взяли на себя). В других случаях, вероятно, владелец продукта способен изменить то, что он планирует, чтобы помочь работать с общей стратегией. В худшем случае проблему придется обострять.

В REA, онлайн-компании по продаже недвижимости, о которой я рассказывал в нескольких предыдущих главах, владельцы продуктов иногда принимали решения о расстановке приоритетов в работе таким образом, что это приводило к накоплению технического долга, а это, в свою очередь, — к последующим сложностям. Проблема заключалась в том, что владельцы продуктов в первую очередь отвечали за их способность предоставлять функции и радовать клиентов, в то время как часто вопросы, связанные с техническим долгом, возлагались на технических руководителей. Был выполнен переход к тому, чтобы владельцы продукта также отвечали за технические аспекты ПО. Это означало, что они должны были играть более активную роль в понимании технических сторон

системы (например, безопасности или производительности) и более тесно сотрудничать с техническими экспертами в плане определения приоритетов работы. Сделать владельцев нетехнических продуктов более ответственными за расстановку приоритетов в технической деятельности нетривиально, но, по моему опыту, это того стоит.

Создание команды

Быть главным специалистом по техническому видению вашей системы и обеспечивать выполнение этого видения, — это не просто способность убедиться, что принимаются правильные технологические решения. Это люди, с которыми вы работаете и которые будут выполнять эту работу. Роль любого технического руководителя заключается в том, чтобы помогать этим людям расти, помогать им быть частью создания этого видения и гарантировать, что они также могут быть активными участниками в формировании и реализации этого видения.

Помощь окружающим вас людям в их собственном карьерном росте может принимать различные формы, большинство из которых выходит за рамки книги. Однако есть один аспект, в котором микросервисная архитектура особенно актуальна. В более крупных монолитных системах у людей меньше возможностей продвинуться и «завладеть» чем-то. С другой стороны, при применении микросервисов у нас есть несколько автономных кодовых баз, имеющих свои собственные независимые жизненные циклы. Помощь людям в том, чтобы подняться на ступеньку выше, дав им возможность стать владельцами отдельных микросервисов, прежде чем брать на себя больше ответственности, может стать отличным способом помочь им достичь своих собственных карьерных целей, и в то же время это облегчает нагрузку на тех, кто отвечает за систему в целом!

Я твердо убежден, что великое ПО исходит от великих людей. Если заботиться только о технологической стороне уравнения, вы рискуете упустить из виду больше половины картины.

Требуемый стандарт

Когда вы прорабатываете свои методы и думаете о компромиссах, на которые предстоит пойти, один из самых важных балансов, который нужно найти, — это то, какую вариативность следует допускать в системе. Один из ключевых способов понять, что должно быть неизменным от микросервиса к микросервису, — это определить, как выглядит исправный микросервис. Что такое «добропорядочный» микросервис в вашей системе? Какими возможностями он должен обладать, чтобы гарантировать, что ваша система управляема и что

один ненадежный микросервис не выведет из строя всю систему? Как и в случае с людьми, если микросервис — «добропорядочный» в одном контексте, это не значит, что он будет таковым в другом. Тем не менее есть некоторые общие характеристики хорошо управляемых микросервисов, которые довольно важно соблюдать. Допущение слишком большого расхождения в этих нескольких ключевых характеристиках может привести к довольно тяжелым последствиям. Как сказал Бен Кристенсен из Facebook, когда вы думаете о более широкой картине, «это должна быть целостная система, состоящая из множества мелких частей с автономными жизненными циклами, но все они должны быть единым целым». Таким образом, вам нужно найти баланс, при котором вы оптимизируете автономность отдельных микросервисов, не упуская из виду общую картину. Определение четких атрибутов, которыми должен обладать каждый микросервис, — это один из способов прояснить, где находится этот баланс. Давайте коснемся некоторых атрибутов.

Мониторинг

Очень важно, чтобы мы могли составить согласованное межсервисное представление о работоспособности нашей системы. Это должно быть общесистемное представление, а не специфичное для микросервиса. В главе 10 мы обсуждали, что знание о состоянии «здоровья» отдельного микросервиса полезно, но часто только в ситуации, когда вы пытаетесь диагностировать более широкую проблему или понять более масштабную тенденцию. Чтобы максимально упростить задачу, я бы предложил сделать так, чтобы все микросервисы одинаково передавали метрики, связанные с работоспособностью и общим мониторингом.

Вы можете выбрать механизм подачи, при котором каждый микросервис будет передавать эти данные в определенный центр обработки. Что бы вы ни выбрали, старайтесь, чтобы это было стандартизировано. Сделайте технологию внутри машины непрозрачной и не требуйте, чтобы ваши системы мониторинга менялись для ее поддержки. Ведение логов здесь относится к той же категории данных: они должны храниться в одном месте.

Интерфейсы

Выбор небольшого числа определенных технологий интерфейса помогает интегрировать новых потребителей. Наличие одного стандарта — это хорошо. Два — не так уж плохо. Наличие 20 различных стилей интеграции — это нехорошо. Речь идет не только о выборе технологии и протокола. Например, если вы выберете HTTP/REST, будете ли вы использовать методы или принципы? Как вы будете разбивать ресурсы на страницы? Как вы будете управлять версиями конечных точек?

Архитектурная безопасность

Мы не можем позволить, чтобы один плохо зарекомендовавший себя микросервис испортил всем праздник. Необходимо убедиться, что микросервисы соответствующим образом защищают себя от нездоровых вызовов, идущих по потоку. Чем больше в системе различных микросервисов, которые должным образом не справляются с потенциальным сбоем нижестоящих вызовов, тем более хрупкими будут системы. Это может означать, например, что нужно ввести определенные правила межсервисного взаимодействия, такие как требование использования автоматических выключателей (тема, которую мы изучили в разделе «Шаблоны стабильности» главы 12).

Игра по правилам также важна, когда дело доходит до кодов ответов. Если ваши автоматические выключатели полагаются на коды HTTP и один микросервис решает отправить обратно коды 2XX для указания ошибок или путает коды 4XX с 5XX, то эти меры безопасности могут не сработать. Аналогичные проблемы будут встречаться, даже если вы не используете HTTP. Необходимо понимать разницу между корректным запросом, который был правильно обработан, неверным запросом, не позволившим микросервису что-либо с ним сделать, и запросом, который может быть корректным, но нельзя сказать точно, потому что сервер упал. Знание этого — ключ к тому, чтобы мы могли оперативно реагировать на отказы и отслеживать проблемы. Если команды микросервисов будут трактовать вольно данные правила, мы получим более уязвимую систему.

Управление и мощеная дорога

Часть того, чем должны заниматься архитекторы, — это управление. Что я подразумеваю под *управлением*? Оказывается, фреймворк COBIT (Control Objectives for Information Technologies — «задачи управления для информационных и смежных технологий») содержит довольно хорошее определение[1]:

> Управление обеспечивает достижение целей предприятия путем оценки потребностей, условий и вариантов заинтересованных сторон, определения направления посредством расстановки приоритетов и принятия решений, а также мониторинга эффективности, соответствия и прогресса согласно утвержденному направлению и целям.

В двух словах: можно рассматривать управление как согласование того, как все должно быть сделано, обеспечение того, чтобы люди знали об этом, и сле-

[1] COBIT 5: Бизнес-модель по руководству и управлению ИТ на предприятии. — Роллинг-Медоус, Иллинойс: ISACA, 2012.

дили, чтобы все делалось именно так. В некоторых средах управление происходит просто неформально, как часть обычной деятельности по разработке ПО. В других средах, особенно в более крупных организациях, это может быть более конкретная функция.

Управление может применяться ко многим вещам на площадке ИТ. Мы хотим сосредоточить внимание на аспекте технического управления, что, на мой взгляд, является обязанностью архитектора. Если одна из задач архитектора заключается в обеспечении наличия технического видения, то управление — это обеспечение соответствия системы этому видению, а также изменение видения при необходимости.

По сути, управление должно быть групповой деятельностью. Должным образом функционирующая группа управления может работать совместно, чтобы разделить рабочие обязанности и сформировать видение. Это может быть неформальная беседа с достаточно небольшой командой или более структурированная регулярная встреча с формальным членством в группе для охвата более широкого круга сотрудников. Именно здесь рассмотренные ранее принципы должны обсуждаться и изменяться по мере необходимости. Если требуется формальная группа, то она должна состоять преимущественно из людей, выполняющих работу, которая является предметом управления. Эта группа также должна отвечать за отслеживание технических рисков и управление ими.

Собраться вместе и договориться о том, как решать определенные задачи, — это хорошая идея. Но тратить время на то, чтобы убедиться, что люди следуют этим рекомендациям, менее приятно, равно как и возлагать на разработчиков бремя реализации всех этих стандартных действий, ожидаемых от каждого микросервиса. Я твердо верю в то, что корректно выполнять свои обязанности легко, и обсуждаемая в главе 15 мощеная дорога здесь действительно нужна. Роль архитектора заключается в том, чтобы четко сформулировать направление деятельности и облегчить путь к достижению цели. Таким образом, архитекторы должны участвовать в формировании требований к любой создаваемой мощеной дороге. Для многих платформа станет самым ярким примером — архитектор в конечном счете становится важной заинтересованной стороной для команды платформы.

Мы уже достаточно подробно рассмотрели роль платформы, поэтому давайте обсудим ряд других методов, которые можно использовать, чтобы максимально облегчить людям выполнение задач.

Примеры

Письменная документация хороша и полезна. Я ясно вижу в этом ценность — в конце концов, я написал эту книгу. Но разработчикам также нравится код. Код, который они могут запускать и исследовать. Если у вас есть набор стандартов или лучших практик, которые вы хотели бы поощрять, то полезно иметь под

рукой примеры, на которые можно ссылаться. Идея заключается в том, что люди не смогут сильно ошибиться, просто имитируя некоторые из лучших частей вашей системы.

В идеале это должны быть корректно работающие в вашей системе микросервисы, а не изолированные сервисы, которые реализованы просто как «эталоны». Убедившись, что ваши примеры действительно используются, вы гарантируете, что все ваши принципы действительно имеют смысл.

Адаптированный шаблон микросервиса

Разве не было бы здорово, если бы вы могли сделать так, чтобы всем разработчикам было действительно легко следовать большинству ваших рекомендаций, проделав при этом небольшую работу? Что, если бы разработчики «из коробки» подготовили подавляющую часть кода для реализации основных атрибутов, необходимых каждому микросервису?

Существует множество фреймворков для разных языков программирования, пытающихся предоставить строительные блоки для вашего собственного шаблона микросервиса. Spring Boot (https://oreil.ly/KYWe5), вероятно, самый успешный пример подобного фреймворка для JVM. Базовая платформа Spring Boot framework довольно легкая, но вам может потребоваться объединить набор библиотек для предоставления таких функций, как проверка работоспособности, обслуживание HTTP или предоставление метрик. Итак, прямо из коробки вы получите простое микросервисное приложение Hello World, которое можно запустить из командной строки.

Затем многие люди берут эти фреймворки и стандартизируют такую настройку для своей компании. Например, при запуске нового микросервиса они могут написать скрипты таким образом, чтобы получить шаблон Spring Boot с уже подключенными базовыми библиотеками, используемыми в их организации. Шаблон может уже содержать библиотеки для обработки автоматических выключателей и быть настроен на обработку аутентификации JWT для входящих вызовов. Обычно такое автоматическое создание шаблона также создает соответствующий конвейер сборки.

Осторожность оправданна

Выбор и настройка этих специализированных шаблонов микросервисов, как правило, являются задачей команды платформы. Они могут, например, предоставить шаблон для каждого поддерживаемого языка, гарантируя, что при использовании шаблона получаемые микросервисы будут хорошо работать с самой платформой. Однако это может вызвать трудности.

Я видел, как моральный дух и продуктивность многих команд были подорваны навязыванием им обязательных фреймворков. В стремлении улучшить

повторное использование кода все больше и больше работы помещается в централизованную структуру, пока она не становится непомерным чудовищем. Если вы решите применять специально разработанный шаблон микросервиса, очень тщательно подумайте о том, в чем заключается его задача. В идеале его использование должно быть строго опциональным, но если вы собираетесь более активно его внедрять, то должны понимать, что простота использования для разработчиков должна быть главной направляющей силой. Разрешение разработчикам, использующим шаблон, рекомендовать и даже вносить изменения в фреймворк, вероятно, как часть внутренней модели с открытым исходным кодом может здесь сильно помочь.

Из раздела «DRY и опасности повторного использования кода в мире микросервисов» главы 5 мы помним об опасностях совместного использования кода. Стремясь создать многократно используемый код, мы можем создать источники связанности между микросервисами. По крайней мере, одна организация, с представителем которой я общался, настолько обеспокоена этим, что фактически копирует код шаблона своего микросервиса вручную в каждый микросервис. Это означает, что обновление основного шаблона микросервиса занимает больше времени, но это меньше заботит организацию, чем опасность повышения связанности. Другие команды, с которыми я общался, просто рассматривали шаблон микросервиса как общую двоичную зависимость, хотя они должны быть очень внимательны, чтобы не допустить, чтобы тенденция DRY (не повторяйтесь) привела к чрезмерно связанной системе!

Мощеная дорога при масштабировании

Использование внутренних шаблонов и фреймворков микросервисов часто встречается в организациях с большим количеством микросервисов. Netflix и Monzo — две такие компании. В каждой из них решили в некоторой степени стандартизировать свой технологический стек (JVM в случае Netflix, Go в случае Monzo), что позволяет ускорить создание нового микросервиса со стандартным, ожидаемым поведением, используя общепринятый набор инструментов. При более разнородном технологическом стеке создать стандартный шаблон микросервиса для собственных нужд становится уже сложнее.

Если бы вы использовали несколько различных технологических стеков, вам понадобился бы соответствующий шаблон микросервиса для каждого из них. Однако это может быть способом незаметно ограничить выбор языка в своих командах. Если собственный шаблон микросервиса поддерживает только JVM, то у людей может пропасть желание выбирать альтернативные стеки, чтобы не делать лишнюю работу. Netflix, например, особенно заботится о таких аспектах, как отказоустойчивость, чтобы гарантировать, что сбой в работе одной части его системы не приведет к сбою всей системы. Чтобы справиться

с этим, была проделана огромная работа по обеспечению наличия клиентских библиотек в JVM, чтобы предоставить командам инструменты, необходимые им для обеспечения надлежащего функционирования их микросервиса. Внедрение нового технологического стека означало бы необходимость воспроизвести все эти усилия. В Netflix беспокойство вызывает не столько повторное выполнение работы, сколько тот факт, что при таком подходе легко ошибиться. Риск того, что микросервис неправильно реализует недавно внедренную отказоустойчивость, высок, если это может повлиять на большую часть системы. Netflix смягчает это обстоятельство, используя побочные сервисы (sidecar services), которые локально взаимодействуют с JVM, использующей соответствующие библиотеки.

Сервисные сети дали еще один потенциальный способ снизить нагрузку процесса выполнения общего поведения. Некоторые функциональные возможности, которые обычно рассматривались как ответственность внутреннего микросервиса, теперь могут быть перенесены в микросервисную сеть. Это способно обеспечить большую согласованность поведения между микросервисами, написанными на разных языках программирования, а также уменьшить ответственность этих шаблонов микросервисов.

Технический долг

Мы часто оказываемся в ситуациях, когда не можем в точности следовать своему техническому видению. Иногда приходится делать выбор, чтобы срезать несколько углов для получения некоторых важных функций. Это еще один компромисс, на который нам придется пойти. Существование технического видения обусловлено определенной идеей. Отклонение от этой идеи может принести краткосрочную выгоду, но повлечь за собой долгосрочные издержки. Помогающая нам понять данный компромисс концепция — это технический долг. Когда технический долг накапливается, его ценность остается постоянной, точно так же как у долга в реальном мире, и мы хотим его погасить.

Иногда технический долг — это не только то, что появляется от использования коротких путей в разработке. Что произойдет, если наше видение системы изменится, но не вся система соответствует ему? В такой ситуации мы также создаем новые источники технического долга.

Задача архитектора — взглянуть на картину в целом и понять этот баланс. Важно иметь некоторое представление об уровне долга и о том, какие части системы к нему относятся. В зависимости от организации вы можете предоставить мягкое руководство, но стоит позволить командам самим решать, как отслеживать и погашать долг. Для других организаций может потребоваться более структурированный подход, возможно, нужно регулярно вести и проверять логи по задолженности.

Обработка исключений

Таким образом, принципы и методы определяют, как должны быть построены наши системы. Но что происходит, когда система отклоняется от этого? Иногда мы принимаем решение, которое становится всего лишь исключением из правил. В таких случаях, возможно, стоит зафиксировать подобное решение где-нибудь в логе для дальнейшего использования. Если будет найдено достаточное количество *исключений*, в конечном счете может появиться смысл изменить применяемый принцип или метод, чтобы отразить новое понимание мира. Например, может быть метод, который гласит, что всегда необходимо использовать MySQL для хранения данных. Но затем появляются веские причины использовать Cassandra для масштабируемого хранилища, и в этот момент мы меняем метод и пишем: «Используйте MySQL для большинства требуемых хранилищ, если ожидается большой рост объемов хранения — используйте Cassandra».

Однако стоит повторить, что все организации разные. Я работал с некоторыми компаниями, в которых команды разработчиков обладают высокой степенью доверия и автономии, а принципы легковесны (и необходимость в явной обработке исключений значительно снижается, если не устраняется вовсе). В более структурированных организациях, в которых разработчикам дано меньше свободы, отслеживание исключений может быть жизненно важным для обеспечения того, чтобы действующие правила должным образом отражали проблемы, с которыми сталкиваются люди. Учитывая все сказанное, я являюсь поклонником микросервисов как способа оптимизации для обеспечения автономии команд, предоставляя им как можно больше свободы при решении поставленной задачи. Если вы работаете в организации, накладывающей множество ограничений на то, как разработчики могут выполнять свою работу, то микросервисы, скорее всего, не для вас.

Резюме

Подводя итог этой главе, вот что я вижу в качестве основных обязанностей эволюционного архитектора.

Видение

Убедитесь, что существует четко изложенное техническое видение системы, которое поможет ей соответствовать требованиям ваших клиентов и организации.

Эмпатия

Поймите влияние ваших решений на ваших клиентов и коллег.

Сотрудничество

Взаимодействуйте с как можно большим количеством ваших специалистов и коллег, чтобы помочь определить, усовершенствовать и реализовать видение.

Адаптивность

Убедитесь, что техническое видение меняется в соответствии с требованиями ваших клиентов или организации.

Автономия

Найдите правильный баланс между стандартизацией и предоставлением самостоятельности вашим командам.

Управление

Убедитесь, что внедряемая система соответствует техническому видению и что людям легко выполнять свои обязанности.

Эволюционный архитектор — это тот, кто понимает, что проявление этих основных обязанностей становится постоянным балансированием. Силы всегда толкают нас в ту или иную сторону, и понимание того, где нужно отступить, а где просто плыть по течению, часто приходит только с опытом. Но худшая реакция на все эти силы, которые подталкивают нас к переменам, — это стать более жесткими или зацикленными в своем мышлении.

Хотя многие советы, приведенные в данной главе, применимы к любому системному архитектору, микросервисы дают гораздо больше возможностей для принятия решений. Поэтому крайне важно уметь находить баланс между всеми этими компромиссами. Если вы хотите изучить эту тему более глубоко, я могу порекомендовать уже процитированные здесь книги «Эволюционная архитектура» и «Лифт архитектора ПО»[1] Грегора Хопа, которые помогают архитекторам понять, как преодолеть разрыв между стратегическим мышлением высокого уровня и реализацией на практике.

Мы почти подошли к концу книги и многое уже обсудили. Давайте подытожим, что вы узнали.

---

[1] *Hohpe G.* The Software Architect Elevator. — O'Reilly, 2020.

Послесловие: подведем итог основных тем

Книга охватывает много тем, и я поделился множеством советов. Учитывая широту охвата повествования, я счел разумным обобщить некоторые из моих ключевых советов, касающихся микросервисных архитектур. Для тех, кто прочитал книгу целиком, это должно стать отличной возможностью освежить память. Тем из вас, кто нетерпелив и поспешил перейти к концу, хочу сказать, что за этими советами стоит *много* деталей, и я бы настоятельно рекомендовал вам ознакомиться с подробностями, скрывающимися за этими идеями, а не принимать их вслепую.

Учитывая все сказанное, я постараюсь сделать это послесловие как можно более кратким, так что давайте начнем.

Что такое микросервисы

Как описано в главе 1, *микросервисы* представляют собой тип сервис-ориентированной архитектуры, которая фокусируется на возможности независимого развертывания. *Независимое развертывание* означает, что вы можете внести изменения в микросервис, развернуть его и предоставить функциональность конечным пользователям, не требуя изменения других микросервисов. Получение максимальной отдачи от микросервисной архитектуры означает принятие данной концепции. Как правило, каждый микросервис развертывается как процесс, при этом связь с другими микросервисами осуществляется по той или иной форме сетевого протокола. Обычно развертывается несколько экземпляров микросервиса, возможно, для обеспечения большего масштаба или для повышения надежности за счет избыточности.

Чтобы обеспечить независимое развертывание, нужно убедиться, что при изменении одного микросервиса не нарушается взаимодействие с другими сервисами. Для этого необходимо, чтобы наши интерфейсы взаимодействия с другими микросервисами были стабильными и чтобы изменения вносились с учетом обратной совместимости. *Скрытие информации*, о котором я подробно рассказал в главе 2, описывает подход, при котором за интерфейсом скрывает-

ся как можно больше информации (код, данные). Вы должны предоставлять только самый минимум через интерфейсы сервиса, чтобы удовлетворить своих потребителей. Чем меньше вы раскрываете, тем проще обеспечить обратную совместимость. Скрытие информации также позволяет вносить технологические изменения в рамках микросервиса таким образом, чтобы это не повлияло на потребителей.

Одним из ключевых способов реализации возможности независимого развертывания стало *скрытие базы данных*. Если микросервису необходимо сохранять состояние в БД, это должно быть полностью скрыто от внешнего мира. Внутренние БД не должны предоставляться напрямую внешним потребителям, поскольку это приводит к возникновению слишком сильной связанности между ними, что снижает возможность независимого развертывания. В общем, избегайте ситуаций, в которых несколько микросервисов обращаются к одной и той же БД.

Микросервисы очень хорошо работают с *предметно-ориентированным проектированием* (DDD). DDD предоставляет концепции, помогающие нам определить границы своих микросервисов, при этом результирующая архитектура ориентирована на предметную область бизнеса. Это чрезвычайно полезно в ситуациях, когда организации создают ИТ-команды, более ориентированные на бизнес. Теперь, когда команда сосредоточена на одной части предметной области бизнеса, она может взять на себя управление микросервисами, соответствующими данной части бизнеса.

Переход к микросервисам

Микросервисы привносят *много* сложностей. Настолько, что стоит серьезно подумать, прежде чем решаться на переход к микросервисной архитектуре. Я по-прежнему убежден, что простой монолит с одним процессом является абсолютно разумной отправной точкой для новой системы. Однако со временем мы кое-чему учимся и начинаем видеть причины того, почему нынешняя системная архитектура больше не подходит для нашей цели. На этом этапе уместно стремиться к переменам.

Важно понимать, чего вы пытаетесь добиться от внедрения микросервисной архитектуры. Какова цель? Какой положительный результат, по вашему мнению, принесет переход на микросервисы? Результат, к которому вы стремитесь, напрямую повлияет на то, *как* вы будете разделять свой монолит. Если вы пытаетесь изменить архитектуру своей системы, чтобы лучше справляться с масштабированием, вы в конечном счете внесете иные изменения, чем если бы вашей основной целью было повышение автономии организации. Подробнее я говорил об этом в главе 3 и еще более подробно — в моей книге «От монолита к микросервисам».

Многие проблемы с микросервисами становятся очевидными только после того, как вы выпускаете проект в эксплуатацию. Поэтому я настоятельно рекомендую постепенную, эволюционную декомпозицию существующего монолита, а не переписывание его «с размаху». Определите микросервис, который вы хотите создать, извлеките соответствующую функциональность из монолита, разверните новый микросервис в среде эксплуатации и начните активно использовать его. Исходя из этого, вы увидите, помогаете ли вы себе продвигаться к цели. Но вы также узнаете много нового, что облегчит извлечение следующего микросервиса, или, возможно, это подскажет вам, что микросервисы все-таки не ваш путь развития!

Стили взаимодействия

Я обобщил основные формы взаимодействия между микросервисами в главе 4 и снова представил их на рис. П.1. Это не универсальная модель, а просто обзор некоторых наиболее распространенных типов коммуникации.

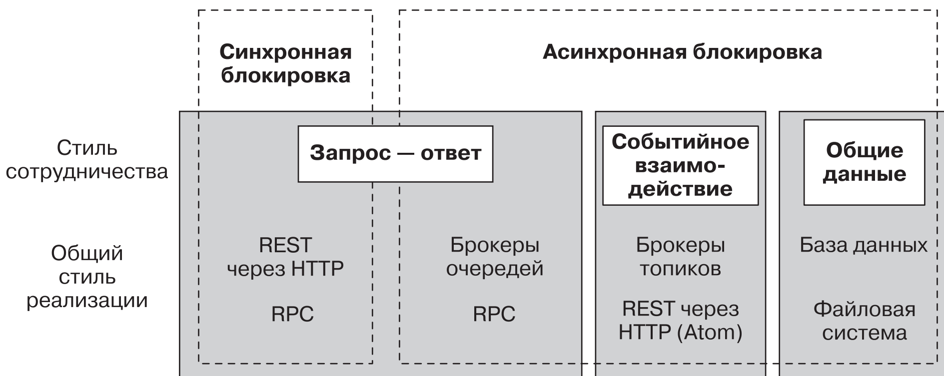


Рис. П.1. Различные стили межсервисной коммуникации наряду с примерами технологий внедрения

При взаимодействии *«запрос — ответ»* микросервис отправляет запрос нижестоящему микросервису и ожидает ответа. При синхронном взаимодействии в стиле «запрос — ответ» ожидается, что ответ вернется к экземпляру микросервиса, отправившему запрос. При асинхронном взаимодействии в данном стиле ответ может вернуться к другому экземпляру вышестоящих микросервисов.

При *событийном взаимодействии* микросервис отправляет событие и другие микросервисы, если они заинтересованы в этом событии, могут отреагировать на него. События представляют собой просто констатацию факта — информацию, которой делятся о чем-то, что произошло. При событийном взаимодействии микросервис не сообщает другому, что делать, — он просто делится событиями. Решение о том,

что делать с полученной информацией, зависит от нижестоящих микросервисов. Событийная коммуникация определяется как асинхронная по своей природе.

Один микросервис может реализовать взаимодействие по нескольким протоколам. Например, на рис. П.2 показан микросервис `Доставка`, предоставляющий интерфейс REST для взаимодействия в стиле «запрос — ответ», который также запускает события при внесении изменений.

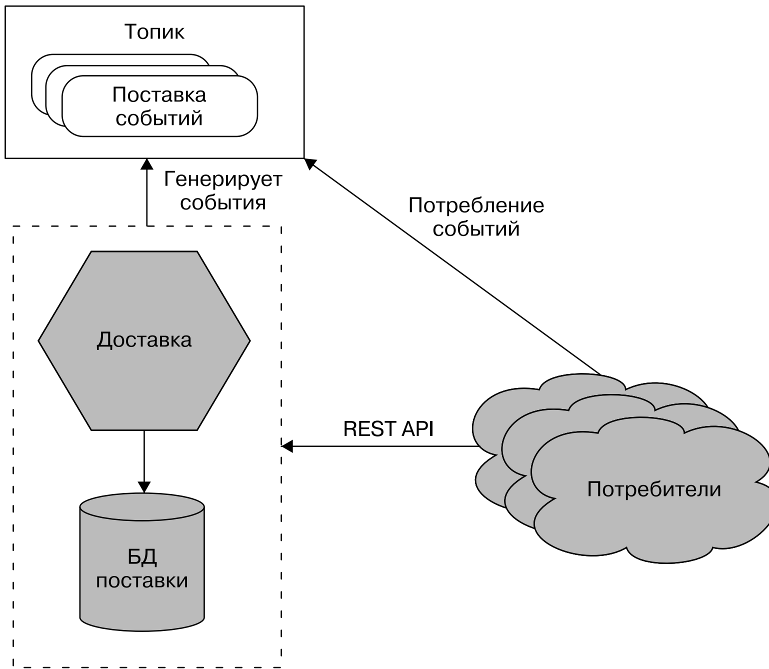


Рис. П.2. Микросервис, предоставляющий свои функциональные возможности через REST API и топик

Событийное сотрудничество может упростить создание более слабо связанных архитектур, но для понимания того, как ведет себя система, может потребоваться больше действий. Данный тип взаимосвязи также часто требует использования специальных технологий, таких как брокеры сообщений, что может еще сильнее усложнить ситуацию. Если вы можете использовать полностью управляемый брокер сообщений, это поможет снизить стоимость систем подобного типа.

Модели взаимодействия «запрос — ответ» и событийное взаимодействие занимают определенное место в проектировании, и часто то, какую из них вы используете, зависит от личных предпочтений. Некоторые проблемы проще решить с помощью одной модели, чем другой, и для микросервисной архитектуры характерно сочетание стилей.

Рабочий поток

Если вы хотите заставить несколько микросервисов сотрудничать для выполнения какой-либо всеобъемлющей операции, попробуйте явно смоделировать процесс с помощью *саг*. Эту тему мы исследовали в главе 6.

Как правило, следует избегать распределенных транзакций в ситуациях, когда вместо них можно использовать сагу. Распределенные транзакции значительно усложняют системы, у них проблемные режимы сбоев и они часто не обеспечивают ожидаемого результата, даже когда работают корректно. Практически во всех случаях саги лучше подходят для реализации бизнес-процессов, охватывающих несколько микросервисов.

Есть два разных стиля саг, которые следует учитывать: *оркестрованные саги* и *хореографические саги*. Оркестрованные саги используют централизованный оркестратор для координации с другими микросервисами и обеспечения выполнения задач. В целом это простой и понятный подход, но центральный оркестратор может в итоге сделать слишком много, если вы не будете осторожны, и это может стать источником разногласий, когда несколько команд работают над одним и тем же бизнес-процессом. В хореографических сагах нет оркестратора. Вместо этого ответственность за бизнес-процесс распределяется между несколькими сотрудничающими микросервисами. Чаще всего это более сложная архитектура для реализации, и она требует больше работы, чтобы гарантировать, что все происходит правильно. С другой стороны, такая архитектура гораздо менее подвержена повышению связанности и хорошо работает для нескольких команд.

Лично я люблю хореографические саги, я часто их использовал, но допустил много ошибок, воплощая их в жизнь. Я считаю, что оркестрованные саги прекрасно работают, когда ответственность за весь процесс лежит на одной команде, но они становятся более проблематичными с несколькими командами. Хореографические саги могут оправдать свою повышенную сложность в ситуациях, когда ожидается, что в процессе будут участвовать несколько команд.

Сборка

У каждого микросервиса должны быть собственная сборка, свой конвейер CI. Когда я вношу изменения в микросервис, я ожидаю, что смогу выполнить сборку этого микросервиса самостоятельно. Избегайте ситуаций, в которых вам приходится создавать все свои микросервисы совместно с другими, так как это значительно затрудняет независимое развертывание.

По причинам, изложенным в главе 7, я не стал поклонником монорепозиториев. Если вы действительно хотите их использовать, то изучите, какие

проблемы они вызывают из-за четкого разграничения прав собственности и потенциальной сложности сборок. Но обязательно убедитесь, что, независимо от того, используете ли вы подход с монорепозиторием или с мультирепозиториями, у каждого микросервиса есть свой собственный процесс сборки CI, который может запускаться независимо от любых других сборок.

Развертывание

Микросервисы обычно развертываются как процесс. Данный процесс может быть развернут на физической или виртуальной машине, контейнере или платформе FaaS. В идеале нужно, чтобы микросервисы были максимально изолированы друг от друга в развернутой среде. Ситуация, в которой один микросервис, использующий много вычислительных ресурсов, может повлиять на другой сервис, нежелательна. В общем, это означает, что требуется, чтобы каждый микросервис использовал свою собственную операционную систему и набор вычислительных ресурсов. Контейнеры особенно эффективны при предоставлении каждому экземпляру микросервиса собственного ограниченного набора ресурсов, что делает их отличным выбором для развертывания микросервисов.

Kubernetes может быть очень полезен, если вы хотите запускать контейнерные рабочие нагрузки на нескольких машинах. Я бы не рекомендовал применять его, если у вас всего лишь несколько микросервисов, поскольку это влечет за собой свои трудности. Там, где это возможно, используйте управляемый кластер Kubernetes, поскольку это позволяет избежать некоторых из этих сложностей.

FaaS — это интересная развивающаяся модель развертывания кода. Вместо того чтобы указывать, сколько копий какой-либо функциональности требуется, вы просто предоставляете свой код платформе FaaS и говорите: «Когда произойдет это событие, запустите этот код». Это *действительно* хорошо, с точки зрения разработчика, и я думаю, что подобная абстракция, скорее всего, станет будущим для большого объема серверной разработки. Однако текущие реализации не лишены проблем. С точки зрения микросервисов развертывание всего микросервиса как единой функции на платформе FaaS — это действительно прекрасный способ начать.

Последнее замечание: разделите в своем сознании концепции развертывания и релиза. Когда что-то прошло развертывание в эксплуатацию, не значит, что это должно быть доступно вашим пользователям. Разделяя данные концепции, вы открываете возможность развертывать свое ПО различными способами, например с помощью канареечных релизов или параллельных запусков. Все это и многое другое подробно описано в главе 8.

Тестирование

Иметь под рукой набор автоматизированных функциональных тестов — хорошая идея. Такие тесты дадут вам быструю обратную связь о качестве ПО до того, как его увидят пользователи. Микросервисы предоставляют множество вариантов с точки зрения различных типов тестов, которые можно написать, о чем мы подробно говорили в главе 9.

Однако по сравнению с другими типами архитектур сквозные тесты могут быть особенно проблематичными для микросервисов. В конечном счете их составление и обслуживание для микросервисных архитектур может оказаться более дорогостоящим, чем для простых нераспределенных систем, а сами тесты могут привести к гораздо большему количеству сбоев, не обязательно указывающих на проблему с вашим кодом. Комплексные тесты, охватывающие несколько команд, особенно сложны.

Со временем постарайтесь уменьшить зависимость от сквозных тестов. Рассмотрите возможность замены части усилий, затрачиваемых на эту форму тестирования, контрактами, управляемыми потребителями, проверкой совместимости схем и эксплуатационным тестированием. Эти действия вполне обеспечат гораздо более эффективную работу, чем сквозные тесты, позволят быстро выявлять проблемы в сильно распределенных системах.

Мониторинг и наблюдаемость

В главе 10 я объяснил, что *мониторинг* — это деятельность, нечто, что мы делаем с системой. Но такое сосредоточение внимания на деятельности, а не на результате — проблема, и эта тема проходит через всю книгу. Вместо этого мы должны сосредоточиться на *наблюдаемости* своих систем. Наблюдаемость — это степень, в которой можно понять, что делает система, основываясь на внешних выходных данных. Создание системы, обладающей хорошей наблюдаемостью, требует, чтобы мы внедрили это мышление в свое ПО и обеспечили доступность правильных типов внешних выходных данных.

Распределенные системы могут выходить из строя странным образом, и микросервисы ничем не отличаются. Мы не можем предсказать все причины сбоя системы, поэтому может быть трудно заранее определить, какая информация понадобится для диагностики и устранения неполадок. Использование инструментов, помогающих запрашивать эти внешние выходные данные неожиданными способами, становится все более важным. Я предлагаю вам взглянуть на такие инструменты, как Lightstep и Honeycomb, созданные с учетом данного подхода.

Наконец, по мере расширения вашей системы вероятность обнаружить ошибку резко возрастает. Но в крупномасштабной системе проблема на одной машине не повод для всеобщей тревоги, и не обязательно кого-то бесцеремонно будить в три часа ночи. Использование методов эксплуатационного тестирования, таких как параллельные прогоны и синтетические транзакции, может быть гораздо более эффективным для выявления проблем. Но такая практика также может повлиять на конечных пользователей.

Безопасность

Микросервисы дают больше возможностей для глубокой защиты приложения, что, в свою очередь, может привести к созданию более безопасных систем. Парадоксально, но они часто способствуют увеличению точек для атаки, что может сделать систему более уязвимой! Это уравновешивающее действие — вот почему так важно получить целостное понимание безопасности, о чем я рассказывал в главе 11.

С увеличением объема информации, передаваемой по сетям, становится все более важным уделять внимание защите передаваемых данных. Увеличенное количество подвижных элементов также означает, что автоматизация стала жизненно важной частью безопасности микросервисов. Управление патчами, сертификатами и секретами с помощью ручных процессов, подверженных ошибкам, делает систему уязвимой для атак. Поэтому используйте инструменты, обеспечивающие простоту автоматизации.

JWT-токены можно использовать для децентрализации логики авторизации таким образом, чтобы избежать необходимости в дополнительных обходах. Это поможет защититься от уязвимости, известной как confused deputy problem, и в то же время обеспечить более независимую работу микросервиса.

Наконец, все больше людей придерживаются образа мышления с *нулевым доверием*. При нулевом доверии вы действуете так, как будто ваша система уже была скомпрометирована и вам необходимо соответствующим образом построить свои микросервисы. Это может показаться параноидальной позицией, но я все больше склоняюсь к мнению, что принятие данного принципа на самом деле упростит ваше представление о безопасности системы.

Отказоустойчивость

В главе 12 мы рассмотрели отказоустойчивость в целом, и я поделился с вами четырьмя ключевыми концепциями, которые необходимо учитывать при размышлениях об отказоустойчивости.

Надежность

Способность поглощать ожидаемые возмущения.

Восстановление

Способность восстанавливаться после травмирующего события.

Стабильная расширяемость

Насколько хорошо мы справляемся с неожиданной ситуацией.

Непрерывная адаптивность

Способность постоянно адаптироваться к меняющимся условиям, заинтересованным сторонам и требованиям.

В целом микросервисные архитектуры могут помочь обеспечить некоторые из этих качеств (а именно, надежность и восстановление), однако сами по себе эти свойства не делают систему отказоустойчивой. В значительной степени отказоустойчивость зависит от командного и организационного поведения и культуры производства.

По сути, все команды должны создавать прозрачные решения, тогда надежность приложения возрастает. Но она не бесплатна. Микросервисы дают возможность повысить отказоустойчивость систем, и необходимо сделать выбор в их пользу или отказаться от них. Например, необходимо понимать, что любой вызов, выполняемый к другому микросервису, может завершиться неудачей, что машины могут сломаться и что с корректными сетевыми пакетами может произойти что-то нехорошее. Модели стабильности, такие как переборки, автоматические выключатели и правильно настроенные тайм-ауты, значительно помогут с этим справиться.

Масштабирование

Микросервисы предоставляют несколько различных способов масштабирования приложения. В главе 13 мы исследовали четыре оси масштабирования, перечисленные ниже.

Вертикальное масштабирование

В двух словах это означает приобретение более мощной машины.

Горизонтальное дублирование

Наличие нескольких устройств, способных выполнять одну и ту же работу.

Разделение данных

Разделение работы на основе какого-либо атрибута данных, например группы клиентов.

Функциональная декомпозиция

Разделение работы в зависимости от типа, например декомпозиция микросервиса.

При масштабировании сначала реализуйте простые способы. Вертикальное масштабирование и горизонтальное дублирование выполняются быстро и легко по сравнению с двумя другими представленными вариантами. Если они сработают — отлично! Если нет, вы можете применить другие механизмы. Также часто разработчики смешивают различные типы масштабирования, например разделение трафика на основе клиентов, а затем масштабирование каждого раздела по горизонтали.

Пользовательские интерфейсы

Слишком часто разделение UI становится запоздалой мыслью, когда дело доходит до декомпозиции системы: мы разбиваем свои микросервисы на части, но оставляем монолитный UI. Это, в свою очередь, приводит к проблемам, связанным с наличием отдельных фронтенд- и бэкенд-команд. Вместо этого нужны потоковые команды, когда один коллектив владеет всей функциональностью, связанной со сквозным фрагментом пользовательских функций. Чтобы это изменение произошло и появилась возможность избавиться от разрозненных фронтенд- и бэкенд-команд, необходимо разделить пользовательские интерфейсы.

В главе 14 рассказывается, как можно использовать *микрофронтенды* для предоставления разделенных пользовательских интерфейсов с использованием одностраничных фреймворков приложений, таких как React. Системы UI часто сталкиваются с проблемами с точки зрения количества вызовов, которые им необходимо совершить, или потому, что им требуется выполнить объединение вызовов и фильтрацию для мобильных устройств. Шаблон «Бэкенд для фронтенда» (BFF) поможет обеспечить объединение и фильтрацию на стороне сервера в этих ситуациях, хотя, если вы умеете использовать GraphQL, можете отказаться от применения BFF.

Организация

В главе 15 мы рассмотрели переход от горизонтально выровненных разрозненных команд к командным структурам, организованным на основе сквозных функциональных блоков. Эти *потоковые команды*, как их описывают авторы «Топологии команд», сопровождаются командами поддержки, как показано на рис. П.3. Команды поддержки часто имеют конкретную сквозную направленность, например фокусируются на безопасности или удобстве использования и поддерживают потоковые команды в этих аспектах.

Сделать эти потоковые команды максимально автономными означает, что им нужны инструменты самообслуживания, чтобы избежать необходимости постоянно просить другие команды что-то делать за них. В этом плане платформа может быть невероятно полезной. Однако важно, чтобы мы рассматривали платформу как разновидность *мощеной дороги*, то есть как нечто, что помогает легко выполнять свои обязанности, не требуя при этом ее *обязательного* использования. Необязательность платформы гарантирует, что простота ее использования остается ключевым направлением деятельности команды, владеющей ею, а также позволяет командам делать выбор в пользу иной платформы, когда это оправданно.

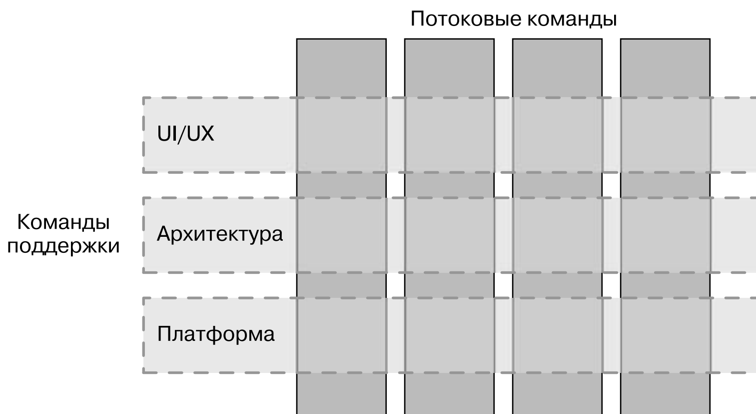


Рис. П.3. Команды поддержки сопровождают несколько потоковых команд

Архитектура

Важно, чтобы мы не рассматривали архитектуру своей системы как фиксированную и неизменяемую. Необходимо рассматривать ее как нечто, что должно быть способно постоянно меняться в зависимости от обстоятельств. Для того чтобы получить максимальную отдачу от архитектуры микросервисов, переход в организацию, где командам предоставляется больше автономии, означает, что ответственность за техническое видение должна стать более совместным процессом. Архитектор, сидящий в башне из слоновой кости, будет либо серьезным препятствием для микросервисной архитектуры, либо превратится в формальную, ничего не решающую должность.

Сопровождение архитектуры системы может быть полностью распределено между командами, и на определенном уровне масштаба организации это может работать хорошо. Однако по мере роста организации становится необходимым наличие людей, выделяющих время для изучения системы в целом. Называйте их как хотите: главными инженерами, владельцами технических продуктов

или архитекторами — это не имеет значения. Задачи, которые они должны выполнять, одни и те же. Как показано в главе 16, архитекторам микросервисной организации необходимо поддерживать команды, подключать людей, выявлять возникающие закономерности и проводить достаточно времени с командами, чтобы увидеть, как все это происходит в реальности.

Дополнительная литература

На протяжении всего повествования я ссылался на различные статьи, презентации и книги, из которых я многому научился, и я перечислил их в разделе «Библиография». Однако, начиная с первого издания, две книги, оказавшие наибольшее влияние на мое мышление и в результате получившие многочисленные ссылки в текущем издании, заслуживают упоминания здесь как «обязательные к прочтению». Первая — «Ускоряйся!» Николь Форсгрен, Джеза Хамбла и Джина Кима. Вторая — «Топологии команд» Мэтью Скелтона и Мануэля Паиса. Обе книги, на мой взгляд, представляют собой наибольшую ценность в сфере разработки ПО за последние десять лет. Независимо от того, увлекаетесь вы микросервисами или нет, их стоит прочесть.

В качестве дополнения советую ознакомиться с другой моей книгой — «От монолита к микросервисам». Она более подробно рассказывает, как разделить существующие системные архитектуры.

Взгляд в будущее

Я подозреваю, что в будущем технологии, облегчающие создание и запуск микросервисов, продолжат совершенствоваться, и мне особенно интересно посмотреть, как будут выглядеть продукты FaaS второго (и третьего) поколения. Независимо от того, приживется FaaS или нет, Kubernetes получит еще более широкое распространение, даже если он станет чаще скрываться за более удобными для разработчиков слоями абстракции. Kubernetes победил, но так, что, на мой взгляд, большинству разработчиков приложений не стоит беспокоиться. Мне по-прежнему очень интересно посмотреть, как стандарт Wasm изменит наше представление о развертываниях, и у меня все еще есть подозрение, что Unikernel тоже может получить вторую жизнь.

С момента выхода первого издания книги микросервисы действительно стали мейнстримом, что меня удивило, а также обеспокоило. Кажется, что многие люди, внедряющие микросервисы, делают это скорее потому, что так делают все остальные, а не потому, что микросервисы подходят именно им. Таким образом, я ожидаю, что мы услышим больше ужасных историй о неудачных попытках внедрения микросервисов, которые я с удовольствием усвою, чтобы определить,

чему можно научиться. Я также ожидаю, что в какой-то момент, когда количество негативных примеров использования микросервисов достигнет критической массы, в отрасли начнется обратная реакция против данной архитектуры. Применение критического мышления для определения того, какой подход имеет наибольший смысл в той или иной ситуации, не очень привлекательно или востребовано на рынке, и я не ожидаю, что это изменится в мире, где продавать технологии выгоднее, чем идеи.

Я не хочу показаться пессимистом! Мы как отрасль все еще очень молоды и все еще ищем свое место в мире. Количество энергии и изобретательности, которые вкладываются в разработку ПО, продолжает вызывать у меня интерес, и я не могу дождаться, чтобы увидеть, что принесет следующее десятилетие.

Заключительные слова

Микросервисные архитектуры дают огромное количество возможностей и не меньше вариантов решений. Принятие решений в мире микросервисов — более распространенное занятие, чем в мире простых монолитных систем. Вы не сможете принять все эти решения правильно, я могу это гарантировать. Итак, зная, что вы собираетесь что-то сделать неправильно, каковы ваши варианты? Что ж, я бы предложил найти способы сделать каждое решение небольшим по объему. Таким образом, если вы ошибетесь в одном из вариантов, влияние будет ограничено лишь небольшой частью системы. Научитесь принимать концепцию эволюционной архитектуры, в которой ваша система меняет форму и изменяется с течением времени по мере того, как вы изучаете что-то новое. Думайте не о масштабных преобразованиях, а о серии небольших изменений, вносимых в вашу систему с течением времени, чтобы сохранить ее гибкость.

Надеюсь, что к настоящему моменту я поделился достаточным количеством информации и опыта, чтобы помочь вам решить, подходят ли вам микросервисы. Если ответ — да, я надеюсь, вы воспринимаете это как путешествие, а не как пункт назначения. Двигайтесь постепенно. Разбирайте свою систему по частям, учась по ходу дела. И привыкайте к этому: во многих отношениях дисциплина, направленная на постоянное изменение и развитие систем, становится гораздо более важным уроком, чем любой другой, которым я поделился с вами в книге. Перемены неизбежны. Примите это.

Библиография

- *Бейер Б., Джоунс К., Петофф Дж., Мерфи Н. Р.* Site Reliability Engineering. Надежность и безотказность как в Google. — СПб.: Питер, 2021.
- *Бейер Б., Рензин Д., Мерфи Н. Р.* Site Reliability Workbook. Практическое применение. — СПб.: Питер, 2021.
- *Брукс Ф. П.* Мифический Человеко-Месяц, или Как создаются программные системы. — СПб.: Питер, 2021.
- *Вернон В.* Предметно-ориентированное проектирование. Самое основное.
- *Вернон В.* Реализация методов предметно-ориентированного проектирования.
- *Клеппман М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2018.
- *Шапира Г., Палино Т., Сиварам Р., Петти К.* Apache Kafka. Потоковая обработка и анализ данных, 2-е изд. — СПб.: Питер, 2023.
- *Эванс Э.* Предметно-ориентированное проектирование: Структуризация сложных программных систем.
- 2020 Data Breach Investigations Report. Verizon, 2020. https://oreil.ly/ps0Cx.
- *Abbott M. L., Fisher M. T.* The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. 2nd ed. — Boston: Addison-Wesley, 2015.
- *Allspaw J.* Blameless Post-Mortems and a Just Culture // Code as Craft (blog). Etsy, May 22, 2012. https://oreil.ly/P1BcX.
- *Bache E.* End-to-End Automated Testing in a Microservice Architecture // NDC Conferences. July 5, 2017. YouTube video, 56:48. https://oreil.ly/DbFdR.
- *Bell L., Brunton-Spall M., Smith R., Bird J.* Agile Application Security. — Sebastopol: O'Reilly, 2017.
- *Bird C., Nagappan N., Murphy B., Gall H., Devanbu P.* Don't Touch My Code! Examining the Effects of Ownership on Software Quality // ESEC/FSE '11: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 4–14. — New York: ACM, 2011. doi.org/10.1145/2025113.2025119.
- *Brandolini A.* EventStorming. — Victoria, BC: Leanpub, forthcoming.

- *Brown A., Forsgren N., Humble J., Kersten N., Kim G.* 2016 State of DevOps Report. https://oreil.ly/WJjhA.
- *Bryant D.* Apple Rebuilds Siri Backend Services Using Apache Mesos // InfoQ, May 3, 2015. https://oreil.ly/NsjEQ.
- *Burns B., Grant B., Oppenheimer D., Brewer E., Wilkes J.* Borg, Omega, and Kubernetes // Acmqueue 14, no. 1 (2016). https://oreil.ly/2TlYG.
- *Calcado P.* Pattern: Using Pseudo-URIs with Microservices. May 22, 2017. https://oreil.ly/uZuto.
- *Cockburn A.* Hexagonal Architecture // alistair.cockburn.us, January 4, 2005. https://oreil.ly/0JeIm.
- *Cohn M.* Succeeding with Agile. — Upper Saddle River, NJ: Addison-Wesley, 2009.
- *Colyer A.* Information Distribution Aspects of Design Methodology // The Morning Paper (blog), October 17, 2016. https://oreil.ly/qxj2m.
- *Crispin L., Gregory J.* Agile Testing: A Practical Guide for Testers and Agile Teams. — Upper Saddle River, NJ: Addison-Wesley, 2008.
- *Ford N., Parsons R., Kua P.* Building Evolutionary Architectures. — Sebastopol: O’Reilly, 2017.
- *Forsgren N., Smith D., Humble J., Frazelle J.* Accelerate: State of DevOps Report 2019. https://oreil.ly/A3zGn.
- *Forsgren N., Humble J., Kim G.* Accelerate: The Science of Building and Scaling High Performing Technology Organizations. — Portland, OR: IT Revolution, 2018.
- *Fowler M.* CodeOwnership // martinfowler.com, May 12, 2006. https://oreil.ly/a42c7.
- *Fowler M.* Eradicating Non-Determinism in Tests // martinfowler.com, April 14, 2011. https://oreil.ly/sqPOD.
- *Fowler M.* StranglerFigApplication // martinfowler.com, June 29, 2004. https://oreil.ly/foti0.
- *Freeman S., Pryce N.* Growing Object-Oriented Software, Guided by Tests. — Upper Saddle River, NJ: Addison-Wesley, 2009.
- *Friedrichsen U.* The Limits of the Saga Pattern // ufried.com (blog). February 19, 2021. https://oreil.ly/X1BfK.
- *Garcia-Molina H., Gawlick D., Klein J., Kleissner K.* Modeling Long-Running Activities as Nested Sagas // Data Engineering 14, no. 1 (March 1991): 14.18. https://oreil.ly/RVp7A1.
- *Garcia-Molina H., Salem K.* Sagas // ACM Sigmod Record 16, no. 3 (1987): 249–59.
- *Governor J.* Towards Progressive Delivery // James Governor’s MonkChips (blog). RedMonk, August 6, 2018. https://oreil.ly/OlkEY.

- *Hansson D. H.* The Majestic Monolith // Signal v. Noise, February 29, 2016. https://oreil.ly/fN5CR.
- *Hodgson P.* Feature Toggles (aka Feature Flags) // martinfowler.com, October 9, 2017. https://oreil.ly/pSPrd.
- *Hohpe G.* The Software Architect Elevator: Redefining the Architect's Role in the Digital Enterprise. Sebastopol: O'Reilly, 2020.
- *Hohpe G., Woolf B.* Enterprise Integration Patterns. — Boston: Addison-Wesley, 2003.
- *Humble J., Farley D.* Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. — Upper Saddle River, NJ: Addison-Wesley, 2010.
- *Hunt T.* Passwords Evolved: Authentication Guidance for the Modern Era // troyhunt.com, July 26, 2017. https://oreil.ly/r4ava.
- *Ingles P.* Convergence to Kubernetes // Medium, June 18, 2018. https://oreil.ly/oB2FI.
- *Ishmael J.* Optimising Serverless for BBC Online // Technology and Creativity at the BBC (blog), BBC, January 26, 2021. https://oreil.ly/mPp2L.
- *Jackson C.* Micro Frontends // martinfowler.com, June 19, 2019. https://oreil.ly/nYu15.
- *Kingsbury K.* Jepsen: Elasticsearch // Aphyr, June 15, 2014. https://oreil.ly/6l2sR.
- *Kingsbury K.* Jepsen: Elasticsearch 1.5.0 // Aphyr, April 27, 2015. https://oreil.ly/jlu8p.
- *Krishnan K.* Weathering the Unexpected // acmqueue 10, no. 9 (2012). https://oreil.ly/BN2Ek.
- *Kubis R.* Google Cloud Spanner: Global Consistency at Scale by Robert Kubis // Devoxx. November 7, 2017. YouTube video, 33:22. https://oreil.ly/fwbMD.
- *Lamport L.* Time, Clocks and the Ordering of Events in a Distributed System // Communications of the ACM. 21, no. 7 (July 1978): 558–65. https://oreil.ly/Y07gU.
- *Lewis J.* Scale, Microservices and Flow // YOW! Conferences. February 10, 2020. YouTube video, 51:03. https://oreil.ly/nzXqX.
- *Losio R.* Elastic Changes Licences for Elasticsearch and Kibana: AWS Forks Both // InfoQ, January 25, 2021. https://oreil.ly/PClFv.
- *MacCormack A., Baldwin C. Y., Rusnak J.* Exploring the Duality Between Product and Organizational Architectures: A Test of the Mirroring Hypothesis // Research Policy 41, no. 8 (October 2012): 1309–24.
- *Majors C.* Metrics: Not the Observability Droids You're Looking For // Honeycomb (blog), October 24, 2017. https://oreil.ly/RpZaZ.

- *Majors C., Fong-Jones L., Miranda G.* Observability Engineering. — Sebastopol: O'Reilly, 2022.
- *McAllister N.* Code Spaces Goes Titsup FOREVER After Attacker NUKES Its Amazon-Hosted Data // The Register, June 18, 2014. https://oreil.ly/IUOD0.
- *Miles R.* Learning Chaos Engineering. — Sebastopol: O'Reilly, 2019.
- *Moore J.* Architecture with 800 of My Closest Friends: The Evolution of Comcast's Architecture Guild // InfoQ, May 14, 2019. https://oreil.ly/dVfhi.
- *Morris K.* Infrastructure as Code. 2nd ed. — Sebastopol: O'Reilly, 2016.
- *Nagappan N., Murphy B., Basili V.* The Influence of Organizational Structure on Software Quality: An Empirical Case Study // ICSE '08: Proceedings of the 30th International Conference on Software Engineering. — New York: ACM, 2008.
- *Newman S.* Monolith to Microservices. — Sebastopol: O'Reilly, 2019.
- *Noursalehi S.* Git Virtual File System Design History. https://t.co/mIQR4uzWKS?amp=1.
- *Nygard M.* Release It! 2nd ed. — Raleigh: Pragmatic Bookshelf, 2018.
- *Oberlehner M.* Monorepos in the Wild // Medium, June 12, 2017. https://oreil.ly/Sk6am.
- *Padmanabhan S., Jha P.* WebAssembly at eBay: A Real-World Use Case // eBay, May 22, 2019. https://oreil.ly/rlr7d.
- *Page-Jones M.* Practical Guide to Structured Systems Design, 2nd ed. — New York: Yourdon Press, 1980.
- *Parnas D.* Information Distribution Aspects of Design Methodology // Information Processing: Proceedings of the IFIP Congress, 339–44. Vol. 1. — Amsterdam: North Holland, 1972.
- *Parnas D.* On the Criteria to Be Used in Decomposing Systems into Modules // Journal contribution, Carnegie Mellon University, 1971. https://oreil.ly/nWtQA.
- *Plotnicki L.* BFF @ Soundcloud // ThoughtWorks, December 9, 2015. https://oreil.ly/ZyR0l.
- *Potvin R., Levenberg J.* Why Google Stores Billions of Lines of Code in a Single Repository // Communications of the ACM 59, no. 7 (July 2016): 78–87. https://oreil.ly/Eupyi.
- *Pyhajarvi M.* Ensemble Programming Guidebook. Self-published, 2015.2020. https://ensembleprogramming.xyz.
- *Riggins J.* The Rise of Progressive Delivery for Systems Resilience // The New Stack, April 1, 2019. https://oreil.ly/merIs.
- *Rodriguez D., Sicilia M. A., Barriocanal E. G., Harrison R.* Empirical Findings on Team Size and Productivity in Software Development // Journal of Systems and Software 85, no. 3 (2012). doi.org/10.1016/j.jss.2011.09.009.

- *Rossman J*. Think Like Amazon: 50 1/2 Ideas to Become a Digital Leader. — New York: McGraw-Hill, 2019.
- *Ruecker B*. Practical Process Automation. — Sebastopol: O'Reilly, 2021.
- *Sadalage P., Fowler M*. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. — Upper Saddle River, NJ: Addison-Wesley, 2012.
- *Schneider J*. Understanding Design Thinking, Lean and Agile. — Sebastopol: O'Reilly, 2017.
- *Shankland S*. Google Uncloaks Once-Secret Server // CNET, December 11, 2009. https://oreil.ly/hHKvE.
- *Shorrock S*. Alarm Design: From Nuclear Power to WebOps // Humanistic Systems (blog), October 16, 2015. https://oreil.ly/AiJ5i.
- *Shostack A*. Threat Modeling: Designing for Security. — Indianapolis: Wiley, 2014.
- *Sigelman B*. Three Pillars with Zero Answers — Towards a New Scorecard for Observability // Lightstep (blog post), December 5, 2018. https://oreil.ly/qdtSS.
- *Skelton M., Pais M*. Team Topologies. — Portland, OR: IT Revolution, 2019.
- *Steen M. van, Tanenbaum A*. Distributed Systems. 3rd ed. — Scotts Valley, CA: CreateSpace Independent Publishing Platform, 2017.
- *Stopford, Ben*. Designing Event-Driven Systems. — Sebastopol: O'Reilly, 2017.
- *Valentino J. D*. Moving One of Capital One's Largest Customer-Facing Apps to AWS // Medium/Capital One Tech, May 24, 2017. https://oreil.ly/IEIC3.
- *Vaughan D*. The Challenger Launch Decision: Risky Technology, Culture, and Deviance at NASA. — Chicago: University of Chicago Press, 1996.
- *Vocke H*. The Practical Test Pyramid // martinfowler.com, February 26, 2018. https://oreil.ly/6rRoU.
- *Webber E*. Building Successful Communities of Practice. — San Francisco: Blurb, 2016.
- *Webber J., Parastatidis S., Robinson I*. REST in Practice: Hypermedia and Systems Architecture. — Sebastopol: O'Reilly, 2010.
- *Woods D. D*. Four Concepts for Resilience and the Implications for the Future of Resilience Engineering // Reliability Engineering & System Safety 141 (September 2015): 5–9. doi.org/10.1016/j.ress.2015.03.018.
- *Yourdon E., Larry L*. Constantine. Structured Design. — New York: Yourdon Press, 1976.
- *Zimman A*. Progressive Delivery, a History... Condensed // Industry Insights (blog). LaunchDarkly, August 6, 2018. https://oreil.ly/4pVY7.

Глоссарий

Amazon Web Services (AWS) — публичное облачное предложение от Amazon.

API-шлюз — компонент, который обычно находится по периметру системы и, помимо всего прочего, направляет вызовы из внешних источников (таких как пользовательские интерфейсы) к микросервисам.

Azure — публичное облачное предложение от Microsoft.

Docker — набор инструментов, помогающих создавать контейнеры и управлять ими.

GraphQL — протокол, позволяющий клиенту выдавать пользовательские запросы, способные вызывать несколько нижестоящих микросервисов. Облегчает объединение вызовов и фильтрацию для внешних клиентов, не требует использования BFF или API-шлюзов.

Kubernetes — платформа с открытым исходным кодом, управляющая рабочими нагрузками контейнера на нескольких базовых машинах.

Автоматический выключатель — механизм, размещенный в точках подключения к нижестоящему сервису. Этот механизм позволит системе быстро выйти из строя, если нижестоящий сервис испытывает проблемы.

Авторизация — процесс, который определяет, разрешен ли авторизованному принципалу доступ к данной части функциональности.

Агрегат — набор объектов, управляемых как единое целое, обычно ссылающихся на концепции реального мира. Концепция от DDD.

Аутентификация — процесс, посредством которого принципал доказывает, что он тот, за кого себя выдает. Он может быть организован просто в виде ввода пользователем имени и пароля.

Бессерверный — обобщающий термин для облачных продуктов, которые, с точки зрения пользователя, абстрагируются от базовых компьютеров до такой степени, что пользователю больше не нужно заботиться о них. Примерами таких продуктов могут служить AWS Lambda, AWS S3 и Azure Cosmos.

Библиотека — набор кода, который упакован таким образом, что его можно повторно использовать в нескольких программах.

Брокер сообщений — специальное ПО, управляющее асинхронной связью между процессами, обычно предоставляя такие возможности, как гарантированная доставка.

Бэкенд для фронтенда (BFF) — компонент на стороне сервера, который обеспечивает агрегацию и фильтрацию для определенного пользовательского интерфейса. Альтернатива API-шлюзу общего назначения.

Бюджет ошибок — допустимый уровень отказов для сервиса за определенный период.

Веб-токены JSON — стандарт для создания структуры данных JSON, которая может быть дополнительно зашифрована. Обычно он используется для передачи информации об аутентифицированных участниках.

Вертикальное масштабирование — увеличение масштаба системы за счет применения более мощной машины.

Ветвление функций — создание новой ветви для каждой функции, над которой ведется работа, и по завершении разработки объединение этой ветви с основной. Я не одобряю этот подход.

Виджет — компонент графического пользовательского интерфейса.

Виртуальная машина (ВМ) — в этой эмуляции машина представлена в виде выделенной физической машины, выполняющей все необходимые цели и задачи.

Горизонтальное дублирование — масштабирование за счет создания копий элементов системы.

Детективный контроль — элемент управления безопасностью, который поможет определить, ведется ли атака/произошла ли она.

Единый язык — общий язык, определенный и принятый для использования в коде и при описании предметной области с целью облегчения коммуникации. Концепция от DDD.

Закон Конвея — заключается в том, что коммуникационные структуры организаций в конечном счете определяют проект компьютерных систем, создаваемых этими организациями.

Запрос отправляется одним микросервисом другому с просьбой к нижестоящему микросервису что-то сделать.

Идемпотентность — свойство функции, при котором, даже если она вызывается несколько раз, результат остается неизменным. Полезно для безопасного повторного выполнения операций с микросервисами.

Инфраструктура как код — моделирование вашей инфраструктуры в виде кода, позволяющее автоматизировать управление ею и контролировать версии кода.

Коллективное владение — стиль владения, при котором любому разработчику разрешается изменять любую часть системы.

Команда поддержки — помогает работе потоковых команд. Как правило, у команды поддержки особая направленность, например работа над удобством использования системы, улучшение архитектуры, повышение безопасности.

Контейнер — пакет кода и зависимостей, который может быть запущен изолированным образом на компьютере. Концептуально похож на виртуальные машины, хотя и гораздо более легкий.

Кросс-функциональное требование (CFR) — общее свойство системы, такое как требуемая задержка для операций, безопасность данных в состоянии покоя и т. д. Также известно как нефункциональное требование.

Личная информация (PII) — данные, которые при использовании отдельно или в дополнение к другой информации позволяют идентифицировать личность.

Магистральная разработка — стиль разработки, при котором все изменения вносятся непосредственно в основную магистраль системы управления версиями, включая изменения, которые еще не завершены.

Микросервис — независимо развертываемый сервис, который взаимодействует с другими микросервисами через один или несколько протоколов связи.

Моделирование угроз — процесс, в ходе которого вы понимаете угрозы, способные возникнуть в вашей системе, и определяете, какие из них имеют наибольший приоритет и требуют первоочередной обработки.

Монорепозиторий — единый репозиторий, содержащий весь исходный код для всех ваших микросервисов.

Мультирепозиторий — отдельный репозиторий для хранения исходного кода каждого микросервиса.

Надежность — способность системы продолжать работать, даже когда происходит что-то плохое.

Настраиваемые готовые программные продукты (COTS) — стороннее ПО, которое в значительной степени используется конечными пользователями, а также обычно запускается в их собственной инфраструктуре. Типичными примерами могут служить системы управления контентом и платформы управления взаимоотношениями с клиентами.

Независимое развертывание — возможность внести изменения в микросервис и развернуть его в рабочей среде без необходимости изменять или развертывать что-либо еще.

Непрерывная адаптивность — способность постоянно адаптироваться к меняющимся условиям, заинтересованным сторонам и требованиям.

Непрерывная интеграция (CI) — регулярная (ежедневная) интеграция изменений с остальной частью кодовой базы, а также набор тестов для проверки того, что интеграция сработала.

Непрерывная доставка (CD) — подход к доставке, при котором вы четко моделируете путь сервиса в эксплуатацию, рассматриваете каждую точку фиксации кода в качестве кандидата на релиз и можете легко оценить пригодность любого кандидата для развертывания в эксплуатации.

Непрерывное развертывание — подход, при котором любая сборка, прошедшая все автоматизированные этапы, автоматически развертывается в рабочей среде.

Ограниченный контекст — четкая граница внутри предметной области бизнеса, которая обеспечивает функциональность более широкой системы, но

также скрывает сложность. Часто соответствует границам организации. Концепция от DDD.

Одностраничные приложения (SPA) — тип графического пользовательского интерфейса, в котором пользовательский интерфейс предоставляется на одной панели браузера без необходимости перехода на другие веб-страницы.

Оркестрация — стиль саги, в котором центральное устройство (также известное как оркестратор) управляет работой других микросервисов для выполнения бизнес-процесса.

Ответ — то, что передается обратно в результате запроса.

Переборки — часть системы, в которой сбой может быть изолирован, так что остальная часть системы способна продолжать работать.

Показатели уровня качества обслуживания (SLI) — четко определенное числовое значение конкретной характеристики сервиса, например времени отклика.

Потоковая команда — команда, сосредоточенная на сквозном выполнении значимого потока работ. Это долгофункционирующая команда, которая, как правило, ориентирована непосредственно на клиента и работает с данными, бэкенд- и фронтенд-кодом.

Поэтапное развертывание — необходимость развертывания двух или более объектов одновременно, поскольку произошло изменение, требующее этого. Противоположность возможности независимого развертывания.

Превентивный контроль — контроль безопасности, целью которого является предотвращение атаки.

Предметная связанность — форма связи, при которой один микросервис «связан» с доменным протоколом, предоставляемым другим микросервисом.

Предметно-ориентированное проектирование (DDD) — концепция, при которой фундаментальная проблема/бизнес-область явно моделируется в программном обеспечении.

Принципал — человек (хотя также может быть программа), который запрашивает аутентификацию и авторизацию для получения доступа.

Разделение данных — масштабирование системы путем распределения нагрузки на основе некоторых аспектов данных. Например, разделение нагрузки в зависимости от клиента или типа продукта.

Реагирующий контроль — элемент управления безопасностью, который помогает реагировать во время/после атаки.

Сага — способ моделирования существующих продолжительное время операций таким образом, чтобы ресурсы не нужно было блокировать на длительные периоды времени. Предпочтительнее распределенных транзакций при реализации бизнес-процессов.

Связанность — степень, в которой изменение одной части системы требует изменения в другой. Обычно желательна слабая связанность.

Связность — степень, в которой изменяемый код остается цельным.

Сервисные сети — распределенный тип промежуточного ПО, который обеспечивает сквозную функциональность в первую очередь для синхронных вызовов «точка — точка» — например, помогает реализовать двухсторонний протокол TLS, обнаружение сервисов или автоматические выключатели.

Сервис-ориентированная архитектура (SOA) — тип архитектуры, при котором система разбита на сервисы, которые могут выполняться на разных машинах. Микросервисы — это тип SOA, в котором приоритет отдается независимому развертыванию.

Сильное владение — стиль владения, при котором части системы принадлежат определенным командам, а изменения в определенную часть системы может вносить только та команда, которой она принадлежит.

Скрытие информации — подход, при котором вся информация по умолчанию скрыта внутри границ, и только абсолютный минимум доступен внешним потребителям.

Событие — что-то, что происходит в системе, что могут обработать другие ее части, — например, «Заказ размещен» или «Пользователь входит в систему».

Соглашение об уровне предоставления услуги (SLA) — соглашение между конечным пользователем и поставщиком услуг (например, между покупателем и поставщиком), содержащее описание услуги, права и обязанности сторон и, самое главное, согласованный уровень качества предоставления данной услуги.

Сообщение — что-то отправленное в один или несколько нисходящих микросервисов через механизм асинхронной связи, такой как брокер. Может содержать полезную нагрузку, например запрос, ответ или событие.

Стабильная расширяемость — показатель того, насколько хорошо мы справляемся с неожиданной ситуацией.

Управление — согласование того, как все должно быть сделано, и обеспечение того, чтобы все было сделано именно так.

Функция как услуга (FaaS) — тип бессерверной платформы, которая вызывает произвольный код на основе определенных типов триггеров — например, запуск кода в ответ на HTTP-вызов или полученное сообщение.

Хореография — стиль саги, где ответственность за то, что должно произойти и когда, распределяется между несколькими микросервисами, а не управляется одним объектом.

Целевой уровень качества обслуживания (SLO) — это количественная оценка работы сервиса, целевые показатели совокупного успеха SLI в течение определенного периода времени.

Об авторе

Сэм Ньюмен — независимый консультант, автор и докладчик. За более чем 20 лет работы в отрасли он имел дело с различными технологическими стеками в разных областях и компаниях по всему миру. В основном он помогает организациям быстрее и безопаснее внедрять ПО в эксплуатацию, а также ориентироваться в сложностях микросервисов. Автор книги «От монолита к микросервисам».

Иллюстрация на обложке

На обложке второго издания изображены медоносные пчелы (из рода *Apis*). Из более чем 20 000 видов пчел только восемь — медоносные. Эти пчелы уникальны тем, что коллективно производят и хранят мед, а также строят ульи из воска. Пчеловодство для сбора меда было занятием людей по всему миру на протяжении тысячелетий.

В ульях медоносных пчел живут тысячи особей, у которых очень организованная социальная структура: есть королева, трутни и рабочие пчелы. В каждом улье есть одна матка, которая остается фертильной в течение 3–5 лет после своего брачного полета и откладывает до 2000 яиц в день. Трутни — это самцы пчел, которые спариваются с маткой. Рабочие пчелы — стерильные самки, которые в течение своей жизни выполняют множество ролей, таких как няня, строитель, бакалейщик, охранник, гробовщик и фуражир. Нагруженные пыльцой рабочие пчелы, возвращающиеся в улей, «танцуют» по определенным схемам, чтобы сообщить информацию о близлежащих источниках пищи своим сородичам.

Все медоносные пчелы похожи между собой: у них прозрачные крылья, шесть ног и тело, разделенное на голову, грудку и желто-черное брюшко, покрытые короткими пушистыми волосками. Рацион взрослых особей состоит исключительно из меда, который образуется в результате частичного переваривания, а затем срыгивания богатого сахаром цветочного нектара.

Пчелы играют решающую роль в сельском хозяйстве — собирая пищу, они опыляют посевы. Коммерческие пчелиные ульи перевозятся пчеловодами туда, где необходимо опылять сельскохозяйственные культуры. В среднем каждый пчелиный улей собирает 30 килограммов пыльцы в год. Однако в последние годы распад колоний, вызванный различными заболеваниями и другими причинами, поспособствовал сокращению численности медоносных пчел.

Многие животные, изображенные на обложках O'Reilly, находятся под угрозой исчезновения, а ведь все они важны для мира.

Цветная иллюстрация на обложке выполнена Карен Монтгомери по мотивам черно-белой гравюры из Музея живой природы.

Сэм Ньюмен

Создание микросервисов

2-е издание

Перевел с английского С. Черников

Руководитель дивизиона *Ю. Сергиенко*
Ведущий редактор *Н. Гринчик*
Литературный редактор *А. Аверьянов*
Художественный редактор *В. Мостипан*
Корректоры *Е. Павлович, Е. Рафалюк-Бузовская*
Верстка *Г. Блинов*

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: 05.2023. 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 01.03.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 50,310. Тираж 1000. Заказ 0000.

Том Лащевски, Камаль Арора, Эрик Фарр, Пийум Зонуз

ОБЛАЧНЫЕ АРХИТЕКТУРЫ: РАЗРАБОТКА УСТОЙЧИВЫХ И ЭКОНОМИЧНЫХ ОБЛАЧНЫХ ПРИЛОЖЕНИЙ

Облачные вычисления — это, пожалуй, наиболее революционная разработка в IT со времен виртуализации. Облачно-ориентированные архитектуры обеспечивают большую гибкость по сравнению с системами предыдущего поколения. В этой книге продемонстрированы три важнейших аспекта развертывания современных cloud native архитектур: организационное преобразование, модернизация развертывания, паттерны облачного проектирования.

Книга начинается с краткого знакомства с облачно-ориентированными архитектурами — на примерах объясняется, какие черты им присущи, а какие нет. Вы узнаете, как организуется внедрение и разработка облачных архитектур с применением микросервисов и бессерверных вычислений как основ проектирования. Далее вы изучите такие столпы облачно-ориентированного проектирования, как масштабируемость, оптимизация издержек, безопасность и способы достижения безупречной эксплуатационной надежности. В заключительных главах будет рассказано о различных общедоступных архитектурах cloud native — от AWS и Azure до Google Cloud Platform.

КУПИТЬ

Митч Сеймур

KAFKA STREAMS И KSQLDB: ДАННЫЕ В РЕАЛЬНОМ ВРЕМЕНИ

Работа с неограниченными и быстрыми потоками данных всегда была сложной задачей. Но Kafka Streams и ksqlDB позволяют легко и просто создавать приложения потоковой обработки. Из книги специалисты по обработке данных узнают, как с помощью этих инструментов создавать масштабируемые приложения потоковой обработки, перемещающие, обогащающие и преобразующие большие объемы данных в режиме реального времени.

Митч Сеймур, инженер службы обработки данных в Mailchimp, объясняет важные понятия потоковой обработки на примере нескольких любопытных бизнес-задач. Он рассказывает о достоинствах Kafka Streams и ksqlDB, чтобы помочь вам выбрать наиболее подходящий инструмент для каждого уникального проекта потоковой обработки. Для разработчиков, не пишущих код на Java, особенно ценным будет материал, посвященный ksqlDB.

КУПИТЬ