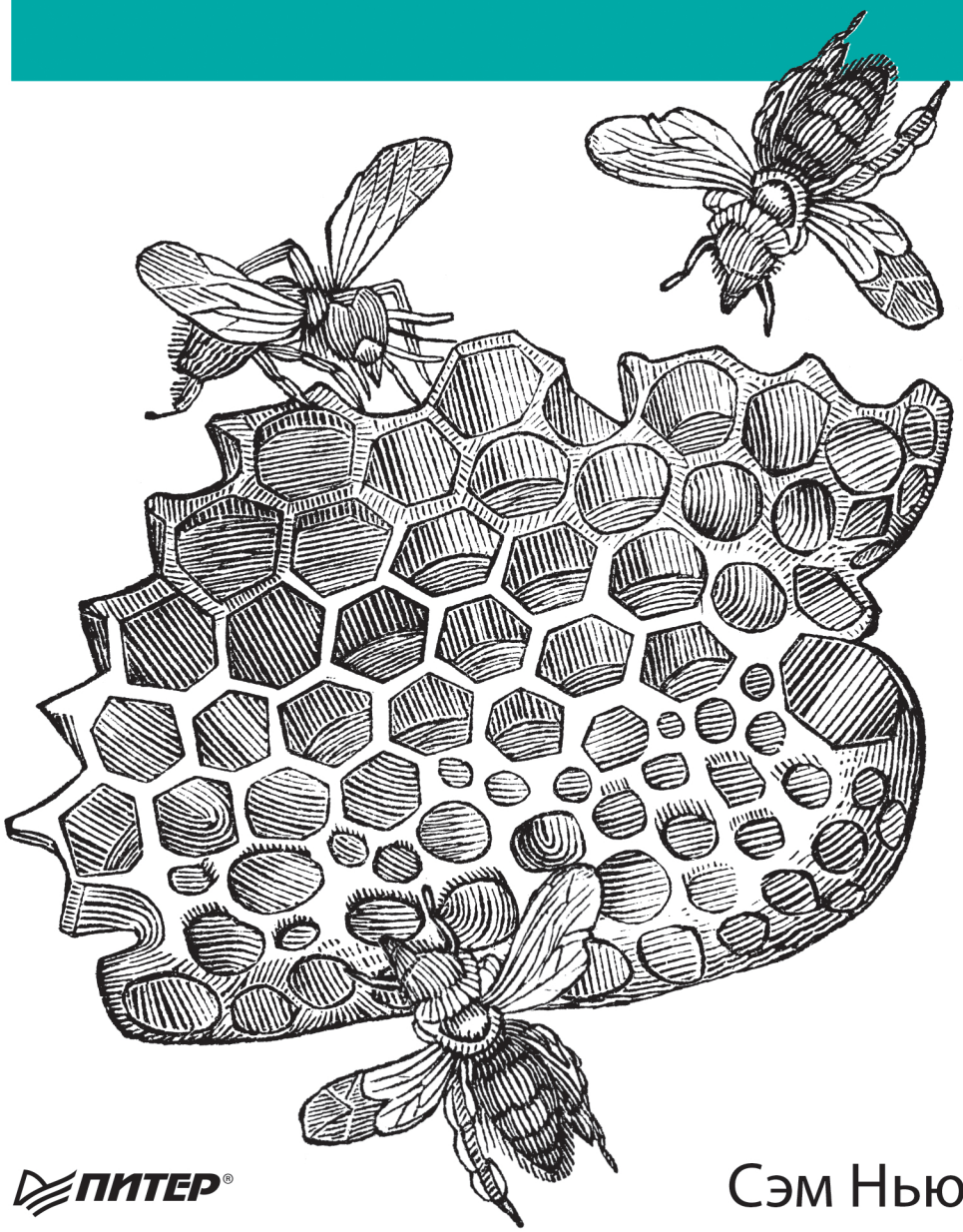
O'REILLY®

СОЗДАНИЕ МИКРОСЕРВИСОВ



ПИТЕР®

Сэм Ньюмен

Building Microservices

Sam Newman

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Сэм Ньюмен

СОЗДАНИЕ МИКРОСЕРВИСОВ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Киев · Екатеринбург · Самара · Минск

2016

ББК 32.988.02-018
УДК 004.438.5
Н93

Ньюмен С.

Н93 Создание микросервисов. — СПб.: Питер, 2016. — 304 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-02011-4

Книга посвящена программированию микросервисов — небольших автономных компонентов, позволяющих добиться модульности и отказоустойчивости любой программы. Теория микросервисов тесно связана с философией Unix, способствует улучшению архитектуры любых приложений, дает возможность избегать громоздкого и запутанного кода. Эта книга поможет читателю заново взглянуть на многие, казалось бы, трудноразрешимые проблемы, масштабировать любые проекты, ювелирно разрабатывать даже самые сложные системы.

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

ББК 32.988.02-018
УДК 004.438.5

ISBN 978-1491950357 англ.

ISBN 978-5-496-02011-4

Краткое содержание

Оглавление

Предисловие

Микросервисы представляют собой такой подход к распределенным системам, при котором поддерживается использование сотрудничающих друг с другом сервисов с высокой степенью детализации и с собственными жизненными циклами. Поскольку моделирование микросервисов ведется преимущественно вокруг бизнес-областей, они позволяют избегать проблем, проявляющихся в традиционных архитектурах, имеющих высокий уровень связанности компонентов. В микросервисах также интегрируются новые технологии, появившиеся в последнее десятилетие, что помогает им обходить те подводные камни, которые возникают при реализации многих сервис-ориентированных архитектур.

Эта книга полна конкретных примеров использования микросервисов, собранных по всему миру, включая их применение в таких организациях, как Netflix, Amazon, Gilt и REA group, пришедших к мысли, что возросшая автономность этой архитектуры дает их командам огромные преимущества.

Для кого написана эта книга

Тематика книги довольно обширна, что, собственно, соответствует масштабам последствий применения архитектур микросервисов, обладающих высокой степенью детализации. Книга предназначена для тех, кто интересуется аспектами проектирования, разработки, развертывания, тестирования и обслуживания систем. А те, кто уже приступил к работе с архитектурами, обладающими высокой степенью детализации, создавая приложения с нуля или разбивая на более мелкие части существующую монолитную систему, найдут в ней массу полезных практических советов. Книга поможет и тем, кто хочет узнать, в чем тут дело, чтобы понять, подходят им микросервисы или нет.

Зачем я написал эту книгу

Тема прикладных архитектур заинтересовала меня много лет назад, когда я работал над тем, чтобы помочь людям ускорить установку их программных продуктов. Я понял, что если в фундаментальной конструкции системы отсутствует возможность легкого внесения изменений, то помощь технологий, позволяющих автоматизировать

инфраструктуру, тестирование и непрерывную доставку, будет носить весьма ограниченный характер.

В то же самое время для достижения подобных целей, а также для повышения возможностей масштабирования, степени автономности команд разработчиков или более эффективного внедрения новых технологий эксперименты по созданию архитектур с высокой степенью детализации проводились многими организациями. Мой собственный опыт, а также опыт коллег по ThoughtWorks и другим организациям позволил подтвердить тот факт, что использование большого количества сервисов с собственными независимыми жизненными циклами создает очень много проблем, требующих решения. Во многих отношениях эта книга представлялась мне своеобразной службой одного окна, помогающей охватить самые разные темы, необходимые для понимания микросервисов, — все, что могло бы существенно помочь мне в былые времена!

Мир микросервисов сегодня

Микросервисы — весьма динамичная тема. Хоть идея и не нова (несмотря на новизну применяемого термина), накопленный по всему миру опыт наряду с новыми технологиями существенно повлияли на сами способы ее использования. Из-за высокого темпа перемен в данной книге я старался сконцентрироваться скорее на идеях, чем на конкретных технологиях, зная, что подробности реализации всегда изменяются быстрее, чем положенные в их основу размышления. И тем не менее я полон ожиданий, что буквально через несколько лет мы еще глубже вникнем в суть областей применения микросервисов и порядок их эффективного использования.

Хотя я приложил все свои силы к тому, чтобы передать в этой книге основную суть затронутой тематики, если у вас появился к ней серьезный интерес и вы хотите быть в курсе самых последний веяний, будьте готовы к тому, чтобы посвятить ее изучению не один год.

Структура книги

Основа книги по большей части тематическая. Поэтому вы можете произвольно выбирать для изучения темы, которые представляют для вас наибольший интерес. Хотя основные положения и идеи я постарался раскрыть в начальных главах, хочется верить в то, что даже достаточно искушенный читатель найдет для себя что-нибудь интересное во всех главах без исключения. Если же вы хотите углубиться в изучение некоторых более поздних глав, я настоятельно рекомендую просмотреть главу 2, касающуюся обширного характера самой темы, а также дающую представление о принятой мною структуре изложения материала.

Для новичков в данной области главы выстроены таким образом, что им, как я надеюсь, будет иметь смысл прочесть всю книгу от начала до конца.

Вот краткий обзор всего, что рассматривается в данной книге.

- *Глава 1. Микросервисы*. Начинается с введения в микросервисы с указанием как их преимуществ, так и некоторых недостатков.

- *Глава 2. Архитектор развития.* Посвящена трудностям компромиссов в архитектурах и многообразию всего, что нужно осмыслить при использовании микросервисов.
- *Глава 3. Как моделировать сервисы.* Начинается с определения границ микросервисов с использованием в качестве вспомогательных средств, направляющих мысли в нужное русло, технологий, позаимствованных из проектирования, основанного на областях применения.
- *Глава 4. Интеграция.* В этой главе начинается погружение в конкретные технологические последствия по мере рассмотрения наиболее подходящих нам разновидностей технологий обеспечения совместной работы сервисов. В ней также более глубоко рассматривается тема пользовательских интерфейсов и интеграции с устаревшими и уже готовыми коммерческими программными средствами (COTS-продуктами).
- *Глава 5. Разбиение монолита на части.* Многие специалисты рассматривают микросервисы в качестве своеобразного антидота от крупных, слабо поддающихся изменениям монолитных систем. Именно этот вопрос и будет подробно рассмотрен в данной главе.
- *Глава 6. Развертывание.* Хотя данная книга носит преимущественно теоретический характер, некоторые темы в ней были подняты под влиянием последних изменений в таких технологиях, как развертывание, которое и станет предметом рассмотрения.
- *Глава 7. Тестирование.* Данная глава посвящена углубленному рассмотрению темы тестирования, к которой следует отнестись с повышенным вниманием, когда речь зайдет о развертывании нескольких отдельных сервисов. Особо будет отмечена роль, которую в содействии обеспечению качества программных средств могут сыграть контракты, основанные на запросах потребителей.
- *Глава 8. Мониторинг.* Тестирование программного средства перед развертыванием не помогает, если проблемы обнаруживаются во время его работы в производственном режиме, поэтому в данной главе исследуются возможности мониторинга систем, обладающих высокой степенью детализации, и методы, позволяющие справиться со сложностями, присущими распределенным системам.
- *Глава 9. Безопасность.* В данной главе исследуются аспекты безопасности микросервисов и рассматриваются методы, позволяющие выполнять аутентификацию и авторизацию пользователя по отношению к сервису и сервиса — по отношению к другому сервису. Безопасность в вычислительных системах является весьма важной темой, однако многие ее охотно игнорируют. Хотя я ни в коем случае не считаю себя специалистом в области безопасности, но все же надеюсь, что эта глава поможет вам по крайней мере обдумать некоторые аспекты, о которых нужно знать при построении систем, в частности систем на основе микросервисов.
- *Глава 10. Закон Конвея и проектирование систем.* Основное внимание в данной главе уделяется взаимодействию организационной структуры и архитектуры. Многие организации уже поняли, что если не добиваться в этом вопросе гармонии, то возникнут существенные затруднения. Мы попытаемся добраться

до самых глубин этой дилеммы и рассмотрим несколько различных способов увязки проектирования системы со структурой команд.

- *Глава 11. Масштабирование микросервисов.* В данной главе исследуется порядок всех предыдущих действий в условиях расширения системы, позволяющий справиться с постоянно возрастающими вероятностями сбоев, возникающих в условиях использования большого количества сервисов, а также при больших объемах трафика.
- *Глава 12. Коротко обо всем.* В заключительной главе предпринимается попытка выделить основные черты, отличающие микросервисы от всего остального. В ней перечислены семь принципов микросервисов, а также подведены итоги по ключевым моментам книги.

Соглашения, принятые в этой книге

В тексте книги действуют следующие типографские соглашения:

Курсив

Служит признаком новых понятий.

`Моноширинный шрифт`

Используется для листингов программ, а также для ссылок на программные элементы в виде переменных или названий функций, баз данных, типов данных, переменных среды, инструкций и ключевых слов внутри абзацев.

`Моноширинный полужирный шрифт`

Служит для выделения команд или другого текста, который должен быть набран самим пользователем.

`Моноширинный курсивный шрифт`

Показывает текст, который должен быть заменен предоставляемыми пользователем значениями или значениями, определяемыми контекстом.

Благодарности

Эта книга посвящается Линде Стивенс (Lindy Stephens), без которой книги бы просто не существовало. Она воодушевила меня начать этот путь, поддерживала в течение всего полного стрессовых ситуаций периода написания книги и проявила себя партнером, лучше которого я бы не смог себе даже пожелать. Мне хотелось бы посвятить эту книгу также своему отцу, Говарду Ньюману (Howard Newman), который всегда был рядом со мной. Эта книга посвящается им обоим.

Мне хотелось бы выделить Бена Кристенсена (Ben Christensen), Вивека Субраманьяма (Vivek Subramaniam) и Мартина Фаулера (Martin Fowler) за подробные отзывы в процессе написания книги и оказание помощи в формировании ее облика. Мне также хочется поблагодарить Джеймса Льюиса (James Lewis), с которым

мы выпили немало пива, обсуждая идеи, представленные в данной книге. Без их помощи и консультаций эта книга представляла бы жалкую тень самой себя.

Кроме того, помощь и отзывы о ранних версиях книги поступали и от многих других людей. Особенно хочется поблагодарить (не придерживаясь какого-либо определенного порядка) Кейна Венейблса (Kane Venables), Ананда Кришнасвами (Anand Krishnaswamy), Кента Макнила (Kent McNeil), Чарльза Хайнса (Charles Haynes), Криса Форда (Chris Ford), Айди Льюиса (Aidy Lewis), Уилла Темза (Will Thames), Джона Ивса (Jon Eaves), Рольфа Рассела (Rolf Russell), Бадринатха Янакирамана (Badrinath Janakiraman), Даниэля Брайанта (Daniel Bryant), Яна Робинсона (Ian Robinson), Джима Уэббера (Jim Webber), Стюарта Глидоу (Stewart Gleadow), Эвана Ботчера (Evan Bottcher), Эрика Суорда (Eric Sword), Оливию Леонард (Olivia Leonard) и всех остальных коллег из ThoughtWorks и других организаций, которые помогли мне пройти этот длинный путь.

И в заключение я хочу поблагодарить всех специалистов издательства O'Reilly, включая Майка Лукидиса (Mike Loukides), за оказанное мне доверие, моего редактора Брайана Макдональда (Brian MacDonald), Рэйчел Монаган (Rachel Monaghan), Кристен Браун (Kristen Brown), Бетси Валишевски (Betsy Waliszewski) и всех других, кто помог мне в том, о чем я сам и не догадывался.

Об авторе

Сэм Ньюмен — инженер из компании ThoughtWorks, где в настоящее время совмещает работу над клиентскими проектами с решением архитектурных задач для внутренних систем ThoughtWorks. Сэму доводилось работать с компаниями всего мира в самых разных предметных областях, зачастую заниматься одновременно и разработкой, и поддержкой ПО. Если спросить Сэма, чем он занимается, то он ответит: «Работаю с людьми, чтобы создавать все более и более классные софтверные системы». Сэм Ньюмен — автор статей, докладов на конференциях, время от времени он участвует в разработке свободных проектов.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты sivchenko@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

1 Микросервисы

Поиски наилучших способов построения систем велись многие годы. Мы изучали истоки, внедряли новые технологии и наблюдали за тем, как технологические компании новой волны работают в разных направлениях, создавая IT-системы, радующие как клиентов, так и разработчиков.

Книга Эрика Эванса (Eric Evans) по предметно ориентированному проектированию Domain-Driven Design (Addison-Wesley) помогла нам осознать важность отображения в коде реального мира и показала более удачные способы моделирования наших систем. Концепция непрерывной поставки показала, как можно более эффективно и рационально внедрять продукты в производство, вселяя в нас идею о том, что завершением каждого рабочего этапа должен считаться выпуск предварительной версии продукта. Понимание порядка работы Всемирной сети привело нас к разработке более эффективных способов организации межмашинного общения. Концепция Алистера Кокберна (Alistair Cockburn) о гексагональной архитектуре увела нас от многоуровневых архитектур, в которых бизнес-логика может быть скрыта. Платформы виртуализации позволяют обеспечивать наши машины и изменять их размеры по своему усмотрению, а автоматизация инфраструктуры дает способ управления этими машинами на должном уровне. Многие крупные и успешные организации, такие как Amazon и Google, поддерживают намерение небольших команд владеть полным жизненным циклом своих сервисов. А совсем недавно компания Netflix поделилась с нами способами создания прочных (antifragile) систем в том масштабе, который трудно было себе даже представить еще 10 лет назад.

Предметно ориентированное проектирование. Непрерывная поставка. Виртуализация по требованию. Автоматизация инфраструктуры. Небольшие автономные команды. Масштабируемые системы. Микросервисы происходят из мира именно этих понятий. Они не были изобретены или озвучены до того, как в них возникла потребность, появившись в качестве тенденции или образца из практики реального мира. Но их существование обусловлено всем тем, что было создано до их появления. Выстраивая по ходу повествования картину создания и развития микросервисов, а также управления ими, я буду проводить параллели со всей предшествующей этому работой.

Многие организации уже пришли к выводу, что, используя совокупность разбитых на мелкие гранулы архитектур микросервисов, они могут ускорить поставку программного обеспечения и внедрить в практику самые новые технологии.

Микросервисы дают нам существенно больше свободы воздействия и принятия различных решений, позволяя быстрее реагировать на неизбежные изменения, касающиеся всех нас.

Что же такое микросервисы

Микросервисы — это небольшие, автономные, совместно работающие сервисы. Разберем это определение по частям и рассмотрим, что определяет отличительные черты микросервисов.

Небольшие и нацеленные на то, чтобы хорошо справляться только с одной работой

При создании кода дополнительных свойств программы разрастается и база программного кода. Со временем из-за слишком большого объема этой базы возникают затруднения при поиске тех мест, куда нужно вносить изменения. Несмотря на стремление к созданию понятных модульных монолитных баз кода, довольно часто эти произвольные, находящиеся в процессе становления границы нарушаются. Код, относящийся к схожим функциям, попадает в разные места, что усложняет устранение дефектов или реализацию функций.

Внутри монолитных систем мы стремимся бороться с этой тенденцией, пытаясь сделать свой код более связанным, зачастую путем создания абстракций или модулей. Придание связности означает стремление сгруппировать родственный код. Эта концепция приобретает особую важность при размышлении о микросервисах. Она усиливается определением, данным Робертом С. Мартином (Robert C. Martin) принципу единственной обязанности — Single Responsibility Principle, которое гласит: «Собирайте вместе все, что изменяется по одной и той же причине, и разделяйте все, что изменяется по разным причинам».

Точно такой же подход в сфере микросервисов используется в отношении независимых сервисов. Границы сервисов формируются на основе бизнес-границ, что позволяет со всей очевидностью определить местонахождение кода для заданной области выполняемых функций. Удерживая сервис в четко обозначенных границах, мы не позволяем себе мириться с его чрезмерным разрастанием со всеми вытекающими из этого трудностями.

Мне часто задают вопрос: а что является критерием понятия «небольшой»? Задание количества строк кода для этого вряд ли подойдет, поскольку одни языки выразительнее других и способны на большее при меньшем количестве строк. Нужно также считаться с тем фактом, что мы могли бы быть втянуты в ряд зависимостей, которые сами по себе содержат множество строк кода. Кроме того, некоторые составляющие вашей области деятельности могут быть сложными по определению и требовать немалого объема кода. Джон Ивс (Jon Eaves) из Австралии на сайте RealEstate.com.au охарактеризовал микросервисы как некий код, который может быть переписан за две недели, что имело вполне определенный смысл в качестве основного правила конкретно в его контексте.

Еще один несколько банальный ответ может прозвучать так: *достаточно небольшой и не меньше этого*. Выступая на конференциях, я почти всегда задаю вопрос: «У кого есть слишком большая система, которую хотелось бы разбить на части?» Руку поднимают практически все. Похоже, мы прекрасно понимаем, что может считаться слишком большим, и можно будет согласиться с тем, что тот фрагмент кода, который уже нельзя назвать большим, будет, наверное, достаточно небольшим.

Ответить на вопрос, насколько небольшим должен быть сервис, поможет вполне конкретное определение того, насколько хорошо он накладывается на структуры команд разработчиков. Если для управления небольшой командой база программного кода слишком велика, весьма разумно будет изыскать возможности ее разбиения на части. Разговор об организационных совпадениях мы продолжим чуть позже.

Когда решается вопрос о достаточности уменьшения объема кода, я предпочитаю размышлять в следующем ключе: чем меньше сервис, тем больше проявляются все преимущества и недостатки микросервисной архитектуры. Чем меньше делается сервис, тем больше становятся его преимущества в смысле взаимозависимости.

Но верно и то, что возникают некоторые осложнения из-за наличия все большего количества движущихся частей, и эта проблема также будет повсеместно исследоваться в данной книге. Как только вы научитесь справляться с подобными сложностями, можно будет направить свои силы на создание все меньших и меньших сервисов.

Автономные

Наш микросервис является самостоятельным образованием, которое может быть развернуто в качестве обособленного сервиса на платформе, предоставляемой в качестве услуги, — Platform as a Service (PAAS), или может быть процессом своей собственной операционной системы. Мы стараемся не заполнять несколькими сервисами одну и ту же машину, хотя определение *машины* в современном мире весьма размыто! Чуть позже мы разберемся в том, что, несмотря на издержки, которые могут быть вызваны обособленностью, получаемая в результате использования такого сервиса простота существенно облегчает рассуждения о распределенной системе, а самые последние технологии позволяют смягчить множество проблем, связанных с этой формой развертывания.

Обмен данными между самими сервисами ведется через сетевые вызовы, чтобы упрочить обособленность сервисов и избежать рисков, сопряженных с тесными связями.

Этим сервисам необходимо иметь возможность изменяться независимо друг от друга и развертываться, не требуя никаких изменений от потребителей. Нам нужно подумать о том, что именно наши сервисы будут показывать и что им можно будет скрывать. Если объем совместно используемого будет слишком велик, потребляемые сервисы станут завязываться на внутренние представления. Это снизит автономность, поскольку при внесении изменений потребует дополнительного согласования с потребителями.

Наши сервисы выставляют напоказ программный интерфейс приложения — Application Programming Interface (API), и работающие на нас сервисы связывают-

ся с нами через эти API. Нужно также подумать о технологии, позволяющей не привязывать сами интерфейсы к потребителям. Это может означать выбор API, нейтральных по отношению к тем или иным технологиям, чтобы гарантировать отсутствие ограничений в выборе технологий. На страницах этой книги мы еще не раз вернемся к вопросу о важности качественных, ни к чему не привязанных API.

Без отсутствия привязки наша работа теряет всякий смысл. Можете ли вы внести изменения в сервис и провести полностью самостоятельное его развертывание, не внося каких-либо изменений во что-нибудь еще? Если нет, то реализовать множество обсуждаемых в этой книге преимуществ будет крайне сложно.

Чтобы добиться качественной обособленности, нужно правильно смоделировать свои сервисы и получить на выходе подходящие API. На эту тему еще предстоит множество разговоров.

Основные преимущества

Микросервисы обладают множеством разнообразных преимуществ. Многие из них могут быть присущи любой распределенной системе. Но микросервисы нацелены на достижение вершин этих преимуществ, что обусловливается в первую очередь тем, насколько глубоко ими принимаются концепции, положенные в основу распределенных систем и сервис-ориентированной архитектуры.

Технологическая разнородность

Имея систему, составленную из нескольких совместно работающих сервисов, можно прийти к решению использовать внутри каждого из них различные технологии. Это позволит выбрать для каждого задания наиболее подходящий инструментарий, не выискивая какой-либо стандартный подход на все случаи жизни, зачастую заканчивающийся выбором наименьшего общего знаменателя.

Если какой-то части системы требуется более высокая производительность, можно принять решение по использованию иного набора технологий, более подходящего для достижения требуемого уровня производительности. Можно также принять решение об изменении способа хранения данных для разных частей нашей системы. Например, пользовательский обмен сообщениями в социальной сети можно хранить в графоориентированной базе данных, отражая тем самым присущую социальному графу высокую степень взаимосвязанности, а записи в блоге, создаваемые пользователями, можно хранить в документоориентированном хранилище данных, давая тем самым повод для использования разнородной архитектуры, похожей на ту, что показана на рис. 1.1.

Микросервисы также позволяют быстрее внедрять технологии и разбираться в том, чем именно нововведения могут нам помочь. Одним из величайших барьеров на пути принятия новой технологии является риск, связанный с ее использованием. Если при работе с монолитным приложением мне захочется попробовать новые язык программирования, базу данных или структуру, то влияние распространится на существенную часть моей системы. Когда система состоит из нескольких сервисов, у меня появляются несколько новых мест, где можно проверить работу

новой части технологии. Я могу выбрать такой сервис, с которым риск будет наименьшим, и воспользоваться предлагаемой им технологией, зная, что могу ограничить любое потенциально отрицательное воздействие. Возможность более быстрого внедрения новых технологий рассматривается многими организациями как существенное преимущество.

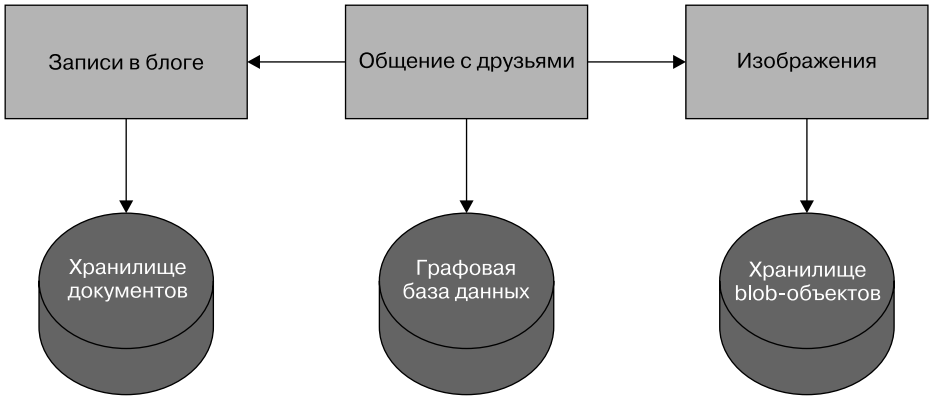


Рис. 1.1. Микросервисы позволяют упростить использование разнообразных технологий

Разумеется, использование нескольких технологий не обходится без определенных издержек. Некоторые организации предпочитают накладывать на выбор языков ограничения. В Netflix и Twitter, к примеру, в качестве платформы используется преимущественно Java Virtual Machine (JVM), поскольку они очень ценят надежность и производительность этой системы. В этих организациях также разрабатываются библиотеки и инструментальные средства для JVM, существенно упрощающие работу в масштабе платформы, но сильно усложняющие ее для сервисов и клиентов, не работающих на Java-платформе. Но для всех заданий ни в Twitter, ни в Netflix не используется лишь один набор технологий. Еще одним аргументом в противовес опасениям по поводу смешения различных технологий является размер. Если я действительно могу переписать свой микросервис за две недели, то вы также можете снизить риск применения новой технологии.

Читая книгу, вы постоянно будете приходить к мысли, что в применении микросервисов, как и во многом другом, нужно находить разумный баланс. Выбор технологий рассматривается в главе 2, там основное внимание будет уделено эволюционной архитектуре, а из главы 4, которая посвящена интеграции, вы узнаете, как можно обеспечить своим сервисам возможность развивать технологию независимо друг от друга без чрезмерной связанности.

Устойчивость

Ключевым понятием в технике устойчивости является перегородка. При отказе одного компонента системы, не вызывающем череду связанных с ним отказов, проблему можно изолировать, сохранив работоспособность всей остальной системы. Именно такими перегородками для вас станут границы сервисов. В монолитном

сервисе при его отказе прекращается вся работа. Работая с монолитной системой, можно уменьшить вероятность отказа, запустив систему сразу на нескольких машинах, но, работая с микросервисами, мы можем создавать такие системы, которые способны справиться с тотальными отказами сервисов и снизить соответствующим образом уровень их функциональных возможностей.

Но во всем нужно проявлять осторожность. Чтобы убедиться в том, что наши системы, составленные из микросервисов, могут воспользоваться улучшенной устойчивостью должным образом, нужно разобраться с новыми источниками отказов, с которыми должны справляться распределенные системы. Отказать могут как сети, так и машины, и такие отказы неизбежны. Нам нужно знать, как с этим справиться и как это может (или не может) повлиять на конечного пользователя нашего программного средства.

Устойчивость в ее лучших проявлениях и методы борьбы с отказами будут рассматриваться в главе 11.

Масштабирование

В больших монолитных сервисах расширять приходится все сразу. Одна небольшая часть всей системы может иметь ограниченную производительность, но, если в силу этого тормозится работа всего огромного монолитного приложения, расширять нужно все как единое целое. При работе с небольшими сервисами можно расширить только те из них, которые в этом нуждаются, что позволяет запускать другие части системы на небольшом, менее мощном оборудовании (рис. 1.2).

Именно по этой причине микросервисами воспользовалась компания Gilt, занимающаяся продажей через Интернет модной одежды. Начав в 2007 году с использования монолитного Rails-приложения, к 2009 году компания столкнулась с тем, что используемая система не в состоянии справиться с возлагаемой на нее нагрузкой. Разделив основные части своей системы, Gilt смогла намного успешнее справляться со всплесками трафика, и теперь компания использует 450 микросервисов, каждый из которых запущен на нескольких отдельных машинах.

При использовании систем предоставления услуг по требованию, подобных тем, которые предлагает Amazon Web Services, можно даже применить такое масштабирование по требованию для тех частей приложения, которые в этом нуждаются. Это позволит более эффективно контролировать расходы. Столь тесная корреляция архитектурного подхода и практически сразу же проявляющейся экономии средств наблюдается крайне редко.

Простота развертывания

Для реализации внесения изменений в одну строку монолитного приложения, состоящего из миллионов строк кода, требуется, чтобы было развернуто все приложение. Это развертывание может быть весьма рискованным и иметь крайне негативные последствия. На практике подобные рискованные развертывания из-за вполне понятных опасений происходят нечасто. К сожалению, это означает, что изменения копятся и копятся между выпусками, пока в производство не будет запущена новая

Рис. 1.2. Можно расширять только те микросервисы, которые в этом нуждаются

версия приложения, имеющая массу изменений. И чем больше будет разрыв между выпусками, тем выше риск, что что-нибудь будет сделано не так!

При использовании микросервисов можно вносить изменения в отдельный микросервис и развертывать его независимо от остальной системы. Это позволит развертывать код быстрее. Возникшую проблему можно быстро изолировать в рамках отдельного сервиса, упрощая тем самым быстрый откат. Это также означает, что новые функциональные возможности могут дойти до клиента быстрее. Именно то, что такая архитектура позволяет устранить максимально возможное количество препятствий для запуска приложения в эксплуатацию, и стало одной из основных причин, по которой такие организации, как Amazon и Netflix, воспользовались ею.

За последние пару лет технология в данной области сильно изменилась, и тема развертывания в мире микросервисов более глубоко будет рассмотрена в главе 6.

Решение организационных вопросов

У многих из нас имеется опыт решения проблем, связанных с большими командами разработчиков и объемными базами исходного кода. Если команда разбросана по разным местам, проблема может только усугубиться. Также общеизвестно, что небольшие команды, работающие с небольшим объемом исходного кода, как правило, показывают более высокую продуктивность.

Микросервисы позволяют эффективнее приспособить архитектуру к решению организационных вопросов, позволяя свести к минимуму число разработчиков каждого отдельно взятого фрагмента исходного кода, чтобы найти баланс между размером команды и продуктивностью ее работы. Можно также распределить принадлежность сервисов между командами, чтобы люди, работающие над тем или иным сервисом, трудились вместе. Более подробно мы поговорим об этом в главе 10 при рассмотрении закона Конвея.

Компонуемость

Одной из ключевых перспектив, открывающихся в результате использования распределенных систем и сервис-ориентированных архитектур, является возникновение новых возможностей повторного использования функциональности. Применение микросервисов позволяет пользоваться какой-либо функциональной возможностью различными способами и для достижения различных целей. Это свойство может приобрести особую важность при обдумывании порядка использования клиентами наших программ. Прошли те времена, когда можно было думать узконаправленно либо о сайте для настольного компьютера, либо о мобильном приложении. Теперь следует думать обо всех многочисленных способах, которые нужны для построения хитросплетений всех возможностей для веб-приложений, простых приложений, мобильных веб-приложений, приложений для планшетных компьютеров или переносных устройств. Организациям, перестающим мыслить весьма узкими категориями и переходящим к осмыслению глобальных возможностей привлечения клиентов, нужны соответствующие архитектуры, которые могут идти в ногу с новым мышлением.

При работе с микросервисами нужно думать о том, что мы открываем стыки, адресуемые внешним частям. По мере изменения обстоятельств мы может сделать что-нибудь иначе. При работе с монолитным приложением у меня зачастую был только один крупномодульный стык, пригодный для использования извне. При возникновении потребности его разбиения для получения чего-либо более полезного мне нужен был молоток! Способы разбиения на части существующих монолитных систем в надежде превратить эти части в пригодные к повторному использованию и перекомпоновке микросервисы будут рассматриваться в главе 5.

Оптимизация с целью последующей замены

Если вы работаете в организации среднего размера или крупной, то, скорее всего, знаете, что такое большая, противная, унаследованная от прошлых времен система, стоящая где-то в углу. Никто не хочет к ней даже прикасаться. Но ваша компания не может без нее работать, несмотря на то что она написана на каком-то странном варианте Фортрана и работает только на оборудовании, которое следовало бы списать лет 25 назад. Почему же никто эту систему не заменил? Вы знаете почему: это слишком объемная и рискованная работа.

А в случае использования отдельных небольших по объему сервисов с их заменой на более подходящую реализацию или даже полным удалением справиться гораздо легче. Часто ли вам приходилось удалять более 100 строк кода в день без особых переживаний? При работе с микросервисами, у которых зачастую примерно такой же объем кода, психологический барьер, мешающий их перезаписи или полному удалению, весьма низок.

Команды, использующие подходы, связанные с микросервисами, не видят ничего страшного в полной переработке сервиса, когда это потребуется, и даже в полном избавлении от него, когда потребность в нем отпадет. Когда исходный код состоит всего из нескольких сотен строк, какая-либо эмоциональная привязка к нему практически отсутствует, а стоимость его замены совсем не велика.

А как насчет сервис-ориентированной архитектуры?

Сервис-ориентированная архитектура (Service-oriented Architecture (SOA)) представляет собой подход в проектировании, при котором несколько сервисов работают совместно для предоставления некоего конечного набора возможностей. Под сервисом здесь обычно понимается полностью отдельный процесс операционной системы. Связь между такими сервисами осуществляется через сетевые вызовы, а не через вызовы методов в границах процесса.

SOA появилась в качестве подхода для борьбы с проблемами больших монолитных приложений. Этот подход был направлен на обеспечение возможности повторного использования программных средств, чтобы два и более приложения для конечного пользователя могли применять одни и те же сервисы. Он был направлен на то, чтобы упростить поддержку или переделку программных средств,

поскольку теоретически, пока семантика сервиса не претерпит существенных изменений, мы можем заменить один сервис другим, никого не ставя об этом в известность.

В сущности, SOA является весьма здравой затеей. Но, несмотря на приложенные к этому существенные усилия, прийти к консенсусу о том, как именно достичь успеха в разработке SOA, пока не удается. По-моему, в нашей отрасли отсутствует целостный взгляд на эту проблему и различные производители в этой области не представили какие-либо стройно изложенные и убедительные альтернативы.

Многие проблемы на пути развития SOA, по сути, имеют отношение к сложностям, связанным с протоколами обмена данными (например, SOAP), поставщиками связующих программных средств, отсутствием методик, позволяющих определить степень детализации сервисов, или неверными методиками выбора мест разделения системы. На страницах этой книги мы попытаемся найти решение каждой из этих проблем. Циник может предположить, что поставщики скооперировались (и в некоторых случаях выступили единым фронтом) в вопросах продвижения SOA с целью продать как можно больше своих продуктов и эти самые продукты дискредитировали саму цель SOA.

Общие рассуждения на тему SOA не помогут вам понять, как можно что-то большое разбить на малые части. В них нет ничего, что помогло бы понять, что это большое на самом деле является слишком большим. Не раскрыт в них в достаточной степени и мир реальных вещей, нет практических способов, позволяющих не допустить чрезмерной связанности сервисов. Многое недосказано, в том числе то, какие подвохи могут подстерегать вас на пути реализации SOA.

Использование микросервисов связано с потребностями реального мира, и для того, чтобы добиться построения качественной SOA, требуется лучше разбираться в системах и архитектурах. Поэтому о микросервисах лучше думать как о конкретном подходе к SOA, в том же ключе, в котором XP или Scrum являются конкретными подходами к разработке гибких программных систем.

Другие технологии декомпозиции

Если разобраться, многие преимущества архитектуры на основе микросервисов возникают из присущей ей детализированности и того факта, что она дает намного больше вариантов решения проблем. Но можно ли достичь таких же преимуществ с помощью подобных технологий декомпозиции?

Совместно используемые библиотеки

Эта весьма стандартная технология декомпозиции, встроенная практически в любой язык программирования, предусматривает разбиение исходного кода на несколько библиотек. Эти библиотеки могут предоставляться сторонними разработчиками или создаваться вашей собственной организацией.

Библиотеки обеспечивают способ совместного использования функциональных возможностей различными командами и службами. Например, я могу создать пригодный для повторного использования набор полезных утилит для сбора данных или, возможно, библиотеку создания статистических данных. Команды могут сосредоточиться на разработке таких библиотек, а сами библиотеки могут использоваться повторно. Но есть в их применении и некоторые недостатки.

Во-первых, утрачивается технологическая разнородность. Как правило, библиотеки должны быть написаны на том же языке или в крайнем случае запускаться на той же самой платформе. Во-вторых, утрачивается та легкость, с которой можно расширять части системы независимо друг от друга. Далее, если только не используются динамически подключаемые библиотеки, вы не можете развернуть новую библиотеку, не свернув весь процесс, поэтому сокращается возможность изолированного развертывания измененного кода. И, возможно, критики отметят отсутствие явных стыков, на основе которых можно предпринять безопасные для архитектуры меры обеспечения отказоустойчивости системы.

У совместно используемых библиотек есть свое место применения. Когда вы поймете, что создаете код не конкретно для своей области бизнеса, а для решения более общей задачи и этот код захочется повторно использовать в рамках всей организации, он станет явным кандидатом на превращение в совместно используемую библиотеку. Но делать это нужно с известной долей осторожности. Общий код, используемый для обмена данными между сервисами, может стать той самой точкой, которая создаст их неразрывную связь. Этот вопрос еще будет рассматриваться в главе 4.

Сервисы могут и должны широко применять библиотеки сторонних разработчиков для повторного использования общего кода. Но всего необходимого нам библиотеки не дают.

Модули

В некоторых языках имеются собственные технологии модульной декомпозиции, выходящие за рамки простых библиотек. Они позволяют в некоторой степени управлять жизненным циклом модулей, допуская их развертывание в запущенном процессе и предоставляя возможность вносить изменения, не останавливая весь процесс.

В качестве одной из технологий подхода к модульной декомпозиции стоит упомянуть спецификацию динамической плагинной (модульной) шины для создания Java-приложений — Open Source Gateway Initiative (OSGI). В самом языке Java понятие «модули» отсутствует, и чтобы увидеть его добавленным к языку, придется, видимо, ждать выхода Java 9. Спецификация OSGI, появившаяся в качестве среды, позволяющей устанавливать дополнительные модули (плагины) в Eclipse Java IDE, используется в данное время в качестве способа подгонки модульной концепции в Java посредством библиотеки.

Проблема OSGI-спецификации в том, что в ней предпринимается попытка применения таких вещей, как управление жизненным циклом модулей без достаточной поддержки в самом языке. Это увеличивает трудозатраты авторов модулей на выполнение приемлемой изоляции модулей. В рамках процесса намного легче попасть в ловушку придания модулям излишней взаимосвязанности, вызывая тем

самым возникновение всяческих проблем. Мой собственный опыт работы с OSGI, совпадающий с опытом коллег, работающих в этой же области, подсказывает, что даже при наличии сильной команды применение OSGI может легко превратиться в еще больший источник осложнений, несопоставимых с обеспечиваемыми преимуществами.

В языке Erlang используется другой подход, при котором модули встроены в рабочий цикл выполнения программы. То есть в Erlang применен весьма зрелый подход к модульной декомпозиции. Модули Erlang можно остановить, перезапустить и обновить, не создавая при этом никаких проблем. В Erlang даже поддерживается запуск более одной версии модуля в любой момент времени, что позволяет обновить модуль более изящным способом.

Возможности модулей в Erlang действительно впечатляют, но, даже если повезет воспользоваться платформой с такими возможностями, все равно придется столкнуться с недостатками, присущими обычным совместно используемым библиотекам. По-прежнему будут действовать строгие ограничения на использование новых технологий, ограничения на независимые расширения, модули могут скатываться в сторону таких технологий объединения, при которых возникнет чрезмерная взаимосвязанность, к тому же у них отсутствуют стыки, позволяющие принимать такие меры, которые бы не нарушали безопасность архитектуры.

Следует поделиться еще одним, последним наблюдением. Технически, может быть, и можно создать четко определенные независимые модули внутри отдельно взятого монолитного процесса. И все же свидетельств этому крайне мало. Сами модули вскоре становятся тесно связанными со всем остальным кодом, из-за чего исчезает одно из их ключевых преимуществ. Наличие у процесса разделительных границ требует в этом отношении соблюдения строгой гигиены (или как минимум чтобы было затруднительно выполнять неверные действия!). Разумеется, я не хочу сказать, что это должно стать основным поводом для разделения процессов, но интересно заметить, что обещание разделения модулей в границах одного процесса в реальном мире выполняется крайне редко.

И пусть модульная декомпозиция в границах процесса может быть именно тем, чего вы хотели добиться вдобавок к декомпозиции вашей системы на сервисы, сама по себе она не поможет решить все проблемы. Если вы пользуетесь лишь языком Erlang, то качество реализации его модулей может устроить вас на весьма продолжительный срок, но я подозреваю, что многие из вас находятся в совершенно другой ситуации. Для всех остальных мы станем рассматривать модули, предлагающие такие же преимущества, как и общие библиотеки.

Никаких универсальных решений

Перед тем как завершить главу, я должен заявить, что микросервисы не похожи на бесплатный обед или универсальное решение, но их нельзя назвать и негодным вариантом вроде золотого молотка. У них существуют те же сложности, что и у всех распределенных систем, и даже если как следует научиться управлять распределенными системами (о чем, собственно, и будет идти речь в данной книге), легкой жизни я все равно не обещаю. Если вы смотрите на все это с позиции монолитной

системы, то, чтобы раскрыть все те преимущества, о которых уже говорилось, придется совершенствоваться в вопросах развертывания, тестирования и мониторинга. Также нужно будет изменить взгляд на расширение своих систем и убедиться в том, что они остаются устойчивыми. И не стоит удивляться, если у вас заболит голова от таких вещей, как распределенные транзакции или теорема CAP!

У каждой компании, организации и системы есть свои особенности. В решении вопроса о том, подойдут вам микросервисы или нет и в какой степени нужно проявлять энтузиазм при их внедрении, сыграют роль многие факторы. В каждой главе этой книги я буду стараться дать вам рекомендации, выделяя потенциальные проблемы, что поможет вам проложить верный путь.

Резюме

Теперь, я надеюсь, вы знаете, что такое микросервис, чем он отличается от других композиционных технологий и что собой представляют некоторые его ключевые преимущества. В каждой из следующих глав мы будем рассматривать подробности того, как достичь этих преимуществ и при этом избежать ряда наиболее распространенных подвохов.

Нам предстоит рассмотреть множество тем, но с чего-то ведь нужно начать. Одной из основных сложностей, с которой приходится сталкиваться при переходе к микросервисам, является изменение роли тех, кто зачастую направляет развитие наших систем, — я имею в виду архитекторов. В следующей главе будут рассмотрены некоторые иные подходы к этой роли, которые могут обеспечить наибольшую отдачу от применения новой архитектуры.

2 Архитектор развития

Мы уже увидели, что микросервисы дают широкое поле для выбора и, соответственно, вынуждают принимать множество решений. Например, сколько различных технологий нужно применять, можно ли разным командам использовать различные идиомы программирования и следует ли разбивать или объединять сервисы. Как прийти к принятию этих решений? С более высокими темпами перемен и более изменчивой средой, допускаемой этой архитектурой, роль архитектора должна измениться. В данной главе я буду отстаивать свой взгляд на роль архитектора и надеюсь взять штурмом эту никому пока не известную высоту.

Неверные сравнения

Ты все еще используешь это слово.
А я думаю, что оно значит совсем не то,
что ты о нем думаешь.

Иниго Монтойя, герой книги У. Голдмана «Принцесса-невеста»

У архитекторов весьма важная задача. Они отвечают за координацию технического представления, помогающего нам дать клиентам именно ту систему, в которой они нуждаются. В одних организациях архитекторам приходится работать с одной командой, в таком случае роли архитектора и технического лидера зачастую сливаются в одну. В других организациях они могут определять представление о всей программе работы в целом, координировать взгляды команд, разбросанных по всему миру или, может быть, действующих под крышей всего одной организации. Но, на каком бы уровне они ни работали, точно определить их роль весьма трудно, и даже несмотря на то, что в компаниях стать архитектором считается карьерным ростом для разработчика, эта роль попадает под огонь критики намного чаще любой другой. Архитектор гораздо чаще других может напрямую повлиять на качество создаваемых систем, условия работы своих коллег и способность организации реагировать на изменения, и все же зачастую нам представляется, что в получении этой роли нет ничего хорошего. Почему же так происходит?

Наша отрасль еще сравнительно молода. Похоже, мы забыли об этом и все еще создаем программы, запускаемые на том, что примерно 70 лет называется компьютерами. Поэтому мы постоянно оглядываемся на другие профессии

в попытке объяснить, чем занимаемся. Мы не врачи и не инженеры, а также не водопроводчики или электрики. Мы находимся в некой золотой середине, поэтому обществу трудно понять нас, а нам трудно понять, к какой категории мы относимся.

Мы заимствуем понятия из других профессий, называя себя инженерами или архитекторами. Но так ли это на самом деле? Архитекторы и инженеры отличаются строгостью и дисциплиной, о которых мы можем только мечтать, и важность их роли в обществе ни у кого не вызывает сомнений. Помнится, разговаривал я с приятелем за день до того, как он стал квалифицированным архитектором. «Завтра, — сказал он, — если я, сидя в баре, дам тебе какой-нибудь совет насчет того, как что-нибудь построить, и окажусь не прав, меня привлекут к ответственности. Мне могут предъявить иск, поскольку с точки зрения закона теперь я квалифицированный архитектор и должен нести ответственность за свои промахи». Важность данной работы для общества определяет то, что ею должны заниматься квалифицированные специалисты. В Великобритании, к примеру, прежде чем назваться архитектором, нужно проучиться семь лет. Но эта работа опирается на фундамент тысячелетних знаний. А у нас такого нет. В том числе и поэтому я считаю разные IT-сертификации совершенно бесполезной затеей, поскольку о том, как выглядит совершенство в нашем деле, мы знаем крайне мало.

Кому-то из нас хочется признания, поэтому мы заимствуем названия из других профессий, у которых уже есть столь вожделенное нами признание. Но это может быть вредным вдвойне. Во-первых, это означает, что мы знаем, чем занимаемся, хотя мы толком этого не понимаем. Не хочу сказать, что здания и мосты никогда не рушатся, но это происходит крайне редко по сравнению с тем количеством обрушений, которые испытывают наши программы, что делает сравнение с инженерами совершенно некорректным. Во-вторых, стоит только взглянуть пристальней, и аналогии очень быстро исчезнут. Посмотрим с другой стороны. Если бы строительство моста было похоже на программирование, то на полпути обнаружилось бы, что противоположный берег на 50 метров дальше и там на самом деле грязь, а не гранит, а вместо пешеходного мостика мы строим автомобильный мост. Наши программы не подчиняются физическим законам, с которыми приходится иметь дело архитекторам или инженерам, мы создаем то, что можно гибко изменять и адаптировать, а также развивать по ходу изменения пользовательских требований.

Наверное, наименее подходящим будет название «архитектор». Смысл профессии в том, что он вычерчивает подробные планы, в которых должны разбираться другие специалисты, и предполагает, что эти планы будут осуществлены. В нем сочетаются художник и инженер, курирующий создание того, что обычно составляет единую концепцию, авторитет архитектора подавляет все другие мнения, за исключением тех редких возражений инженеров-строителей, которые основываются на законах физики. В нашей отрасли такого рода архитектор может натворить ужасных дел. Кучи блок-схем, груды страниц документации, созданные без учета будущего в качестве информационной базы при строительстве совершенной системы. Такой архитектор абсолютно не представляет, насколько сложно все это будет реализовать, не знает, будет ли это все вообще работать, не говоря уже о малейшей возможности что-либо изменить по мере постижения смысла задачи.

Сравнивая себя с инженерами или архитекторами, мы скатываемся к оказанию самим себе медвежьей услуги. К сожалению, сейчас мы застряли на слове «архитектор». Поэтому самое лучшее в данной ситуации — переосмыслить его применительно к нашей среде.

Эволюционное видение для архитектора

Требования у нас изменяются куда быстрее, чем у тех, кто проектирует и строит здания, то же самое можно сказать об инструментах и методах, имеющихся в нашем распоряжении. Создаваемые нами продукты не имеют фиксированной отметки времени. Будучи запущенным в эксплуатацию, программный продукт продолжит развиваться по мере изменений способов его применения. Нужно признать, что после того как большинство создаваемых нами продуктов попадают к потребителям, мы вынуждены реагировать на все замечания последних и подстраивать продукты под них, поэтому нашу продукцию нельзя признать незыблемым творением искусства. Следовательно, архитекторам нужно избавиться от мысли о создании совершенного конечного продукта, а вместо этого сфокусироваться на содействии созданию программной структуры, в которой могут появляться подходящие системы, способные расти по мере более глубокого осмысления стоящих перед нами задач.

Хотя до сей поры мое повествование касалось в основном предостережений о неправомерности слишком близкого сравнения самих себя с представителями других профессий, есть все же одна понравившаяся мне аналогия по поводу роли IT-архитектора, и поэтому я считаю, что лучше дать краткое изложение того, во что бы мы хотели превратить эту роль. Первым, кто разделил со мной идею о том, что следует представлять себе создателя программной структуры скорее градостроителем, чем архитектором, был Эрик Дорненбург (Erik Doernenburg). Роль градостроителя должна быть знакома любому, кто когда-либо играл в SimCity. Она предполагает изучение информации из множества источников с последующей попыткой оптимизировать планировку города в соответствии с насущными нуждами жителей, но при этом не упуская из виду будущие потребности. Нам интересен сам метод, с помощью которого градостроитель влияет на развитие города. Он не говорит: «Постройте именно здесь именно это здание», вместо этого он разбивает город на зоны, точно так же, как в симуляторе SimCity можно сделать одну часть города промышленной зоной, а другую — зоной жилой застройки. А потом уже пускай другие решают, какие именно здания сооружать. Конечно, тут есть ряд ограничений: если нужно построить фабрику, она должна находиться в промышленной зоне. Вместо того чтобы тревожиться о происходящем в одной из зон, градостроитель намного больше времени станет уделять работе над тем, как люди и коммуникации перемещаются из одной зоны в другую.

Город не раз сравнивали с живым существом. Со временем город меняется. Он изменяется и развивается по мере того, как им различными способами пользуются обитатели или его формируют внешние силы. Градостроитель изо всех сил старается предвидеть эти изменения, но признает бесполезность попыток установить непосредственный контроль над всеми аспектами происходящего.

Тут должно просматриваться вполне очевидное сравнение с разработкой программных средств. Поскольку программами кто-то пользуется, мы должны реагировать на процесс их использования и вносить в них изменения. Всего, что может произойти, предвидеть невозможно, и поэтому вместо планирования готовности к любым неожиданностям нужно составлять такой план, который позволит вносить изменения, и избавляться от чрезмерного желания определить окончательный вид каждого компонента. Наш город (то есть программная система) должен быть достаточно подходящим и удобным местом для всех, кто им пользуется. Люди часто забывают, что система должна быть приспособлена не только под пользователей, она должна быть приспособлена и под тех, кто ее разрабатывает, и под тех, кто будет ее эксплуатировать, кому нужно работать в рамках этой системы и быть уверенными в том, что они могут внести в систему все необходимые изменения. Заимствуя высказывание у Фрэнка Бушмана, хочу сказать, что в обязанности архитектора входит обеспечение пригодности системы и для разработчиков.

Градостроителю точно так же, как и архитектору, нужно знать, когда его плану не следуют. Поскольку раздавать предписания на каждом шагу — не его прерогатива, градостроитель должен свести к минимуму внесение собственных корректур в основное направление развития, но, если кто-нибудь задумает построить очистные сооружения в жилой зоне, градостроитель должен быть в состоянии воспрепятствовать этому намерению.

Итак, нашим архитекторам, подобно градостроителям, нужно установить общее направление и вмешиваться в детали реализации в крайних случаях и по весьма конкретному поводу. Они должны гарантировать не только соответствие системы текущим целям, но и ее способность стать платформой для будущих изменений. Они должны сделать так, чтобы и пользователи, и разработчики были одинаково довольны системой. Похоже, предстоит решить нелегкую задачу. Так с чего же начать?

Зонирование

Итак, чтобы на минутку продолжить метафору с архитектором в качестве градостроителя, выясним, что же собой представляют зоны. Это границы сервисов или, возможно, собранных в крупные модули групп сервисов. В качестве архитекторов нам нужно больше заботиться не о том, что происходит внутри зоны, а о том, что происходит между зонами. То есть нужно тратить время на обдумывание вопросов общения сервисов друг с другом или того, как можно будет обеспечить подходящее отслеживание общей жизнеспособности системы. Многие организации приняли микросервисы, чтобы добиться максимальной автономности команд разработчиков, и эта тема будет раскрыта в главе 10. Если ваша организация относится к их числу, то для принятия правильного частного решения придется больше полагаться на команды.

Но между зонами, или блоками, на традиционной архитектурной блок-схеме нам нужно проявлять особую осторожность, поскольку недопонимание в этой области приводит к возникновению всевозможных проблем, устранение которых может оказаться весьма нелегкой задачей.

В пределах отдельно взятого сервиса можно будет вполне смириться с тем, что команда, ответственная за данную зону, выберет другую технологию стека или хранения данных. Но здесь можно пострадать от проблем совершенно иного рода. Разрешая командам выбирать подходящий им инструментарий для работы, будьте готовы к тому, что при необходимости поддержки десяти различных технологий стека станет труднее нанимать людей на работу или переводить их из одной команды в другую. Аналогично, если каждая команда выберет собственное хранилище данных, вам может не хватить опыта для запуска любого из этих хранилищ в масштабе системы. К примеру, компания Netflix при выборе технологий хранения данных придерживается в основном стандарта Cassandra. Может, для всех случаев этот вариант и не самый подходящий, но в компании Netflix считают, что ценность накопленных за счет использования Cassandra инструментария и опыта важнее необходимости поддержки и эксплуатации в масштабе компании нескольких других платформ, которые могли бы лучше подойти для решения конкретных задач. Netflix — это весьма яркий пример, в котором масштаб, вероятно, является первостепенным фактором, но главное — ухватить саму идею.

Однако между сервисами может получиться полный беспорядок. Если будет решено для одного сервиса выставить REST через HTTP, для другого — использовать буферы протокола, а для третьего — Java RMI, то их объединение станет просто кошмаром, поскольку пользующимся ими сервисам придется поддерживать сразу несколько стилей обмена данными. Поэтому я и стараюсь закрепить в качестве руководства к действию обязанность волноваться за все, что происходит между блоками, и снисходительно относиться ко всему, что делается внутри них.

Программирующий архитектор

Если нужно гарантировать, что создаваемые системы не вызывают дискомфорта у разработчиков, архитекторам нужно разбираться в том влиянии, которое имеют их решения. Как минимум это означает, что следует вникать в дела команды, а в идеале это должно означать, что нужно с этой командой еще и заниматься программированием. Тем из вас, кто занимался программированием с компаньоном, не составит труда в качестве архитектора влиться на непродолжительный срок в команду и заняться программированием в паре с кем-нибудь из ее членов. Если хотите быть в курсе хода работ, это должно стать обычной практикой. Не могу не подчеркнуть, насколько важно для архитектора вникать в работу команды! Это куда эффективнее, чем перезваниваться с этой командой или просто просматривать созданный ею код.

Насколько часто нужно погружаться в работу подведомственной вам команды (или команд), зависит главным образом от ее количественного состава. Но главное, чтобы это стало постоянной практикой. Если, к примеру, вы работаете с четырьмя командами, проведите по полдня с каждой из них в течение каждых четырех недель, чтобы гарантировать, что у вас сложились взаимопонимание и хорошие взаимоотношения с их членами.

Принципиальный подход

Правила нужны для того, чтобы им следовало
дурачье и ими руководствовались мудрые люди.

Приписывается Дугласу Бадеру
(Douglas Bader)

Принятие решений при конструировании систем — это путь компромиссов, и архитектуры микросервисов буквально сотканы из решений, принятых на их основе! Разве мы выбираем в качества хранилища данных малознакомую платформу только потому, что она лучше масштабируется? Разве нас устроит наличие в системе двух абсолютно разных стековых технологий? А как насчет трех таких технологий? Некоторые решения могут приниматься на основе уже имеющейся информации, и они даются легче всего. А как насчет решений, которые придется принимать при дефиците информации?

Здесь может помочь формула принятия решений, и отличным способом, помогающим найти такую формулу, станет определение набора ведущих принципов и инструкций, основанного на целях, к достижению которых мы стремимся. Поочередно рассмотрим каждый из аспектов.

Стратегические цели

Роль архитектора сложна и без того, чтобы определять стратегические цели, поэтому, к счастью, обычно нам не нужно делать еще и это! Эти цели должны задавать общее направление деятельности компании и то, как она сама себя представляет в роли творца счастья своих клиентов. Это должны быть цели самого высокого уровня, которые могут не иметь ничего общего с технологиями. Определяться они должны на уровне компании или ее ведущего подразделения. В них закладываются намерения вроде «открыть новые рынки сбыта в Юго-Восточной Азии» или «дать клиенту максимальную возможность работать в режиме самообслуживания». Главное здесь то, к чему стремится ваша организация, а ваша задача — обеспечить технологическую поддержку данных устремлений.

Если вы именно тот человек, который определяет технический кругозор компании, возможно, вам следует больше времени уделять общению с персоналом, не имеющим отношения к технике (или с ее бизнес-составляющей, как эти люди часто себя именуют). Это поможет понять, чем руководствуется бизнес и как можно изменить его взгляды.

Принципы

Принципы являются не чем иным, как правилами, выведенными с целью выверить все, что делается с некой более крупной целью, и эти правила иногда подвергаются изменениям. Например, если одной из стратегических целей организации является сокращение времени вывода на рынок новых функций, можно определить принцип, определяющий, что командам доставки дается полный контроль над жизненным циклом поставок их программных средств по готовности независимо от любых других

команд. Если еще одной целью организации является проведение агрессивной политики предложений своих продуктов в других странах, можно принять решение о реализации принципа переносимости всей системы, позволяющей развертывать ее локально с соблюдением независимости данных.

Наверное, вам не хотелось бы придерживаться слишком большого количества принципов. Лучше, чтобы их было не более десяти, поскольку такое количество вполне можно запомнить или поместить на небольшой плакат. Чем больше принципов, тем выше вероятность того, что они станут накладываться друг на друга или противоречить один другому.

Двенадцать факторов Heroku — это набор конструкторских принципов, сформулированных с целью помочь вам в создании приложений, которые смогли бы неплохо работать на платформе Heroku. Имеет смысл применять их и при других обстоятельствах. Некоторые из этих принципов фактически являются ограничениями, накладываемыми на поведение ваших приложений, чтобы они благополучно работали на платформе Heroku. Под ограничениями понимается то, что очень трудно (или практически невозможно) изменить, но мы ведь решили выбрать принципы. Чтобы отличить то, что нельзя изменить, от всего остального, можно назвать это ограничениями, а все не подпадающее под эту категорию четко назвать принципами. Я думаю, что временами неплохо было бы включать спорные ограничения в список с принципами и смотреть, действительно ли эти ограничения настолько непоколебимы!

Инструкции

Инструкции описывают способы соблюдения принципов. Это набор подробных практических предписаний для выполнения задач. Зачастую они будут носить сугубо технологический характер и должны быть достаточно простыми и понятными любому разработчику. Инструкции могут включать в себя правила программирования, требования о том, что все регистрационные данные должны фиксироваться централизованно, или предписание использовать технологию HTTP/REST в качестве объединяющего стандарта. Из-за своей чисто технологической сути инструкции изменяются чаще принципов.

Как и в принципах, в инструкциях иногда отображают ограничения, принятые в вашей организации. Например, если в ней поддерживается только CentOS, этот факт должен быть отображен в инструкциях.

В основу инструкций должны быть положены наши принципы. Принцип, утверждающий, что полный жизненный цикл систем подконтролен командам, занимающимся доставкой, может означать наличие инструкции, согласно которой все сервисы, развертываемые внутри изолированных учетных записей AWS, обеспечивают возможность самостоятельного управления ресурсами и изолированность от других команд.

Объединение принципов и инструкций

Чьи-то принципы становятся чьими-то инструкциями. К примеру, использование HTTP/REST можно назвать принципом, а не инструкцией, и в этом не будет ничего плохого. Дело в том, что ценность представляет наличие всеобъемлющих идей,

дающих представление о ходе развития системы, и наличие достаточной детализации, позволяющей описать способы осуществления этих идей. Возможно, в объединении принципов и инструкций не будет ничего плохого, если речь идет о небольшой группе команд или даже об одной команде. Но для более крупных организаций, в которых могут применяться различные технологии и рабочие инструкции, для разных мест могут понадобиться разные инструкции, и при этом все они будут соответствовать общему набору принципов. Например, у .NET-команды может быть один набор инструкций, а у Java-команды — другой наряду с наличием набора инструкций, общего для обеих команд. А вот принципы у них должны быть общими.

Практический пример

В ходе работы с одним из наших клиентов мой коллега Эван Ботчер (Evan Bottcher) разработал схему, показанную на рис. 2.1. На рисунке продемонстрирована взаимосвязь целей, принципов и инструкций в очень ясной форме. В течение нескольких лет инструкции будут меняться, в то время как принципы останутся прежними. Такую схему можно распечатать на большом листе бумаги и повесить на видном месте.

Неплохо бы иметь документацию, разъясняющую некоторые из этих пунктов. Но все же мне больше нравится идея иметь в развитие этих замыслов примеры кода, которые можно было бы рассмотреть, изучить и запустить. Еще лучше создать инструментальные средства, допускающие только правильные действия. Все это скоро будет рассмотрено более подробно.

Необходимый стандарт

При проработке инструкций и размышлении насчет компромиссов, на которые необходимо пойти, нужно определить и такой очень важный момент, как допустимая степень изменчивости вашей системы. Одним из основных способов определения неизменных качеств для всех сервисов является установка критериев отбора тех из них, поведение и качество которых не выходят за рамки допустимого. Какой же из сервисов станет добропорядочным жителем вашей системы? Какими возможностями он должен обладать, чтобы гарантировать управляемость системы, и то, что один негодный сервис не приведет к сбою всей системы? Ведь здесь как с людьми: добропорядочность жителя в одних условиях не определяет то, на что он может стать похож в каких-либо других условиях. И тем не менее есть некие общие характеристики подходящих сервисов, которых, на мой взгляд, очень важно придерживаться. Существует ряд ключевых областей, в которых допустимость слишком больших отклонений может привести к весьма жарким временам. Согласно определению Бена Кристенсена (Ben Christensen) из компании Netflix, когда мы мысленно рисуем себе крупную картину, «должна быть стройная система из множества мелких деталей с автономными жизненными циклами, способная представлять собой единое целое». Следовательно, нужно найти разумный баланс оптимизации автономии отдельно взятого микросервиса, не теряя при этом из виду общую картину.

Рис. 2.1. Практический пример принципов и инструкций

Одним из способов выявления сути такого баланса является определение четких признаков, наличие которых обязательно для каждого сервиса.

Мониторинг

Очень важно иметь возможность составить логически последовательное, распространяемое на все сервисы представление о работоспособности системы. Это представление должно быть общесистемным, а не специфичным для конкретного сервиса. Конечно, в главе 8 будет определено, что быть в курсе работоспособности отдельного сервиса также немаловажно, но зачастую только при попытке диагностировать более широкую проблему или разобраться в более

крупной тенденции. Чтобы максимально облегчить задачу, я хотел бы предложить использование однообразных критериев работоспособности и общих отслеживаемых параметров.

Можно выбрать применение механизма активной доставки данных, при котором каждому сервису нужно доставлять данные в центр. Для метрик это может быть Graphite, а для отслеживания работоспособности — Nagios. Или можно принять решение об использовании систем опроса, самостоятельно собирающих данные из узлов. Внутри приемника технологию нужно сделать непрозрачной и не требующей от ваших систем мониторинга изменений с целью ее поддержки. Журналирование здесь входит в ту же категорию: оно должно осуществляться в одном месте.

Интерфейсы

Выбор небольшого числа определенных технологий интерфейса помогает интегрировать новых потребителей. Лучше всего иметь один стандарт. Два — тоже неплохо. А вот наличие 20 различных стилей объединения уже никуда не годится. Это относится не только к выбору технологии и протокола. Если, к примеру, выбор пал на HTTP/REST, что вы будете использовать, глаголы или существительные? Как станете работать со страничной организацией ресурсов? Как будете справляться с управлением версиями конечных точек?

Архитектурная безопасность

Мы не можем позволить, чтобы один плохо работающий сервис нарушил общую работу. Нужно убедиться в том, что наши сервисы сами себя защищают от вредных нисходящих вызовов. Чем больше окажется сервисов, неспособных правильно обработать потенциально сбойный характер нисходящих вызовов, тем более хрупкими будут наши системы. Это означает, что вам как минимум нужно будет выдвинуть требование, чтобы каждый нисходящий сервис получил собственный пул подключений, и вы даже можете дойти до того, чтобы сказать, что каждый из сервисов должен использовать предохранитель. Более подробно этот вопрос будет обсуждаться в главе 11 при рассмотрении микросервисов в более широком масштабе.

Игра по правилам важна и тогда, когда речь заходит о кодах. Если предохранители полагаются на HTTP-коды, а в отношении одного из сервисов принято решение о возврате для ошибок кодов вида 2XX или коды 4XX перепутаны с кодами 5XX, то все меры безопасности окажутся тщетными. Те же опасения возникнут, даже если вы не используете HTTP. Знание того, в чем заключается разница между подходящим и корректно обработанным запросом, плохим запросом, при котором сервис удерживается от каких-либо действий по отношению к нему, и запросом, который может быть в порядке, но точно этого сказать нельзя, поскольку сервер не работал, является ключом уверенности в возможности быстрого отказа и отслеживания возникших проблем. Если наши сервисы будут пренебрегать этими правилами, система получится более уязвимой.

Управление посредством кода

Хорошо бы собраться вместе и согласовать, как именно все нужно сделать. Но тратить время на то, чтобы убедиться, что люди следуют руководящим установкам, менее привлекательно, чем выдвигать разработчикам требования реализовать все стандарты, следование которым ожидается от каждого сервиса. Но я твердо верю в то, что именно такой подход упрощает создание нужных вещей. Насколько я заметил, хорошо срабатывает использование экземпляров и предоставление шаблонов сервисов.

Экземпляры

Написание документации — дело полезное. Я прекрасно это осознаю, ведь в конце концов я же написал эту книгу. И разработчикам нравится код, ведь они могут его запустить и исследовать. Если у вас есть предпочтительные стандарты или передовой опыт, то будет полезно иметь демонстрационные экземпляры. Замысел заключается в том, что, подражая некоторым наиболее удачным частям вашей системы, люди не смогут слишком сильно свернуть с верного пути.

В идеале для обеспечения правильного восприятия это должны быть реальные, имеющиеся у вас в наличии сервисы, а не изолированные сервисы, реализованные просто в качестве совершенных примеров. Уверяя в активном использовании своих экземпляров, вы гарантируете, что используемые вами принципы имеют смысл.

Подгоняемый шаблон сервиса

Как по-вашему, хорошо было бы иметь возможность действительно упростить разработчикам задачу следования большинству ваших указаний, затратив совсем немного усилий? Неужели не было бы замечательно предоставить в распоряжение разработчиков основную часть кода для реализации ключевых свойств, столь необходимых каждому сервису?

Есть два микроконтейнера с открытым кодом на основе JVM-машины — Dropwizard и Karyon. Они делают схожую работу, составляя набор библиотек для обеспечения таких свойств, как проверка работоспособности, обслуживание HTTP-протокола или экспортирование метрик. То есть вы получаете готовый сервис, укомплектованный встроенным сервлет-контейнером, который можно запустить из командной строки. Великолепный способ, чтобы пуститься в путь, почему бы им не воспользоваться? Раз уж все это имеется, почему бы не взять что-нибудь вроде Dropwizard или Karyon и не добавить дополнительные свойства, чтобы добиться соответствия вашему контексту?

Например, вам может потребоваться предписать использование предохранителей. В таком случае можно будет интегрировать такую библиотеку предохранителей, как Hystrix. Или у вас может практиковаться обязательная отправка всех метрик на центральный Graphite-сервер, поэтому нужно будет, наверное, ввести в действие такую библиотеку с открытым кодом, как Metrics из состава

Dropwizard, и настроить ее таким образом, чтобы ее средствами показатели времени отклика и частоты появления ошибок автоматически помещались в известное место.

Приспособив подобный шаблон сервиса под собственный набор инструкций разработчикам, вы тем самым гарантируете то, что ваши команды смогут быстрее продвигаться, а также то, что разработчикам будет весьма трудно задать сервисам неверное поведение, не сбившись с заданного пути.

Разумеется, если используются несколько разнородных технологических стеков, то для каждого понадобится соответствующий шаблон сервиса. Правда, это может стать искусным способом ограничения выбора языка в вашей команде. Если имеющийся шаблон сервиса поддерживает только Java, то на выбор альтернативных стеков у людей может не хватить духу, если при этом им придется самостоятельно проделать существенно больший объем работы. К примеру, в Netflix особое внимание уделяется таким аспектам, как отказоустойчивость, позволяющая гарантировать невозможность сбоя всей системы при выходе из строя какой-либо ее части. Чтобы справиться с такой задачей, нужно проделать большой объем работы и убедиться, что существуют клиентские библиотеки на JVM, предоставляющие командам инструментарий, необходимый для удержания их сервисов в рамках дозволенного. Любое внедрение нового технологического стека будет означать необходимость повторно приложить те же усилия. О подобных повторных усилиях в Netflix заботятся меньше, считая, что порой пойти по неправильному пути очень просто. Если только что реализованный сервис может оказать весьма серьезное влияние на систему, то ошибки, связанные с его отказоустойчивостью, могут создать слишком высокий риск. Netflix смягчает проблему за счет применения попутных (sidecar) сервисов, ведущих локальный обмен данными с JVM-машиной, использующей соответствующие библиотеки.

Нужно остерегаться того, что создание шаблонов сервисов станет работой команды, разрабатывающей основной инструментарий или архитектуру, которая диктует, как все должно быть сделано, хотя и через код. При выработке инструкций нужно добиваться коллективной активности, поэтому в идеале ваша команда (или команды) должна нести общую ответственность за обновление такого шаблона (здесь хорошо срабатывает семейственный подход с открытым исходным кодом).

Мне известно множество случаев, когда принудительно навязываемые командам рамки портили людям настроение и снижали производительность. В стремлении повысить степень повторного использования кода все больше работы заключают в задаваемые из центра рамки, вплоть до того, что они превращаются в некое непреодолимое уродство. Принимая решение о применении подстраиваемых шаблонов сервисов, нужно очень тщательно продумать характер работы этих шаблонов. В идеале их использование должно носить чисто рекомендательный характер, но, если вы собираетесь настоять на их применении, следует понимать, что главным стимулом для разработчиков должна быть легкость этого процесса.

Также нужно осознавать опасность, которую может нести совместно используемый код. В стремлении создать многократно используемый код можно внедрить источники связывания сервисов друг с другом. По крайней мере, в одной из организаций я узнал, что их настолько беспокоила данная проблема, что они фактиче-

ски вручную копировали код шаблона своего сервиса в каждый сервис. Следовательно, на обновление основного сервисного шаблона уходило больше времени, поскольку его нужно было применять в этой системе повсеместно, но их это волновало меньше опасности связывания. Другие команды, с которыми мне приходилось общаться, считали шаблон сервиса общей зависимостью двоичного кода, хотя при этом им нужно было очень постараться, чтобы не скатиться к тенденции DRY (don't repeat yourself — «не повторяйся»), приводящей к чрезмерной связанности системы! С особенностями этой темы нужно будет разобраться, поэтому более подробно она рассматривается в главе 4.

Технические обязательства

Зачастую мы попадаем в такие ситуации, когда не можем следовать нашему техническому видению буквально, и, чтобы получить крайне необходимые функции, нам приходится срезать ряд углов. Это просто один из компромиссов, на которые надо идти. У нашего технического видения есть вполне веские основания. При отклонении от этих оснований могут быть получены краткосрочные преимущества, но в долгосрочной перспективе это может обойтись весьма дорого. Понятие, помогающее нам разобраться с этим компромиссом, называется техническим обязательством. Когда мы получаем техническое обязательство, похожее на обязательство в обычном мире, у него есть текущие затраты и нечто, что нам нужно выплатить по обязательству.

Порой для выполнения технического обязательства не находится кратчайших путей. Что, если наш взгляд на систему изменится, но не все в этой системе будет ему соответствовать? В такой ситуации также придется создавать новые источники технических обязательств.

Архитектор должен иметь более широкий кругозор и понимать суть такого баланса. Важно иметь представление об уровне обязательства и о том, куда нужно вмешиваться. В зависимости от вашей организации у вас могут быть возможности проводить мягкую линию руководства, добиваясь того, чтобы команды сами решали, как отслеживать ход работ и выполнять обязательства. В других же организациях может понадобиться четкая линия поведения, возможно ведение регулярно пересматриваемого журнала учета выполнения обязательств.

Работа с исключениями

Таковы наши принципы и инструкции, определяющие порядок создания систем. А что происходит, когда система от них отклоняется? Иногда принимается решение о том, что это всего лишь исключение из правила. В таком случае стоит, наверное, зарегистрировать такое решение где-нибудь в рабочем журнале как напоминание на будущее. При возникновении довольно большого количества исключений может появиться смысл изменить принцип или инструкцию, чтобы в них отразилось новое понимание действительности. Например, у нас может существовать инструкция, утверждающая, что для хранения данных нужно всегда

использовать MySQL. Но затем возникнет вполне резонный интерес к использованию в качестве широко масштабируемого хранилища системы Cassandra, и тут мы внесем в свою инструкцию поправку, гласящую: «Большинство требований по хранению данных удовлетворять за счет применения MySQL, пока не обнаружится перспектива большого разрастания их объемов, в случает чего применять Cassandra».

Наверное, еще раз нужно напомнить, что все организации разные. Мне приходилось работать с компаниями, где команды разработчиков пользовались высокой степенью доверия и автономности и инструкции не были строгими, а потребности в конкретной работе с исключениями значительно снижены или вовсе отсутствуют. В более жестко структурированных организациях, где разработчики имеют меньше свободы, отслеживание исключений может быть жизненно необходимым для того, чтобы убедиться: установленные правила четко отражают те сложности, с которыми приходится сталкиваться. После всего сказанного я сторонник микросервисов как способа оптимизации автономии команд, дающего им как можно больше свободы в решении текущих проблем. Если вы работаете в организации, накладывающей массу ограничений на деятельность разработчиков, то микросервисы могут вам и не подойти.

Руководство и ведущая роль центра

Чтобы справиться со своими задачами, архитекторам нужна руководящая роль. Что подразумевается под *руководством*? Весьма точное определение этому понятию дается в бизнес-модели по руководству и управлению ИТ на предприятии (Control Objectives for Information and Related Technology (COBIT)): *«Руководство обеспечивает уверенность в достижении целей предприятия путем сбалансированной оценки потребностей заинтересованных сторон, существующих условий и возможных вариантов, установления направления развития через приоритизацию и принятие решений, постоянного мониторинга соответствия фактической продуктивности и степени выполнения требований установленным направлению и целям предприятия».*

Применительно к ИТ руководство может касаться многих аспектов. Мы же хотим уделить особое внимание техническому руководству, что, по моему разумению, и является задачей архитектора. Если одной из задач, выполняемых архитектором, является обеспечение наличия технической концепции, то руководство имеет отношение к тому, чтобы обеспечить соответствие создаваемого этой концепции и участвовать в развитии концепции, если необходимо.

Архитекторы несут весьма многогранную ответственность. Они должны обеспечить наличие набора принципов, которые могут послужить руководством для разработки, а также соответствие этих принципов стратегии организации. Они должны также удостовериться в том, что эти принципы не требуют таких рабочих инструкций, которые будут в чем-то ущемлять разработчиков. Они должны двигаться в ногу с новыми технологиями и знать, когда нужно идти на разумные компромиссы. Это очень высокая ответственность. Кроме всего этого, им еще нужно вести за собой людей, то есть удостовериться в том, что коллеги, с которыми они работают, уяснили принятые решения и готовы к их выполнению. К тому же, как

уже упоминалось, им нужно уделить время командам, чтобы осознать степень влияния их решений, и, возможно, даже вникнуть в создаваемый код.

Слишком большая ответственность? Конечно. Но я твердо придерживаюсь того мнения, что они должны справиться с ней самостоятельно. Чтобы частично разгрузить архитектора и уточнить концепцию, с ним вместе может работать толковая группа представителей руководства.

Обычно руководство является групповой деятельностью. Оно может осуществляться в ходе неформальной беседы с небольшой командой или — с целью широкого охвата — проводиться в форме более структурированных регулярных совещаний с формальными участниками группы. Я думаю, что именно здесь должны быть рассмотрены и при необходимости изменены те самые принципы, о которых говорилось ранее. Эту группу должен возглавлять технолог, и состоять она должна преимущественно из людей, выполняющих работу, в отношении которой осуществляется руководство. Группа должна также отвечать за отслеживание технических рисков и управление ими. Наиболее предпочтительной моделью для меня является та, в которой руководит группой архитектор, но при этом основная часть группы состоит из технологов от каждой исполнительной команды, как минимум руководящих командой. Архитектор должен убедиться в том, что группа работает, но за руководство отвечает группа в целом. Это приводит к разделению нагрузки и обеспечению более высокого уровня заинтересованности. Это также гарантирует, что информация свободно поступает из команд в группу, в результате чего принимаются толковые и осмысленные решения.

Временами группа может принимать решения, с которыми архитектор не согласен. Как в таком случае он должен поступать? Я, который бывал ранее в подобной роли, могу вас уверить в том, что это самая непростая ситуация, с которой приходится сталкиваться. Зачастую я склоняюсь к тому, что нужно соглашаться с решением группы. Я считаю, что сделал все возможное, чтобы убедить людей, но в конечном счете был недостаточно убедителен. Группа чаще всего намного мудрее индивидуума, и я не раз убеждался в том, что не прав! И представьте себе, насколько могут опуститься руки у группы, которой дали свободу для принятия решения, но в конечном счете его проигнорировали. Однако иногда я отвергал решение группы. Но почему и когда? Как выбрать линию поведения?

Вспомните, как детей учат кататься на велосипеде. Вы не можете ехать на нем за них. Вы наблюдаете за тем, как их шатает из стороны в сторону, но если каждый раз, когда вам будет казаться, что они могут упасть, вы станете поддерживать велосипед, они никогда не научатся ездить. И в любом случае они падают гораздо реже, чем вы ожидаете! Если же вы видите, что они вот-вот выедут на дорогу с интенсивным движением или въедут в ближайший пруд с утками, приходится вмешиваться. Так же, будучи архитектором, вы должны хорошо понимать, когда, образно говоря, ваша команда зарулит в пруд с утками. Вам также нужно знать о том, что, даже будучи уверенным в своей правоте и в необходимости остановить команду, вы можете совершить подкоп под собственную позицию, а также создать у группы ощущение, что у них нет права голоса. Иногда приходится смириться с тем решением, с которым вы не согласны. Понимание того, когда это нужно, а когда не нужно делать, дается нелегко, но временами оно просто жизненно необходимо.

Формирование команды

Пребывание в роли главного лица, ответственного за техническую концепцию вашей системы и обеспечение выработки данной концепции, не означает принятия технологических решений. Это придется делать тем людям, с которыми вы работаете. В основном роль технического лидера сводится к тому, чтобы помочь команде повысить мастерство, разобраться в концепции, а также убедиться, что ее члены тоже могут быть активными участниками уточнения и реализации концепции.

Содействие окружающим вас людям в их карьерном росте может принимать множество форм, большинство из которых выходит за рамки данного повествования. Но есть один аспект, где архитектура микросервисов особенно актуальна. При работе с крупными монолитными системами у людей меньше возможностей получить повышение и чем-нибудь *овладеть*. А при работе с микросервисами имеется множество автономных наборов исходного кода, имеющих собственные жизненные циклы. Помочь людям получить повышение, возложив на них ответственность за отдельные сервисы, — это отличный способ посодействовать им в достижении личных карьерных целей и в то же время облегчить нагрузку на тех, кто перегружен работой!

Я твердо верю в то, что замечательные программы создают выдающиеся люди. Если заботиться только о технологической стороне дела, то можно упустить из виду более половины общей картины происходящего.

Резюме

Подводя итог сказанному в данной главе, хочу перечислить все, что считаю основными аспектами, за которые отвечает архитектор развития.

- **Концепция.** Следует обеспечить наличие четко воспринимаемой технической концепции системы, что поможет последней отвечать требованиям потребителей и вашей организации.
- **Чуткость.** Нужно чутко воспринимать влияние ваших решений на потребителей и коллег.
- **Сотрудничество.** Нужно взаимодействовать с как можно большим числом коллег и сотрудников, чтобы стало легче определять, уточнять и реализовывать положения концепции.
- **Приспособляемость.** Следует обеспечить внесение в техническую концепцию изменений в соответствии с требованиями, которые выдвигают потребители или организация.
- **Автономность.** Нужно найти разумный баланс между стандартизацией и возможностью автономности в работе ваших команд.
- **Руководство.** Следует обеспечить соответствие реализуемой системы технической концепции.

Архитектор развития понимает, что справиться со своей нелегкой задачей он может, лишь постоянно поддерживая баланс. Внешние события так или иначе воздействуют на вас, и понимание того, где нужно им сопротивляться, а где — плыть по течению, зачастую приходит только с опытом. Но наихудшей реакцией на события, подталкивающие нас к изменениям, будет проявление еще большей жесткости или косности мышления.

Хотя многие советы, приведенные в данной главе, применимы к архитектору любых систем, микросервисы дают нам более широкие возможности для принятия решений. Поэтому способность успешно справляться со сбалансированностью компромиссов приобретает весьма большое значение.

В следующей главе мы воспользуемся вновь приобретенными знаниями о роли архитектора и приступим к обдумыванию путей поиска правильных границ для наших микросервисов.

3 Как моделировать сервисы

Рассуждения моего оппонента напоминают
мне о язычниках, которые на вопрос о том,
на чем стоит мир, отвечали: «На черепахе».
А на чем тогда стоит черепаха?
«На другой черепахе».

Джозеф Баркер (1854)

Итак, вам известно, что такое микросервисы, и, будем надеяться, вы понимаете, в чем их основные преимущества. Теперь, наверное, вам не терпится приступить к их созданию, не так ли? Но с чего начать? В этой главе будет рассмотрен подход к определению границ ваших микросервисов, что, надеюсь, позволит максимизировать их положительные качества и избежать ряда потенциальных недостатков. Но сначала нам нужно что-нибудь, с чем можно будет работать.

Представление MusicCorp

Книги о замыслах лучше воспринимаются, если в них есть примеры. Там, где это возможно, я буду делиться с вами историями из реальной жизни, но в то же время я пришел к выводу, что не менее полезно иметь под рукой какую-либо вымышленную область, с которой можно будет работать. В книге мы еще не раз будем обращаться к этой области, наблюдая за тем, как в ней работает концепция микросервисов.

Итак, перед нами современный онлайн-продавец MusicCorp. Совсем недавно компания MusicCorp занималась традиционной розничной торговлей, но, когда бизнес по продаже грампластинок рухнул, их усилия все больше стали сосредотачиваться на онлайн-торговле. У компании имеется сайт, и в ней зреет стремление удвоить свои онлайн-продажи. Ведь все эти iPod всего лишь дань моде (плееры Zune, конечно, лучше), и истинные музыкальные фанаты готовы ждать доставки на дом компакт-дисков. Качество превыше удобства, не так ли? Исходя из этого, можно ли говорить о каком-то стриминговом сервисе Spotify, который, по сути, может пользоваться успехом только у подростков?

Несмотря на некоторое отставание от общих тенденций, у MusicCorp большие амбиции. К счастью, в компании было принято решение о том, что завоевать мир будет легче всего путем максимального упрощения внесения изменений. И для победы нужны микросервисы!

Как создать хороший сервис

Перед тем как команда из MusicCorp рванет по дистанции, создавая сервис за сервисом в попытке доставлять всем подряд восьмидорожечные ленты, притормозим и немного поговорим о наиболее важном основном замысле, которого нужно придерживаться. Как создать хороший сервис? Если вы уже испытали на себе горечь поражения при создании сервис-ориентированной архитектуры, то вполне можете понять, к чему я клоню. Но на случай, если сия участь вас миновала, хочу выделить две основные концепции: *слабую связанность* и *сильное зацепление*. Конечно, в книге будут подробно рассматриваться и другие замыслы и инструкции, но все усилия по их воплощению в жизнь будут тщетны, если неверно истолкованы эти две концепции.

Несмотря на то что эти два понятия используются довольно широко, особенно в контексте объектно-ориентированных систем, стоит все же поговорить о том, что они означают, когда речь идет о микросервисах.

Слабая связанность

Когда между сервисами наблюдается слабая связанность, 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 тесной 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. Более подробно способы, позволяющие избегать подобных ситуаций, будут рассматриваться в главе 4.

Слабо связанные сервисы имеют необходимый минимум сведений о сервисах, с которыми приходится сотрудничать. Это также, наверное, означает лимитирование количества различных типов вызовов одного сервиса из другого, потому что, помимо потенциальных проблем производительности, слишком частые связи могут привести к тесной связанности.

Сильное зацепление

Хотелось бы, чтобы связанное поведение находилось в одном месте, а несвязанное родственное поведение — где-нибудь в другом. Почему? Да потому, что при желании изменить поведение нам хотелось бы иметь возможность произвести все изменения в одном месте и как можно быстрее выпустить их. Если же придется изменять данное поведение во многих разных местах, то для выпуска изменения нужно будет выпускать множество различных служб (вероятнее всего, одновременно). Изменения во многих разных местах выполняются медленнее, а одновременное развертывание множества сервисов очень рискованно, и оба этих обстоятельства нам нужно как-то обойти.

Следовательно, нужно найти в нашей проблемной 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.

Ограниченный контекст

В книге Эрика Эванса (Eric Evans) *Domain-Driven Design* (Addison-Wesley) основное внимание уделялось способам создания систем, моделирующих реально существующие области. В книге множество великолепных идей вроде использования единого языка, хранилища абстракций и т. п., а еще там представлено одно очень важное понятие, которое я поначалу упустил из виду: *ограниченный контекст* (bounded context). Суть его в том, что каждая отдельно взятая область состоит из нескольких ограниченных контекстов и то, что в каждом из них находится, — это предметы (Эрик широко использует слово *«модель»*, что, наверное, лучше *предмета*), которые не должны общаться с внешним миром, а также предметами, которые во внешнем мире используются совместно с другими ограниченными контекстами. У каждого ограниченного контекста имеется четко определенный интерфейс, где он решает, какие модели использовать совместно с другими контекстами.

Мне нравится еще одно определение ограниченного контекста: *«конкретная ответственность, обеспечиваемая четко обозначенными границами»*[1]. Если нужна информация из ограниченного контекста или нужно сделать запросы на какие-либо действия внутри ограниченного контекста, происходит обмен данными с его четко обозначенной границей с помощью моделей. В своей книге Эванс использовал аналогию с клетками: «Клетки могут существовать благодаря своим мембранам, определяющим, что попадает внутрь, что выходит наружу и что именно может через них проходить».

Ненадолго вернемся к бизнесу MusicCorp. Нашей областью будет весь бизнес, в пределах которого мы действуем. Он охватывает все: от товарного склада до регистратуры и от финансов до заказов. Мы можем создавать или не создавать модели всего этого в наших программах, но это все равно будет нашей рабочей областью. Рассмотрим части этой области, похожие на ограниченные контексты, на которые ссылался Эванс. В MusicCorp центром активности является товарный склад, где управляют доставляемыми заказами (и случайными возвратами), принимают новые запасы, разъезжают вилочные погрузчики и т. д. А в финансовом отделе, возможно, не так оживленно, но все же там происходят весьма важные внутриведомственные дела. Его работники занимаются начислением зарплаты, ведут счета компании и составляют важные отчеты. Множество отчетов. И у них, наверное, есть и другая интересная бумажная работа.

Общие и скрытые модели

Финансовый отдел и товарный склад MusicCorp можно рассматривать как два отдельных ограниченных контекста. У них обоих имеется вполне определенный интерфейс для связи с внешним миром (в понятиях отчетов об инвентаризации, кви-

[1] http://bit.ly/bounded-context-explained.

танциях об оплате и т. д.) и существуют детали, о которых должны знать только они (например, вилочные погрузчики и калькуляторы).

Финансовому отделу не нужно знать ничего о подробностях работ, выполняемых внутри товарного склада. Но ему все-таки нужно знать об определенных вещах, например о складских запасах, чтобы поддерживать счета в актуальном состоянии. На рис. 3.1 показан пример схемы контекстов. На ней можно увидеть те понятия, которые являются внутренними для товарного склада, такие как сборщик (человек, составляющий заказы), полки, представляющие собой места хранения, и т. д. Аналогично главная бухгалтерская книга относится к внутренним понятиям финансового отдела и не является предметом общего пользования вне подразделения.

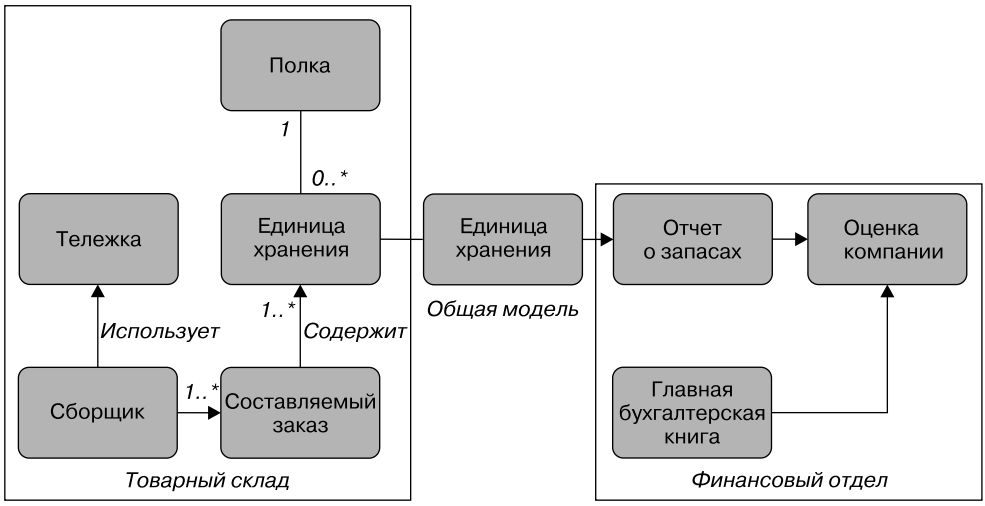


Рис. 3.1. Модель, совместно используемая финансовым отделом и товарным складом

Но чтобы провести оценку компании, работникам финансового отдела нужна информация о складских запасах. Поэтому общей моделью между двумя контекстами становится единица хранения. При этом следует заметить, что совсем не нужно безоглядно показывать все, что касается единицы хранения, из контекста товарного склада. Например, несмотря на то, что внутренняя запись о единице хранения содержится в том виде, в котором она присутствует на товарном складе, показывать абсолютно все в общей модели не нужно. Следовательно, имеется только внутреннее представление и внешнее представление, выставляемое напоказ. Во многом это предваряет рассмотрение REST в главе 4.

Иногда можно столкнуться с моделями с одинаковыми именами, у которых совершенно разное назначение, а также совершенно разные контексты. Например, может существовать такое понятие, как return («возврат»), представляющее собой то, что потребитель отправляет назад. В контексте потребителя понятие return касается распечатки ярлыка доставки, выдачи заказа на посылку и ожидания поступления наложенного платежа. Для товарного склада это понятие может представлять собой поступающую посылку и единицу хранения, запасы которой пополняются. Из этого следует, что в среде товарного склада мы сохраняем дополнительную информацию,

связанную с return, которая относится к будущим задачам, например, на ее основе может быть создан запрос на пополнение запасов. Общая модель return становится связанной с разными процессами и поддерживающей объекты внутри каждого ограниченного контекста, но во многом это внутренняя проблема в пределах самого контекста.

Модули и сервисы

Проясняя вопрос о том, какие модули должны применяться совместно, не допуская при этом совместного использования своих внутренних представлений, мы обходим один из потенциальных подвохов, который может вылиться в тесную связанность (то есть в прямо противоположное желаемому результату). Мы также определяем границу внутри нашей области, в пределах которой должны находиться все однотипные бизнес-возможности, дающие нам желаемое сильное зацепление. Впрочем, эти ограниченные контексты сами по себе играют роль структурных границ.

Как говорилось в главе 1, у нас есть вариант использования модулей в пределах границы процесса, при котором можно держать связанный код вместе и пытаться снизить уровень связанности с другими модулями системы. Это может послужить неплохой отправной точкой при создании нового кода. Итак, если в вашей области обнаружились ограниченные контексты, нужно обеспечить их моделирование внутри кода в виде модулей с наличием как общих, так и скрытых моделей.

Затем эти модульные границы превращают их в превосходных кандидатов в микросервисы. Вообще-то, микросервисы нужно четко вписывать в ограниченные контексты. С обретением достаточного опыта вы можете решиться пропустить этап моделирования ограниченного контекста в виде модулей в составе более монолитной системы и сразу перейти к отдельному сервису. Но на начальном этапе нужно сохранять монолитность новой системы. Неверное определение границ сервиса может обойтись довольно дорого, поэтому нужно дождаться стабилизации представлений, чтобы справиться с новой областью более разумно. Подробнее данный вопрос, а также технологии, помогающие разбить существующие системы на микросервисы, рассматриваются в главе 5.

Итак, если границы нашего сервиса вписываются в ограниченный контекст в нашей области и наши микросервисы представляют собой подобные ограниченные контексты, значит, мы взяли хороший старт в обеспечении слабой связанности и сильного зацепления микросервисов.

Преждевременная декомпозиция

В ThoughtWorks мы сами столкнулись с проблемами слишком быстрого разбиения на микросервисы. Помимо консалтинга, мы также создали несколько продуктов. Одним из них был SnapCI — работающий на хост-машине инструментарий непрерывной интеграции и непрерывной доставки (эти понятия будут рассматриваться в главе 6). Ранее команда работала над другим подобным продуктом, Go-CD — инструментом доставки с открытым исходным кодом, который может развертываться локально, а не размещаться в облаке.

Хотя на самой ранней стадии в проектах SnapCI и Go-CD существовал повторно используемый код, в конечном итоге SnapCI оказался обладателем совершенно нового кода. Тем не менее предыдущий опыт команды в области разработки инструментария по доставке компакт-дисков стимулировал разработчиков к более быстрому определению границ и построению создаваемой системы в виде набора микросервисов.

Через несколько месяцев стало понятно, что сценарий использования SnapCI имел достаточно отличий, чтобы признать изначально определенные границы сервисов не совсем правильными. Это повлекло за собой внесение в сервисы множества изменений и связанные с этим большие затраты. В итоге команда опять объединила сервисы в единую монолитную систему, чтобы лучше понять, где должны пролегать границы. Год спустя команда смогла разбить монолитную систему на микросервисы, границы которых оказались гораздо более стабильными. И это далеко не единственный известный мне пример подобной ситуации. Преждевременная декомпозиция системы на микросервисы может обойтись весьма дорого, особенно если область вам плохо известна. Во многих отношениях куда проще иметь весь исходный код, требующий декомпозиции и разбиения на микросервисы, чем пытаться создавать микросервисы с самого начала.

Бизнес-возможности

Приступая к обдумыванию ограниченных контекстов, имеющихся в вашей организации, нужно размышлять не в понятиях совместно используемых данных, а в понятиях возможностей, предоставляемых такими контекстами всей остальной области. Товарный склад, к примеру, может предоставить возможность получения текущего списка запасов, а финансовый контекст — выдать состояние счетов на конец месяца или позволить внести новичка в платежную ведомость. Для этих возможностей может понадобиться взаимный обмен информацией, то есть совместно используемые модели, но мне довольно часто приходилось наблюдать, что обдумывание данных приводило к созданию безжизненных, основанных на CRUD (create — «создание», read — «чтение», update — «обновление», delete — «удаление») сервисов. Поэтому сначала нужно задать себе вопрос «Чем этот контекст занимается?», а затем уже вопрос «А какие данные ему для этого нужны?».

При проведении моделирования в виде сервисов эти возможности становятся ключевыми операциями, которые могут быть показаны по сети другим участникам системы.

Внизу сплошные черепахи

В самом начале вы, наверное, определите ряд приблизительных ограниченных контекстов. Но они, в свою очередь, могут содержать следующие ограниченные контексты. Например, можно разбить товарный склад по признакам возможностей, связанных с выполнением заказа, управлением запасами или получением товаров. Рассуждая о границах своих микросервисов, сначала нужно оперировать понятиями наиболее

крупных, приблизительных ограниченных контекстов, а затем, выискивая преимущества разбиения в пределах этих границ, приступать к дальнейшему дроблению на вложенные контексты.

Для пущего эффекта я видел эти вложенные контексты скрытыми ото всех остальных сотрудничающих микросервисов. Для внешнего мира они по-прежнему используются с целью реализации бизнес-возможностей на товарном складе, но при этом не знают, что, как показано на рис. 3.2, их запросы фактически открыто отображаются на два и более отдельных сервиса. Временами, как показано на рис. 3.3, можно прийти к решению, что для ограниченного контекста более высокого уровня больше смысла в том, чтобы не быть промоделированным в качестве границы сервиса, и вместо единой границы товарного склада можно выделить запасы, выполнение заказов и получение товаров.

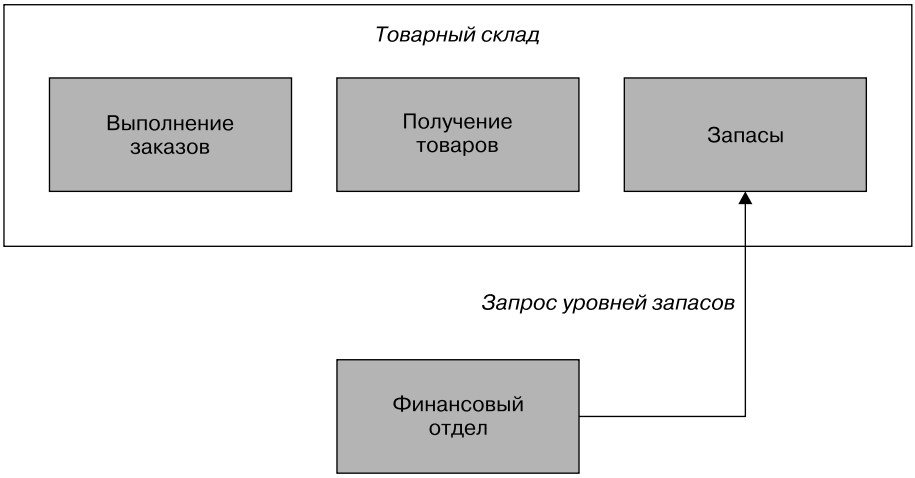


Рис. 3.2. В микросервисах представляются вложенные ограниченные контексты, скрытые внутри товарного склада

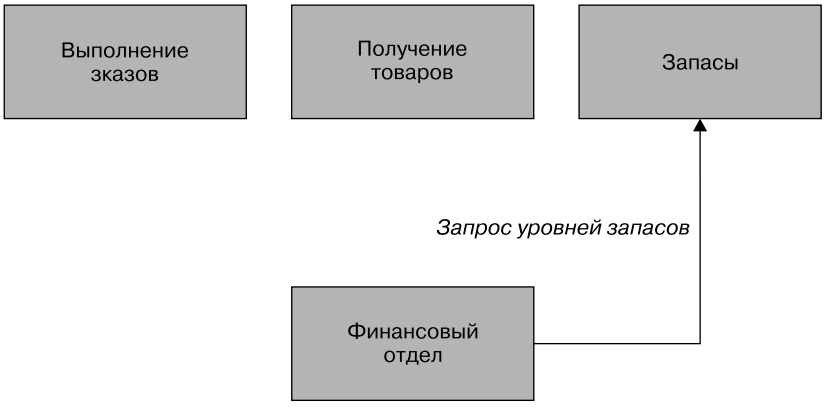


Рис. 3.3. Ограниченные контексты внутри товарного склада, выскочившие на свои собственные контексты самого верхнего уровня

В общем, какого-либо непреложного правила о том, какой из подходов имеет больший смысл, просто не существует. Но выбор подхода с вложенными контекстами, а не подхода с полным отделением должен основываться на структуре вашей организации. Если выполнение заказов, управление запасами и получение товаров управляются разными командами, то они, по-видимому, заслуживают своего статуса микросервисов самого верхнего уровня. Если же все они управляются одной командой, больше смысла будет в модели с вложениями. Все дело во взаимосвязанности организационных структур и архитектуры программного продукта, которая рассматривается ближе к концу книги, в главе 10.

Еще одной причиной, по которой нужно отдавать предпочтение подходу с использованием вложений, может быть разбиение архитектуры на части с целью упрощения тестирования. Например, при тестировании сервисов, использующих товарный склад, не нужно будет ставить заглушки на каждый сервис внутри контекста товарного склада, как при более приблизительном API. Это также может дать вам единицу изолированности при рассмотрении более масштабных тестов. К примеру, я могу принять решение об использовании сквозных тестов при запуске всех сервисов внутри контекста товарного склада, но для всех других сотрудничающих компонентов системы могу их заглушить. Более подробно тестирование и изоляция будут рассматриваться в главе 7.

Обмен данными с точки зрения бизнес-концепций

Изменения, реализуемые в нашей системе, зачастую относятся к изменениям, требующимся бизнесу, чтобы определить поведение системы. Мы изменяем функциональность, то есть возможности, которые раскрываются для наших потребителей. Если наши системы прошли декомпозицию по ограниченным контекстам, представляющим область, изменения, которые нужно произвести, скорее всего, должны быть изолированы одной отдельно взятой границей микросервиса. Это сократит количество мест, в которые нужно вносить изменение, и позволит быстро развернуть это изменение.

Важно также продумать обмен данными между этими микросервисами с точки зрения одних и тех же бизнес-концепций. Моделирование программного продукта относительно вашей области бизнеса не должно останавливаться на замысле ограниченных контекстов. Одинаковые понятия и идеи, совместно используемые отделами вашей организации, должны быть отображены в интерфейсах. Было бы полезно продумать формы, отправляемые между этими микросервисами, почти так же, как и формы, отправляемые в пределах всей организации.

Техническая граница

Полезно было бы взглянуть на то, что может пойти не так, когда сервисы смоделированы неправильно. В недавнем прошлом я с рядом коллег работал с клиентом из Калифорнии, помогая компании внедрить несколько инструкций по очищению кода и приближению к автоматизированному тестированию. Начали мы

с самого легкодоступного — декомпозиции сервиса и тут заметили нечто более тревожное. Я не могу вдаваться в подробности того, чем занималось приложение, но оно относилось к категории общедоступных и имело обширную глобальную клиентскую базу.

Команда и система разрослись. Изначально в представлении отдельно взятого человека система вбирала в себя все больше и больше функций и у нее становилось все больше и больше пользователей. В конце концов организация решила увеличить штат команды — создать новую группу разработчиков, находящуюся в Бразилии, и переложить на нее часть работы. Система подверглась разбиению, причем одна половина приложения, по существу, утратила конкретное «гражданство» и стала представлять собой общедоступный сайт (рис. 3.4). Другая половина системы стала простым интерфейсом удаленного вызова процедуры (Remote Procedure Call (RPC)) в отношении хранилища данных. Представьте, что вы, по сути, берете в своем исходном коде уровень хранилища данных и превращаете его в отдельный сервис.

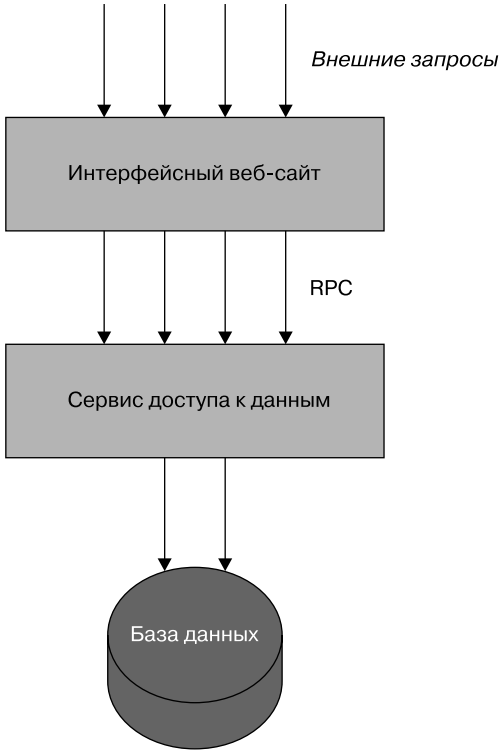


Рис. 3.4. Граница сервиса, проложенная по техническому стыку

В оба сервиса пришлось часто вносить изменения. И оба сервиса рассматривались в понятиях низкоуровневого вызова методов в RPC-стиле, которые обладали излишней хрупкостью (этот вопрос будет рассматриваться в главе 4). Сервисный интерфейс был также слишком многословен, что влекло за собой проблемы с производительностью. Все это вылилось в необходимость усовершенствования меха-

низмов RPC-пакетирования. Я назвал это луковой архитектурой, поскольку в ней имелось множество уровней и она заставляла меня плакать, когда ее приходилось разрезать.

На первый взгляд идея разбиения ранее монолитной системы по географическим или организационным линиям была вполне осмысленной, и ее развернутое представление будет рассмотрено в главе 10. Но в данном случае вместо того, чтобы разрезать стек по вертикали на бизнес-ориентированные куски, команда сделала выбор в пользу того, что ранее было API внутри процесса, и произвела горизонтальный разрез.

Принятие решения о моделировании границ сервиса по техническим стыкам нельзя признать абсолютно неправильным. Я действительно видел оправданность такого подхода, когда организация, к примеру, рассчитывала на достижение определенных целей в повышении производительности. Но поиск подобных стыков должен стать вторичной, но отнюдь не первичной побудительной причиной.

Резюме

В данной главе вы научились в какой-то мере определять критерии хорошего сервиса, а также узнали о способах поиска стыков в своем проблемном пространстве, что дает нам двойные преимущества — как слабой связанности, так и сильного зацепления. Жизненно важным инструментом, помогающим нам находить такие стыки, являются ограниченные контексты, а вписывание микросервисов в определяемые ими границы позволяет гарантировать, что получающаяся в результате система имеет все шансы сохранить свои достоинства неизменными. Кроме того, была дана подсказка о том, как можно выполнить дальнейшее дробление на микросервисы, а углубленное рассмотрение этого вопроса будет приведено чуть позже. Кроме того, состоялось представление MusicCorp, области, взятой в качестве примера, которая будет использоваться в книге и в дальнейшем.

Идеи, представленные Эриком Эвансом в книге *Domain-Driven Design*, весьма полезны при поиске разумных границ наших сервисов, и мы пока что рассмотрели их весьма поверхностно. Чтобы разобраться в практических аспектах данного подхода, я рекомендую обратиться к книге Вона Вернона (Vaughn Vernon) *Implementing Domain-Driven Design* (Addison-Wesley).

В данной главе мы прошлись в основном по верхнему уровню, а далее необходимо углубиться в технические подробности. При реализации интерфейсов между сервисами нас подстерегает множество подвохов, вызывающих разнообразные проблемы, и если мы стремимся уберечь свои системы от сплошной путаницы, в эту тему придется углубиться.

4 Интеграция

На мой взгляд, правильная интеграция является наиболее важным аспектом технологии, связанной с микросервисами. При должном выполнении ваши микросервисы сохранят свою автономию, в то же время можно будет вносить в них изменения и выпускать их новые версии независимо от всей остальной системы. При ненадлежащем исполнении вас ждут серьезные неприятности. К счастью, прочитав эту главу, вы научитесь обходить самые большие из возможных просчетов, от которых страдают другие попытки применения сервис-ориентированной архитектуры и которые могут все еще поджидать вас на пути перехода к применению микросервисов.

Поиск идеальной интеграционной технологии

Для определения способа общения одного микросервиса с другим имеется широкое поле выбора. Но какой из вариантов будет правильным: SOAP, XML-RPC, REST, Protocol Buffers? Прежде чем углубиться в решение этой задачи, подумаем о том, что нам нужно получить от той технологии, на которую падет выбор.

Уклонение от разрушающих изменений

Время от времени мы можем вносить изменения, требующие изменений и от наших потребителей. Как с этим справиться, мы рассмотрим чуть позже, но хотелось бы подобрать такую технологию, которая бы гарантировала, что подобная ситуация станет возникать как можно реже. Например, если микросервис добавляет новые поля к той части данных, которые он куда-нибудь отправляет, это не должно касаться уже имеющихся потребителей.

Сохранение технологической независимости применяемых API

Если ваш стаж пребывания в IT-индустрии превышает 15 минут, то мне не нужно говорить вам о том, что мы работаем в быстро меняющемся пространстве. Что-то обязательно изменится. Все время появляются новые инструменты, среды

программирования и языки, реализуются новые идеи, которые могут помочь нам работать быстрее и эффективнее. Сегодня вы можете пользоваться всем многообразием .NET-технологии. А что будет через год или через пять лет? Что, если вам захочется поэкспериментировать с набором альтернативной технологии, который может увеличить вашу продуктивность?

Я ярый сторонник свободы выбора и именно поэтому являюсь горячим приверженцем микросервисов. Это также является причиной того, что я считаю очень важным сохранение технологической независимости API, используемых для обмена данными между микросервисами. Это означает, что нужно избегать применения тех интеграционных технологий, которые диктуют, какие технологические наборы следует применять при реализации микросервисов.

Сохранение простоты использования сервиса потребителями

Хотелось бы сделать сервис для потребителей как можно более простым в использовании. Красиво представленный микросервис не может рассчитывать на что-то существенное, если цена его применения заоблачно высока! Поэтому задумаемся над средствами, упрощающими использование потребителями нашего замечательного нового сервиса. В идеале хотелось бы дать клиентам полную свободу технологического выбора, но в то же время предоставление клиентской библиотеки может упростить внедрение сервиса. И все же зачастую такие библиотеки несовместимы с другими вещами, которые хотелось бы реализовать. Например, клиентские библиотеки можно использовать в качестве послаблений для потребителей, но это может происходить и за счет повышения связанности.

Скрытие внутренних деталей реализации

Нам не хотелось бы, чтобы наши потребители были привязаны к внутренней реализации сервисов. Это приводит к повышению связанности. Из этого также следует, что при возникновении необходимости внести какие-либо изменения в микросервис мы можем расстроить потребителей, потребовав от них ответных изменений. Это повышает цену изменения, то есть происходит именно то, чего мы стремимся избежать. Это означает также, что нам, скорее всего, не захочется вносить изменения из-за опасений заставить своих потребителей что-либо обновлять, что может повлечь за собой увеличение объема технических обязательств внутри сервиса. Следовательно, нужно избегать любых технологий, вынуждающих нас выставлять напоказ внутренние представления деталей сервисов.

Взаимодействие с потребителями

После получения наставлений, которые могут помочь в выборе подходящей технологии, используемой для интеграции сервисов, рассмотрим некоторые из наиболее

востребованных вариантов и попробуем определить, какой из них нам больше всего подходит. Чтобы проще было все обдумать, возьмем реальный пример из MusicCorp.

На первый взгляд создание клиентов можно рассматривать в виде простого набора CRUD-операций, но для большинства систем все далеко не так просто. Внесение в список нового клиента может потребовать инициирования дополнительных процессов, таких как создание финансовых платежей или отправка приветственных сообщений по электронной почте. А при изменении данных клиента или их удалении могут запускаться и другие бизнес-процессы.

Итак, помня об этом, мы должны рассмотреть ряд других способов работы с клиентами в системе MusicCorp.

Совместно используемая база данных

До сих пор самой распространенной в промышленности формой интеграции, известной мне или любому из моих коллег, является интеграция с использованием базы данных (DB). Если в этой среде другим сервисам нужно получить информацию от какого-нибудь другого сервиса, они обращаются к базе данных. И если им нужно внести в нее изменения, они также обращаются к базе данных! Действительно, на первый взгляд все просто, и для начала это, пожалуй, наиболее быстрый вид интеграции, чем, вероятно, и объясняется его популярность.

На рис. 4.1 показан пользовательский интерфейс регистрации, с помощью которого создаются клиенты путем выполнения SQL-операций непосредственно над базой данных. Там также показано приложение центра обработки заказов, из которого осуществляется просмотр или редактирование клиентских данных путем запуска SQL-запросов в адрес базы данных. А с товарного склада также путем запросов в адрес базы данных производится обновление информации о клиентских заказах. Это довольно широко распространенная схема, но и тут без трудностей не обходится.

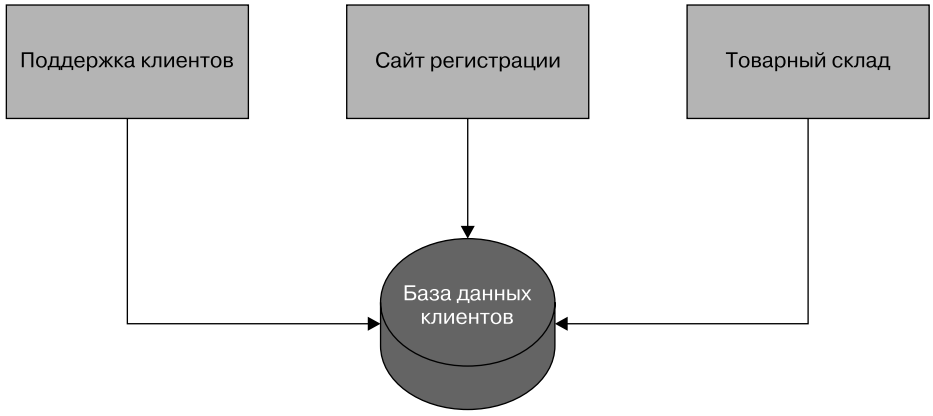


Рис. 4.1. Использование DB-интеграции для доступа к клиентской информации и внесения в нее изменений

Во-первых, разрешены просмотр подробностей внутренней реализации и привязка к ним извне. Структуры данных, хранящихся в базе, становятся законной добычей для любого, они во всей своей полноте используются всеми, кто имеет доступ к базе данных. Если мною будет принято решение изменить свою схему для более подходящего представления своих данных или упростить систему, я могу нарушить работу потребителей. Фактически база данных представляет собой слишком большой совместно используемый API, который к тому же весьма хрупок. Если мне потребуется внести изменения в логику, связанную, скажем, с управлением сервисом поддержки клиентов, и для этого нужно будет вносить изменения в базу данных, мне потребуется предельное внимание, чтобы не нарушить те части схемы, которые используются другими сервисами. Обычно в такой ситуации требуется большой объем регрессионного тестирования.

Во-вторых, мои потребители привязаны к конкретному технологическому выбору. Возможно, именно сейчас имеет смысл хранить сведения о клиентах в реляционной базе данных, и поэтому потребители для обращения с этими данными используют соответствующую (потенциально характерную для баз данных) управляющую программу. А что, если со временем станет понятно, что данные лучше хранить в нереляционной базе данных? Могу ли я принять такое решение? Следовательно, потребители тесно связаны с реализацией сервиса по обслуживанию клиентов. Как упоминалось ранее, мы действительно хотим обеспечить скрытие реализации деталей от потребителей, допуская приобретение нашим сервисом определенного уровня автономности в том, как со временем изменяется его внутреннее содержание. Придется распрощаться со слабой связанностью.

И наконец, на минутку задумаемся о поведении. Должна быть какая-то логика, связанная с тем, как вносятся изменения в данные о клиенте. Где же эта логика? Если потребители напрямую работают с базой данных, значит, связанной с этим логикой должны владеть именно они. Логика для выполнения подобных действий с данными клиентов может не распространяться среди нескольких потребителей. Если в редактировании информации о клиенте нуждаются сразу пользовательские интерфейсы товарного склада, системы регистрации и центра обработки заказов, устранять недочет или изменять поведение нужно в трех разных местах, а кроме того, нужно будет еще и развертывать такие изменения. И тут уже придется распрощаться с зацеплением.

Помните, что говорилось об основных принципах, закладываемых в качественные микросервисы? Сильное зацепление и слабая связанность. А с интеграцией при помощи базы данных мы теряем и то и другое. База данных упрощает для сервисов совместное использование данных, но ничего не может поделать с общим поведением. Внутреннее представление по сети выставляется напоказ другим потребителям, и избежать разрушительных изменений становится очень трудно, что неминуемо приводит к возникновению страха перед внесением любых изменений. А этого нужно избегать практически любой ценой.

Далее в главе будут рассмотрены разные стили интеграции, связывающие совместно работающие сервисы, которые сами скрывают собственное внутреннее представление.

Сравнение синхронного и асинхронного стилей

Перед тем как углубиться в особенности технологического выбора, нужно рассмотреть одно из наиболее важных принимаемых решений в понятиях способов совместной работы сервисов. Каким должен быть обмен данными, синхронным или асинхронным? Этот основополагающий выбор неминуемо приводит нас к конкретным деталям реализации.

При синхронном обмене данными делается вызов на удаленный сервер, блокирующий всю работу вплоть до завершения операции. При асинхронном обмене данными вызывающая сторона перед тем, как вернуть управление, не будет дожидаться завершения операции и даже может не беспокоиться о ее завершении.

Проще рассуждать о синхронном обмене данными. Мы знаем, когда все завершается удачно, а когда — нет. Асинхронный обмен данными может оказаться весьма полезным для продолжительных заданий, где долговременное удержание открытым подключения между клиентом и сервером непрактично. Оно также хорошо себя проявляет, когда не нужны большие задержки, при которых вызов блокирует работу в ожидании результата, что может замедлить весь процесс. Вследствие особенностей, присущих мобильным сетям и устройствам, выдача запросов при условии, что все продолжает работать (если не указано обратное), может гарантировать сохранение отзывчивости пользовательского интерфейса, даже если сеть будет сильно тормозить. В то же время, как мы вскоре узнаем, технология управления асинхронным обменом данными может быть несколько сложнее.

Эти два разных режима обмена данными могут допускать два различных идиоматических стиля совместной работы: *«запрос — ответ»* или *«опора на события»*. При применении стиля «запрос — ответ» клиент инициирует запрос и ждет получения ответа. Эта модель полностью вписывается в синхронный обмен данными, но может работать и при асинхронном обмене. Можно начать операцию и зарегистрировать обратный вызов, обращаясь к серверу с просьбой дать знать о том, когда операция будет завершена.

При совместной работе, основанной на применении событий, все наоборот. Вместо того чтобы клиент инициировал запросы на выполняемые действия, он говорит о том, что *случилось нечто конкретное*, и ожидает того, что другие стороны знают, что им следует делать. О том, что нужно делать, никому другому никогда не говорится. По своей природе системы, основанные на использовании событий, относятся к асинхронным. Интеллектуальные решения распределяются более равномерно, то есть бизнес-логика не централизована в основных интеллектуальных ядрах, а вытеснена в различные совместно работающие сервисы. Кроме того, совместная работа на основе событий обладает высокой степенью разобщенности. Клиент, выдающий событие, не имеет возможности узнать, кто или как на него среагирует, что также означает: вы можете добавить новых подписчиков на эти события без необходимости уведомлять об этом клиента.

Итак, есть ли какие-нибудь другие побудительные причины, которые могли бы подтолкнуть нас к выбору того или иного стиля? Заслуживает рассмотрения то, насколько хорошо эти стили подходят для решения самых сложных задач: как мы справляемся с процессами, выходящими за границы сервисов и выполняемыми достаточно долго?

Сравнение оркестрового и хореографического принципов

Приступая к моделированию все более сложной логики, нам приходится справляться с проблемами управления бизнес-процессами, выходящими за границы отдельных сервисов. А при работе с микросервисами с этим ограничением приходится сталкиваться еще чаще. Возьмем пример из MusicCorp и посмотрим, что происходит при создании клиента.

1. В банке очков лояльности по отношению к клиенту создается новая запись.
2. Наша почтовая система отправляет набор приветственных сообщений.
3. Клиенту отправляется приветственное сообщение по электронной почте.

Концептуально это легко поддается моделированию в виде блок-схемы, что, собственно, и сделано на рис. 4.2.

Рис. 4.2. Процессы, предназначенные для создания нового клиента

Когда наступает черед фактической реализации того, что изображено на блок-схеме, можно придерживаться двух стилей архитектуры. При использовании оркестрового принципа за основу берется центральный интеллект, направляющий процессы и управляющий ими, во многом напоминающий своими действиями дирижера оркестра. При использовании хореографического принципа каждую часть системы информируют о поставленной перед ней задаче, а детали разрешается прорабатывать самостоятельно, они подобны танцорам,

находящим собственный путь и реагирующим на всех окружающих их артистов балета.

Подумаем, какой вид согласно этой блок-схеме приобретет решение по использованию оркестрового принципа. Здесь, наверное, проще всего было бы заставить наш сервис работать в качестве центрального интеллекта. Как показано на рис. 4.3, при создании через серию вызовов «запрос — ответ» происходит общение с банком очков лояльности по отношению к клиенту, сервисом электронной почты и сервисом обычной почты. В дальнейшем сервис клиентов самостоятельно может отслеживать положение клиента в этом процессе. Он может проверить установку учетной записи клиента, или отправку электронной почты, или доставку почтового сообщения. Мы можем взять блок-схему, показанную на рис. 4.2, и смоделировать ее непосредственно в коде. И даже можем воспользоваться инструментарием, который сделает это за нас, возможно, с применением соответствующего обработчика правил. Для этой цели существуют коммерческие инструменты в виде программ моделирования бизнес-процессов. Предположив, что используется синхронный стиль вида «запрос — ответ», мы даже можем узнать, пройден ли тот или иной этап.

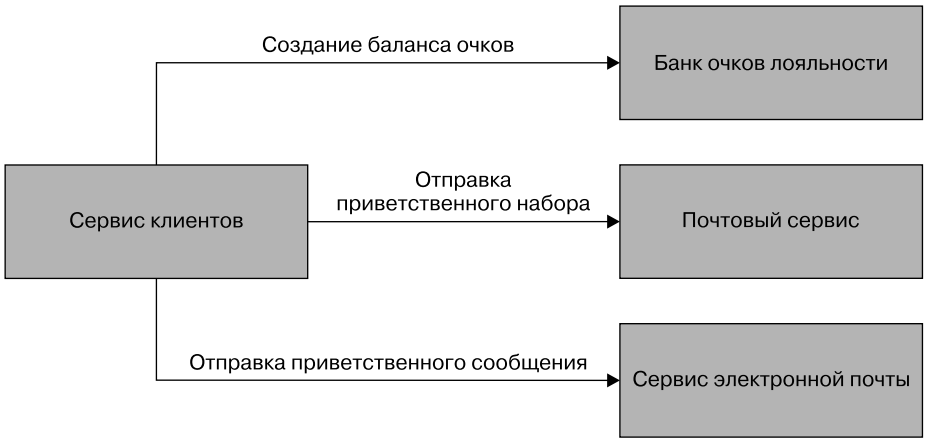


Рис. 4.3. Подход к созданию клиента с помощью оркестрового принципа

Недостаток подхода с использованием оркестрового принципа заключается в том, что сервис клиентов может получить излишне централизованные руководящие полномочия. Он может стать узлом в середине сети и центральной точкой, из которой исходит логика. Я видел, как такой подход приводит к возникновению небольшого количества «божественных» сервисов, предписывающих вялым сервисам на CRUD-основе, что им надлежит делать.

При подходе с использованием хореографического принципа вместо этого можно обязать сервис клиентов выдавать в асинхронной манере события, которые бы оповещали о создании клиента. Затем, как показано на рис. 4.4, сервис электронной почты, сервис обычной почты и банк очков лояльности просто подписались бы на подобные события и реагировали на них соответствующим образом. Этот подход имеет намного более разобщенный характер. Если какому-либо другому сервису

потребуется добраться до создания клиента, ему просто нужно будет подписаться на события и выполнять по мере необходимости возложенную на него задачу. Недостатком является то, что явное представление бизнес-процесса, которое показано на рис. 4.2, теперь отражается в нашей системе лишь в неявном виде.

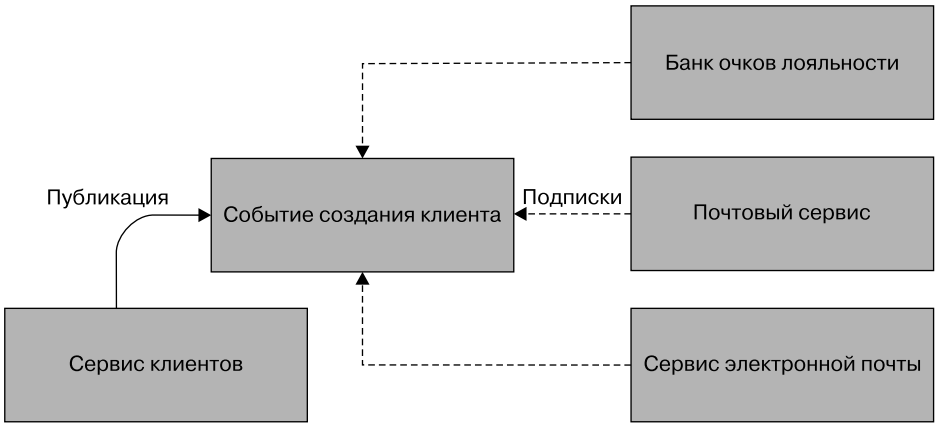


Рис. 4.4. Подход к созданию клиента с использованием хореографического принципа

Это означает необходимость выполнения дополнительной работы, дающей возможность наблюдать и отслеживать надлежащее развитие событий. Например, узнаете ли вы о том, что банк очков лояльности имеет некий изъян и по какой-то причине не завел нужную учетную запись? Одним из предпочитаемых мной подходов является построение системы наблюдения, которая в точности соответствует представлению бизнес-процессов, показанному на рис. 4.2, с последующим отслеживанием того, что делает каждый сервис в качестве независимой единицы. Это позволит замечать случайные исключения, отображаемые на более явное течение процесса. Рассмотренная ранее блок-схема является не побудительной причиной, а всего лишь одной из линз, через которую можно увидеть характер поведения системы.

Вообще-то я понял, что системы, больше тяготеющие к подходу с использованием хореографического принципа, обладают намного более слабой связанностью и большей гибкостью и податливостью к изменениям. Но при этом приходится выполнять дополнительную работу для наблюдения и отслеживания процессов, проходящих через системные границы. Я понял, что системы, обладающие самой сильной приверженностью к использованию оркестрового принципа, получаются слишком хрупкими и требующими более высоких затрат на внесение изменений. Помня об этом, я являюсь стойким приверженцем нацеливания на реализацию системы с использованием хореографического принципа, где каждый сервис обладает достаточным интеллектом для понимания своей роли во всем танце.

Мы можем рассмотреть довольно много факторов. Синхронные вызовы проще, и мы будем осведомлены о том, что все идет намеченным курсом. Если нам нравится семантика «запрос — ответ», но мы имеем дело с долговременными процессами, можем просто инициировать асинхронные запросы и ждать обратных вызовов. В то же время совместная работа с использованием асинхронных событий помогает

ввести подход, использующий хореографический принцип, который может привести к созданию намного более разобщенных сервисов, к чему, собственно, мы и стремимся, чтобы обеспечить для своих сервисов независимую разъемность.

Разумеется, никто нам не мешает заняться смешиванием и подгонкой. К тому или иному стилю некоторые технологии могут подходить более естественно. Тем не менее нужно оценить особенности ряда различных технических реализаций, что в дальнейшем поможет сделать правильный выбор.

Для начала обратимся к двум технологиям, которые хорошо подходят для рассмотрения стиля «запрос — ответ»: удаленному вызову процедуры (RPC) и передаче репрезентативного состояния (REST).

Удаленные вызовы процедуры

Удаленный вызов процедуры является технологией локального вызова, который выполняется где-то на удаленном сервисе. У технологии RPC имеется множество разновидностей. Некоторые из них основаны на применении определенного интерфейса (SOAP, Thrift, Protocol Buffers). Использование отдельного определения интерфейса может облегчить создание клиентских и серверных заглушек для различных технологических стеков, таким образом, к примеру, у меня может быть Java-сервер, выставляющий SOAP-интерфейс, и .NET-клиент, созданный из определения интерфейса на языке описания веб-сервисов (WSDL). Другие технологии вроде Java RMI предусматривают более тесную связанность между клиентом и сервером, требующую, чтобы оба они использовали одинаковую исходную технологию, но при этом исключается необходимость в определении совместно используемого интерфейса. Но все эти технологии имеют одинаковые основные характеристики в том смысле, что они делают локальный вызов, похожий на удаленный вызов.

Многие из этих технологий, такие как Java RMI, Thrift или Protocol Buffers, являются двоичными по своей природе, в то время как в SOAP для форматов сообщений используется язык XML. Некоторые реализации привязаны к конкретному сетевому протоколу (например, SOAP, который номинально использует HTTP), в то время как другие могут допускать использование различных типов сетевых протоколов, которые сами по себе могут обеспечить дополнительные возможности. Например, TCP предоставляет гарантии доставки, а UDP их не дает, но имеет намного меньше издержек. Это может позволить применять различные сетевые технологии для различных вариантов использования.

Те RPC-реализации, которые позволяют создавать клиентские и серверные заглушки, содействуют весьма быстрому старту. Я могу в два счета отправлять контекст через сетевую границу. Зачастую это служит одним из основных аргументов в пользу RPC — эта технология проста в использовании. Тот факт, что я могу просто сделать обычный вызов метода и теоретически проигнорировать все остальное, является существенным подспорьем.

Но некоторые RPC-реализации не лишены недостатков, которые могут вызвать проблемы. Изначально эти проблемы могут быть не столь очевидными, но тем не менее могут оказаться довольно серьезными, чтобы это перевесило преимущества от простоты получения реализации и ее быстрой работы.

Технологическая связанность

Некоторые RPC-механизмы, такие как Java RMI, сильно привязаны к конкретной платформе, что может ограничить выбор технологий для применения на клиенте и сервере. У Thrift и Protocol Buffers имеется впечатляющий диапазон поддержки альтернативных языков, что может некоторым образом сгладить данный недостаток, но при этом следует иметь в виду, что у некоторых RPC-технологий существуют ограничения по возможностям взаимодействия.

В известном смысле эта технологическая связанность может быть одной из форм обнажения деталей внутренней технической реализации. Например, при использовании RMI осуществляется привязка к JVM не только клиента, но и сервера.

Локальные вызовы не похожи на удаленные

Основной замысел RPC заключается в скрытии сложности удаленного вызова. Но многие реализации RPC скрывают их слишком сильно. Управление в некоторых разновидностях RPC с целью сделать удаленный вызов метода похожим на локальный вызов скрывает тот факт, что эти два вызова очень не похожи друг на друга. Можно сделать огромное количество локальных вызовов в рамках одного и того же процесса, практически не переживая о потере производительности. Но при использовании RPC затраты на маршализацию и обратный ей процесс могут быть весьма существенными, даже если не обращать внимания на пересылку по сети. Это означает, что конструкцию удаленного API нужно продумывать иначе, чем конструкцию локальных интерфейсов. Нельзя просто взять локальный API и попытаться без дополнительных размышлений сделать из него границу сервиса, поскольку, скорее всего, ничего, кроме проблем, вы не получите. Если абстракция слишком непрозрачна, то в самых худших примерах разработчики могут использовать удаленные вызовы, даже не зная об этом.

Нужно подумать и о самой сети. Когда речь идет о распределенном вычислении, самым распространенным первым заблуждением является уверенность в надежности сети. Сети не могут быть абсолютно надежными. Они могут и будут отказывать, даже если речь идет о вполне благополучных клиентах и серверах. Они могут сбоить часто или редко, а могут и вовсе портить ваши пакеты. Всегда нужно предполагать, что сети могут подвергнуться воздействию недоброжелателей, готовых в любой момент излить на вас свою злость. Поэтому можно ожидать всяческих отказов. Сбой может быть вызван тем, что удаленный сервер возвратил ошибку, или тем, что вы составили неверный вызов. Можете вы отличить одно от другого, и если да, то можете ли что-то с этим сделать? А что делать, когда удаленный сервер просто начинает медленно реагировать на ваши вызовы? К этой теме мы еще вернемся, когда в главе 11 будем рассматривать эластичность системы.

Хрупкость

Некоторые из наиболее популярных реализаций RPC могут стать причиной опасных форм хрупкости, и хорошим примером этого может послужить Java RMI. Рассмотрим весьма простой Java-интерфейс, который мы решили сделать удаленным

API для нашего сервиса клиентов. В примере 4.1 объявляются методы, которые мы собираемся удаленно представить на всеобщее обозрение. Затем Java RMI сгенерирует клиентскую и серверную заглушки для нашего метода.

Пример 4.1. 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 данном интерфейсе `findCustomer` получает имя, фамилию и адрес электронной почты. А что произойдет, если будет принято решение разрешить объекту `Customer` также быть созданным лишь с адресом электронной почты? В данном случае мы без особого труда можем добавить следующий новый метод:

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

Проблема в том, что теперь необходимо заново создать также клиентские заглушки. Клиенты, желающие использовать новый метод, нуждаются в новых заглушках, и в зависимости от характера изменений в спецификации тем потребителям, которые не нуждаются в новом методе, также может потребоваться обновление их заглушек. Конечно, с этим можно справиться, но до определенного момента. Дело в том, что подобные изменения встречаются довольно часто. На поверку конечные точки RPC зачастую имеют множество методов для различных способов создания объектов или взаимодействия с ними. Отчасти это связано с тем, что мы до сих пор думаем об этих удаленных вызовах как о локальных.

Но есть еще одна разновидность хрупкости. Посмотрим, на что похож наш объект `Customer`:

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

Что, если теперь выяснится, что, несмотря на выставление на всеобщее обозрение в наших объектах `Customer` поля возраста `age`, никто из потребителей им не пользуется? Мы решим, что поле нужно удалить. Но если серверная реализация удаляет поле `age` из своего определения данного типа, а мы не сделаем того же самого для всех потребителей, даже если они никогда не воспользуются этим полем, код, связанный с десериализацией объекта `Customer` на стороне потребителя, будет

поврежден. Чтобы отменить это изменение, я буду вынужден одновременно развернуть как новый сервер, так и новых клиентов. В этом и состоит основная проблема с использованием любого RPC-механизма, продвигающего создание двоичной заглушки: вы не получаете возможности отдельных развертываний клиента и сервера. При использовании данной технологии в будущем вас ожидают такие вот одновременные выпуски с блокировкой всей работы.

Такая же проблема возникнет при желании реструктурировать объект `Customer`, даже если я не стану удалять какие-либо поля, например, если мне захочется для упрощения управления заключить `firstName` и `surname` в новый поименованный тип. Разумеется, я могу справиться с этим путем повсеместной передачи словарных типов в качестве параметров своего вызова, но тогда мне придется расстаться со многими преимуществами создания заглушек, поскольку мне все равно придется вручную искать соответствия и извлекать нужные поля.

На практике объекты, используемые как часть двоичной сериализации отправляемых по сети данных, могут рассматриваться как типы, предназначенные только для раскрытия. Такая хрупкость приводит к типам, раскрываемым для всех, кто находится в сети, и превращаемым во множество полей, часть из которых больше не используются, но не могут быть безопасно удалены.

Неужели RPC настолько страшен?

Несмотря на все недостатки RPC, я не стану сгущать краски и называть его страшным. Некоторые из наиболее распространенных реализаций, с которыми мне приходилось сталкиваться, могли приводить к возникновению тех проблем, которые здесь уже были очерчены. Из-за сложностей использования RMI я, конечно же, постарался бы обойтись без этой технологии. В модель на основе RPC неплохо вписываются многие операции, а более современные механизмы, такие как Protocol Buffers или Thrift, сглаживают некоторые из прошлых грехов, исключая необходимость одновременных выпусков кода клиента и сервера с блокировкой всей работы.

Собираясь остановить свой выбор на этой модели, вы должны быть в курсе всех потенциально возможных подводных камней, связанных с использованием RPC. Не нужно доводить свои удаленные вызовы до такого состояния, при котором использование сети полностью скрыто и следует обеспечить возможность такого развития серверного интерфейса, которое исключало бы необходимость настоятельного требования обновления кода клиентов в режиме блокировки всей работы. К примеру, очень важно выдерживать правильный баланс клиентского кода. Нужно гарантировать, что клиенты не будут обращать никакого внимания на тот факт, что будет производиться вызов по сети. В контексте RPC часто используются клиентские библиотеки, и при неправильной структуризации они могут вызвать ряд проблем. Более подробно данный вопрос будет рассмотрен чуть позже.

По сравнению с интеграцией с использованием баз данных, RPC, несомненно, совершеннее, когда рассматриваются варианты для совместной работы в стиле «запрос — ответ». Но есть и другие варианты для рассмотрения.

REST

Передача репрезентативного состояния (REpresentational State Transfer (REST)) представляет собой архитектурный стиль, инспирированный Всемирной сетью. У REST-стиля имеется множество принципов и ограничений, но мы собираемся сконцентрировать внимание на тех из них, которые действительно помогут нам при встрече с интеграционными трудностями в мире микросервисов и поиске стилей интерфейсов для наших сервисов, выступающих в качестве альтернативы RPC.

Наиболее важным является понятие ресурсов. Под ресурсами можно понимать то, о чем знает сам сервис, например `Customer`. В запросе сервер создает различные образы (или репрезентации) этого объекта `Customer`. Теперь ресурс, выставляемый напоказ, полностью разобщен с тем своим представлением, которое находится на внутреннем хранении. Клиент, к примеру, может запросить JSON-репрезентацию объекта `Customer`, даже если он сохранен совершенно в другом формате. Получив репрезентацию этого объекта `Customer`, он может делать запросы на его изменение и сервер может их выполнять или не выполнять.

Существует множество различных стилей REST, но здесь они будут рассмотрены весьма поверхностно. Я настоятельно рекомендую вам ознакомиться с интернет-ресурсом Richardson Maturity Model, где сравниваются различные стили REST.

В самой REST-технологии речь об исходных протоколах не идет, хотя чаще всего при ее реализации используется протокол HTTP. Ранее мне попадались реализации REST, использующие разные протоколы, такие как последовательный протокол или USB, хотя это может потребовать довольно высоких трудозатрат. Некоторые из свойств, предоставляемые нам протоколом HTTP в качестве части своей спецификации, например глаголы, упрощают реализацию REST по HTTP, тогда как при использовании других протоколов приходится справляться с реализацией подобных свойств самостоятельно.

REST и HTTP

В самом протоколе HTTP определяется ряд весьма полезных возможностей, которые очень хорошо работают на реализацию REST-стиля. Например, фигурирующие в HTTP-спецификации глаголы, такие как GET, POST и PUT, имеют вполне понятный смысл, определяющий характер их работы с ресурсами. Фактически архитектурный стиль REST подсказывает нам, что эти методы будут вести себя так же и в отношении всех ресурсов, и получается, что HTTP-спецификация уже определила тот набор методов, которыми мы можем воспользоваться. GET извлекает ресурс идемпотентным способом, а POST создает новый ресурс. Это означает, что можно исключить применение многочисленных методов `createCustomer` или `editCustomer`. Вместо этого можно просто воспользоваться глаголом POST с репрезентацией клиента, чтобы сделать запрос на сервер с целью создания репрезентации ресурса, и инициировать GET-запрос для извлечения репрезентации ресурса. Концептуально в этих сценариях имеется одна *конечная точка* в форме ресурса `Customer`, а операции, которые мы можем выполнить в ней, готовятся в HTTP-протоколе.

HTTP привносит также обширную экосистему поддерживающих инструментов и технологий. Мы получаем в свое распоряжение такие кэширующие прокси-серверы HTTP, как Varnish, и такие балансировщики нагрузки, как mod_proxy, а также множество средств мониторинга, у которых уже имеется великолепная поддержка HTTP. Эти строительные блоки позволяют справляться с большими объемами HTTP-трафика и осуществлять их интеллектуальную маршрутизацию абсолютно прозрачным способом. С HTTP мы также получаем в свое распоряжение всевозможные средства управления безопасностью обмена данными. Экосистема HTTP дает нам множество инструментов, упрощающих процесс защиты данных, начиная с обычной аутентификации и заканчивая клиентской сертификацией. Более подробно эта тема будет рассматриваться в главе 9. Это говорит о том, что для получения всех этих преимуществ нужно использовать HTTP должным образом. Применение этого протокола неподобающим образом может превратить его в небезопасное и трудно масштабируемое средство, что, впрочем, справедливо и в отношении любой другой используемой в данной сфере технологии. Но при правильном применении вы сможете получить от него весьма большое подспорье.

Следует заметить, что HTTP может использоваться также при реализации RPC. К примеру, SOAP проходит машрутизацию по протоколу HTTP, но, к сожалению, использует весьма скромную часть его спецификации. Глаголы игнорируются, как, впрочем, и такие простые вещи, как коды ошибок HTTP. У меня слишком часто возникает ощущение, что уже существующие и вполне понятные стандарты и технологии игнорируют в угоду новым стандартам, которые можно реализовать только с помощью совершенно новой технологии, любезно предоставляемой теми же самыми компаниями, которые в первую очередь содействуют разработке новых стандартов!

Гиперсреда как механизм определения состояния приложения

Еще одним принципиальным нововведением, представленным в REST и способным помочь нам избежать связанности клиента с сервером, является *гиперсреда, используемая в качестве механизма определения состояния приложения*, часто обозначаемая аббревиатурой HATEOAS, смысл использования которой мне непонятен. Это довольно сжатая формулировка и весьма интересная концепция, поэтому немного в ней разберемся.

Гиперсреда является понятием, в соответствии с которым часть содержимого содержит ссылки на другие части содержимого, представленного разнообразными форматами (например, в виде текста, изображений, звуков). Вам это должно быть знакомо, поскольку тем же самым занимается типовая веб-страница: для просмотра родственного содержимого вы следуете по ссылкам, являющимся формой элементов управления гиперсредой. Идея, положенная в основу HATEOAS, заключается в том, что клиенты должны взаимодействовать с сервером (что потенциально приводит к передаче состояния) посредством этих ссылок на другие ресурсы. При этом, зная, на какой идентификатор ресурса (URI) нужно попасть, не требуется знать, где именно на сервере располагаются данные о клиентах. Вместо этого клиент, чтобы найти нужную информацию, ищет ссылки и переходит по ним.

Это немного необычная концепция, поэтому сначала отвлечемся и посмотрим, как люди работают с веб-страницей, на которой, как уже известно, имеется множество элементов управления гиперсредой.

Рассмотрим торговый сайт Amazon.com. Со временем местонахождение корзины для виртуальных покупок меняется. Меняется графическое представление, меняется ссылка. Но люди достаточно сообразительны для того, чтобы все равно распознавать эту корзину и работать с нею. Мы понимаем, что означает корзина для покупок, даже если изменяются ее конкретная форма и используемый элемент управления. Мы знаем, что при желании увидеть корзину нам нужно работать вот с этим элементом управления. Поэтому со временем веб-страницы могут постепенно изменяться. Пока эти подразумеваемые соглашения между потребителем и сайтом соблюдаются, изменениям не нужно быть разрушительными.

Используя элементы управления гиперсредой, мы пытаемся достичь такого же уровня сообразительности для наших электронных потребителей. Посмотрим на элементы управления гиперсредой, которые подошли бы для MusicCorp. В примере 4.2 мы получаем доступ к ресурсу, представляющему собой запись каталога для заданного альбома. Вместе с информацией об альбоме мы видим номера элементов управления гиперсредой.

Пример 4.2. Элементы управления гиперсредой, используемые для перечня произведений альбома

```
<album>
  <name>Give Blood</name>
  <link rel="/artist" href="/artist/theBrakes" /> (1)
  <description>
    Awesome, short, brutish, funny and loud. Must buy!
  </description>
  <link rel="/instantpurchase" href="/instantPurchase/1234" /> (2)
</album>
```

Этот элемент управления гиперсредой показывает нам, где найти информацию об артисте.

А если нужно приобрести альбом, мы знаем, куда перейти.

В данном документе присутствуют два элемента управления гиперсредой. Клиент, который читает такой документ, должен знать, что элемент управления, относящийся к артисту, переводит к информации об артисте и что `instantpurchase` является частью протокола, используемого для приобретения альбома. Клиент должен понимать семантику API во многом так же, как человеку нужно понимать, что на торговом сайте корзина будет находиться там же, где и потенциальные покупки.

Как клиент, я не должен знать, к какой именно URI-схеме нужно обращаться, чтобы купить альбом, мне нужно просто получить доступ к ресурсу, найти элемент управления, связанный с покупкой, и с его помощью выполнить переход. Элемент управления, связанный с покупкой, может изменить местоположение, может измениться URI-идентификатор, или сайт может даже отослать меня вообще к другой службе, и, как клиента, меня это не должно волновать. Тем самым мы получаем большое количество развязок между клиентом и сервером.

Здесь мы сильно отдалились от исходных деталей. До тех пор пока клиент все еще будет в состоянии находить элемент управления, соответствующий его представлению протокола, мы можем полностью изменить реализацию представления элемента управления. Точно так же элемент управления торговой корзины может из простой ссылки превратиться в более сложный элемент с кодом на JavaScript. Мы также можем вполне свободно добавлять к документу новые элементы управления, возможно, представляющие новые передачи состояния, которые можно будет задействовать в работе с рассматриваемым ресурсом.

Потребителей можно сбить с толку, только если основательно изменить семантику одного из элементов управления, кардинально изменив тем самым его поведение, или если вообще удалить элемент управления. Применение таких элементов управления для разобщения клиента и сервера со временем приносит существенные выгоды, с лихвой компенсирующие небольшое повышение расхода времени, которое занимают оформление и выполнение используемых протоколов. Следуя по ссылкам, клиент получает возможность постепенного раскрытия API, что становится весьма удобным при реализации новых клиентов.

Одним из недостатков навигации по элементам управления является ее возможная многословность, поскольку клиенту для поиска нужной операции приходится следовать по ссылкам. В конечном счете это разумный компромисс. Я советую начать с обеспечения для клиентов возможности переходить по этим элементам, а чуть позже оптимизировать систему, если это потребуется. Следует помнить, что мы получаем большой объем уже готовой помощи от использования HTTP, о чем говорилось ранее. Вред от преждевременной оптимизации мы уже рассматривали, поэтому здесь развивать эту тему я не буду. Нужно также отметить, что для создания распределенных систем с гиперсредами было разработано множество подобных подходов и не все они могут нам подойти! Иногда вы можете поймать себя на мысли об ожидании хорошо зарекомендовавшего себя старомодного удаленного вызова процедур (RPC).

Лично я сторонник того, чтобы в качестве средства навигации по конечным точкам API предоставить потребителям не что иное, как ссылки.

Преимущества постепенного раскрытия API и уменьшения степени связанности могут стать весьма существенными аргументами. Тем не менее понятно, что не все можно реализовать, и я не вижу возможности повсеместного использования этой технологии, как бы мне этого ни хотелось. Я полагаю, что суть данного вопроса в том, что требуется некий авансовый задел, а награды за его создание зачастую приходят позже.

JSON, XML или что-то другое?

Применение стандартных текстовых форматов дает клиентам гибкость в потреблении ресурсов, а REST с использованием HTTP позволяет применять различные форматы. В ранее показанных примерах применялся XML, но на данном этапе намного более популярным форматом содержимого для сервисов, работающих с использованием HTTP, является JSON.

Тот факт, что JSON — намного более простой формат, означает, что его использование также дается проще. Сторонники этого формата также указывают на его

относительную компактность по сравнению с XML как на еще один выигрышный фактор, хотя в реальности это не так уж и существенно.

Но у JSON есть и недостатки. В XML определяется элемент управления `link`, который ранее использовался нами в качестве элемента управления гиперсредой. В стандарте JSON ничего подобного не определяется, поэтому для содействия этой концепции часто используются внутренние стили. В прикладном гипертекстовом языке (Hypertext Application Language (HAL)) предпринимается попытка исправить ситуацию путем определения общих стандартов для создания гиперссылок в JSON (а также в XML, хотя XML, возможно, меньше нуждается в такой помощи). Если следовать стандарту HAL, то для выявления элементов управления гиперсредой можно воспользоваться такими инструментами, как HAL-браузер на веб-основе, который может существенно упростить задачу создания клиента.

Но мы, конечно же, не ограничены этими другими форматами. При желании через HTTP можно отправить практически все что угодно, даже двоичный код. Я все чаще и чаще вижу, как в качестве формата вместо XML используется просто HTML. Для некоторых интерфейсов HTML помогает убить сразу двух зайцев при его применении как в качестве пользовательского интерфейса, так и в качестве API, но при этом все же следует обойти ряд подводных камней, поскольку взаимодействие с человеком и с компьютером — слишком разные вещи! Но это, конечно, весьма привлекательная идея. В конце концов, для HTML существует множество парсеров.

Но лично я предпочитаю XML. У него более подходящая инструментальная поддержка. Например, когда нужно извлечь только вполне определенную часть полезной нагрузки (эта технология будет рассмотрена позже, в разделе «Управление версиями»). Можно воспользоваться XPATH — широко распространенным стандартом, который поддерживают многие инструментальные средства, или даже CSS-селекторами — их многие считают еще более простыми. При использовании JSON есть JSONPATH, но он не получил широкой поддержки. Я считаю странным, что люди выбирают JSON из-за его красоты и легкости применения, затем пытаются внедрить в него такие понятия, как элементы управления гиперсредой, которые уже имеются в XML. Но я понимаю, что, наверное, в данном вопросе отношусь к меньшинству и что JSON является форматом, который выбирает большинство!

Опасайтесь слишком больших удобств

С ростом популярности REST появились среды, помогающие создавать веб-сервисы RESTful. Но в некоторых из них кроется слишком много компромиссов с краткосрочными приобретениями и долгосрочными проблемами. В попытке ускорить процесс эти среды могут потворствовать неприемлемому поведению. Например, некоторые среды действительно упрощают получение представления объектов, сформированных в базах данных, проводя их десериализацию в объекты, встроенные в процесс, после чего эти объекты выставляются на всеобщее обозрение. Я помню, как на конференции состоялся показ с использованием Spring Boot, где все это выдавалось за главное преимущество. Унаследованная связанность, провоцируемая такой системой, зачастую становится причиной куда более серьезных проблем, чем приложение усилий, необходимых для правильного разобщения этих представлений.

Здесь следует заняться решением более общей задачи. Нас в первую очередь должно интересовать решение вопроса о способах хранения данных и их показа потребителям. В одной из схем, увиденной мною и успешно применяемой одной из наших команд, предусматривалась задержка реализации должного постоянства микросервиса вплоть до достаточной стабилизации интерфейса. В промежуточный период образы просто сохранялись в файле на локальном диске, что, конечно же, не было подходящим долговременным решением. Тем самым гарантировалось, что решения по конструкции и реализации диктовались способом использования сервиса потребителями. В обосновании, подтвержденном результатами, утверждалось, что способ хранения объектов нашей предметной области в основном хранилище слишком легко и открыто влияет на те модели, которые посылаются по сети нашим сотрудникам. Одним из недостатков такого подхода является то, что мы откладываем работу, необходимую для подключения к сети хранилища данных. Но я полагаю, что для определения границ нового сервиса это вполне приемлемый компромисс.

Недостатки REST с использованием HTTP

Если говорить о простоте потребления, то создать клиентскую заглушку для REST с применением HTTP так же просто, как при использовании RPC, не удастся. Несомненно, факт применения HTTP означает, что при этом вы можете воспользоваться преимуществами великолепных клиентских библиотек HTTP, но если в качестве клиента вам потребуется реализовать и использовать элементы управления гиперсредой, то во многом придется рассчитывать на собственные силы. Лично я полагаю, что клиентские библиотеки могли бы справляться со своим предназначением намного лучше, чем сейчас, и, конечно же, сейчас они лучше, чем в прошлом, но я увидел, что их явное усложнение приводит к тому, что люди втайне склоняются к возврату к RPC с применением HTTP или создают общие клиентские библиотеки. Совместно используемый клиентом и сервером код может быть очень опасен, о чем будет говориться в разделе «DRY и риски повторного использования кода в мире микросервисов».

Еще одним негативным обстоятельством является то, что в некоторых средах веб-серверов фактически отсутствует качественная поддержка всех HTTP-глаголов. Это означает, что для вас может быть проще создать обработчик GET- или POST-запросов, но, чтобы добиться работы PUT- или DELETE-запросов, возможно, придется заняться прыжками через обруч. У надлежащих REST-сред, таких как Jersey, этих проблем не существует, и со всем этим можно нормально работать, но, если вы замкнуты на выбор конкретных сред, это может ограничить количество доступных для использования стилей REST.

Проблемой может стать также производительность. Полезная нагрузка REST с использованием HTTP может фактически быть более компактной, чем SOAP, поскольку здесь поддерживаются альтернативные форматы вроде JSON или даже двоичный код, но все же эта технология даже не приблизится к той лаконичности двоичного протокола, которая может быть предоставлена языком Thrift. Издержки HTTP для каждого запроса могут также стать проблемой для систем с требованиями малого времени ожидания.

Хотя технология HTTP может оказаться вполне подходящей при больших объемах трафика, с обменом данными, требующим малого времени ожидания, она справляется хуже, если сравнивать ее с альтернативными протоколами, являющимися надстройками над протоколом управления передачей (Transmission Control Protocol (TCP)), или с другими сетевыми технологиями. Несмотря на свое название, протокол WebSocket, к примеру, имеет очень мало общего с Web. После первоначального HTTP-квитирования он представляет собой простое TCP-соединение клиента и сервера, но при этом может стать намного более эффективным способом передачи потоковых данных для браузера. Если вас интересует именно это, следует заметить, что HTTP в нем используется по минимуму, не говоря уже о том, что он не имеет ничего общего с REST.

Для обмена данными между серверами, при котором особую важность приобретает малое время ожидания или малый размер сообщений, связь на основе HTTP вообще может показаться неприемлемой затеей. Для достижения желаемой производительности может понадобиться подобрать другие исходные протоколы, такие как протокол пользовательских датаграмм (User Datagram Protocol (UDP)), и многие RPC-среды будут вполне успешно работать поверх сетевых протоколов, отличных от TCP.

Само же использование полезных нагрузок требует большего объема работы, чем тот, который предоставляется некоторыми RPC-реализациями, поддерживающими улучшенные механизмы сериализации и десериализации. Это само по себе может стать точкой связанности между клиентом и сервером, поскольку реализация приемлемых механизмов чтения данных является не такой уж простой задачей (о чем мы вскоре поговорим), но с точки зрения получения готовой работоспособной технологии они могут быть весьма привлекательными.

Несмотря на указанные недостатки, REST с использованием HTTP является вполне разумным исходным выбором для взаимодействия между сервисами. Если хотите углубить свои знания, я рекомендую почитать книгу *REST in Practice* (O'Reilly), в которой тема REST с использованием HTTP раскрывается намного лучше.

Реализация асинхронной совместной работы на основе событий

Мы уже немного поговорили о некоторых технологиях, содействующих реализации схем «запрос — ответ». А как насчет асинхронного обмена данными на основе событий?

Выбор технологии

Нам предстоит рассмотреть две основные части: способ выдачи микросервисами событий и способ определения потребителями момента наступления того или иного события.

Традиционно такие брокеры сообщений, как RabbitMQ, стараются охватить сразу обе проблемы. Поставщики используют API для публикации события брокеру.

Брокер обрабатывает подписки, позволяя потребителям получить информацию при поступлении того или иного события. Такие брокеры могут даже обрабатывать состояние потребителей, например содействуя отслеживанию того, какие сообщения они видели ранее. Эти системы обычно разрабатываются с возможностями масштабирования и приспособляемости, но это даром не обходится. Возможно, расплачиваться придется усложнением процесса развертывания, поскольку для разработки и тестирования ваших сервисов может понадобиться запуск еще одной системы. Для сохранения работоспособности этой инфраструктуры могут также понадобиться дополнительные машины и наличие определенного опыта. Но если удастся справиться со всеми трудностями, это может стать очень эффективным способом реализации слабо связанных архитектур, управляемых событиями. В общем, я являюсь сторонником именно такого подхода.

Но со связующими системами нужно проявлять разумную осторожность, ведь брокер сообщений составляет лишь малую их часть. В череде поставок имеется еще множество весьма полезных программ. Поставщики, как правило, стремятся включить в пакет наряду с основной массу других программ, способных развить интеллектуальную составляющую, внедряемую в связующие системы, о чем свидетельствуют такие программы, как Enterprise Service Bus. Вы должны понимать, что именно приобретаете: связующие системы не должны проявлять какую-либо инициативу, а интеллектуальные компоненты должны оставаться только в конечных точках.

Еще один подход заключается в попытке использования HTTP в качестве способа распространения событий. Для публикации каналов ресурсов используется такая REST-совместимая спецификация, как ATOM, в которой наряду с другими вещами определяется соответствующая семантика. Существует множество клиентских библиотек, позволяющих создавать и потреблять подобные каналы. Поэтому наш сервис обслуживания клиентов может просто опубликовать событие в таком канале при каких-либо происходящих в нем изменениях. Потребители просто подписываются на канал в поисках изменений. Тот факт, что мы можем воспользоваться уже существующей спецификацией ATOM и любой связанной с ней библиотекой, можно считать положительным, и нам известно, что HTTP весьма неплохо справляется с масштабируемостью. Но HTTP недостаточно хорошо справляется с требованиями малого времени ожидания (в чем преуспевают некоторые брокеры сообщений), и нам все еще нужно считаться с тем фактом, что потребителям требуется отслеживать просматриваемые сообщения и управлять собственным графиком опроса.

Мне попадались люди, тратившие много времени на реализацию все новых и новых линий поведения, которые позволяли воспользоваться ими с соответствующим брокером сообщений и приспособить ATOM для работы в ряде различных сценариев. Например, в системе Competing Consumer описывается метод, позволяющий организовать соревнование за получение сообщений среди нескольких рабочих экземпляров, хорошо подходящий для расширения количества исполнителей, обрабатывающих список независимых заданий. Но нам хотелось бы избежать такого сценария, при котором два и более исполнителя выискивают одно и то же сообщение, поскольку в результате мы получим выполнение одного и того же задания большее количество раз, чем требуется. При использовании

брокера сообщений с этим справляется обычная очередь. А при использовании ATOM нам потребуется управлять нашим общим состоянием, вовлекая в это всех исполнителей с целью уменьшения воспроизводимых усилий.

Если вам уже доступен неплохой приспособляемый брокер сообщений, подумайте о том, чтобы воспользоваться им для обработки публикаций и подписки на события. Если же такой брокер отсутствует, рассмотрите возможность использования спецификации ATOM, но при этом отдавайте себе отчет в неизбежности издержек. Если потребуется более объемная поддержка по сравнению с предлагаемой брокером сообщений, то рано или поздно вам, наверное, захочется изменить свой подход.

В понятиях того, что фактически мы отправляем с использованием этих асинхронных протоколов, мы можем исходить из тех же соображений, которые применялись при синхронном обмене данными. Если на текущий момент вас вполне устраивают зашифрованные запросы и ответы с использованием JSON, то на этом можно и остановиться.

Сложности асинхронных архитектур

В асинхронности есть нечто забавное, не правда ли? Казалось бы, архитектуры, управляемые событиями, обусловливают более разобщенные, масштабируемые системы. И от них этого можно добиться. Но применяемые при этом стили программирования вызывают повышение сложности. Это не только те усложнения, которые, как мы уже выяснили, требуются для управления публикациями и подписками на сообщения, но и другие проблемы, с которыми можно столкнуться. Например, при рассмотрении долгосрочных асинхронных запросов — ответов нам нужно подумать о том, что делать при возвращении ответа. Возвращается ли он на тот же узел, который инициировал запрос?

Если да, то что, если этот узел вышел из строя? Если нет, то нужно ли где-нибудь сохранить информацию, чтобы на нее можно было соответствующим образом среагировать? Краткосрочной асинхронностью может быть проще управлять, если используются надлежащие API, но даже при этом у программистов, привыкших к вызовам синхронных сообщений внутри процессов, должен появиться другой способ мышления.

Здесь самое время рассказать поучительную историю. В далеком 2006 году я работал над созданием системы ценообразования для банка. Мы следили за событиями на рынке и решали, какие элементы в портфеле ценных бумаг требовали переоценки. Как только определялся список всего, над чем нужно было поработать, мы помещали все это в очередь сообщений. Для создания пула исполнителей мы прибегали к использованию сетки, позволявшей по запросу увеличивать и уменьшать ценовую структуру. Для этих исполнителей использовалась система соревновательного потребления, каждый из них выхватывал сообщения как можно быстрее, пока не становилось нечего обрабатывать.

Система была готова к работе, и мы были удовлетворены. Но в один прекрасный день, как раз после выпуска новой версии, столкнулись с весьма неприятной проблемой. Наши исполнители стали гибнуть, и гибнуть, и гибнуть.

В конце концов мы отследили причину возникновения проблемы. В программу вкралась ошибка, при которой конкретный запрос на ценообразование приводил к аварии исполнителя. Нами использовалась очередь, работавшая по принципу транзакции: как только исполнитель выходил из строя, срок его блокировки на запросе истекал, запрос на ценообразование возвращался в очередь, но только для того, чтобы быть выбранным другим исполнителем, который тут же выходил из строя. Это был классический пример того, что Мартин Фаулер называл катастрофическим аварийным переключением.

Кроме самой ошибки, мы не удосужились определить лимит максимального числа попыток для запуска задания в очереди. Мы исправили ошибку, а также настроили максимальное количество повторных попыток. А кроме этого, поняли, что нужен способ просмотра и потенциального воспроизведения подобных ошибочных сообщений. В итоге мы пришли к выводу, что нужно создать изолятор сообщения (или очередь мертвых точек), куда должны попадать сбойные сообщения. Мы также создали пользовательский интерфейс для просмотра таких сообщений и повторной попытки их обработки по мере надобности. Если раньше вы сталкивались только с синхронным обменом данными двух конечных корреспондентов, то сразу же заметить подобные проблемы вам было бы нелегко.

В общем, связанные с архитектурами, управляемыми событиями и асинхронным программированием сложности приводят меня к мысли, что вместо того, чтобы бурно принимать эти идеи, лучше проявлять осмотрительность. Следует убедиться в наличии подходящей системы слежения и серьезно продумать использование взаимосвязи идентификаторов, что позволит проследить запросы по границам процесса. Более подробно эти вопросы будут рассмотрены в главе 8.

Я также настоятельно рекомендую ознакомиться с книгой *Enterprise Integration Patterns* (Addison-Wesley), в которой содержится намного больше подробностей о различных шаблонах программирования, чем вам может понадобиться рассмотреть в данной области.

Сервисы как машины состояний

Если вы решите стать мастером REST-технологии или остановите свой выбор на таком механизме на основе RPC, как SOAP, в действие вступит понятие сервиса как машины состояния. Раньше мы уже достаточно (возможно, даже чрезмерно) наговорились о сервисах, выстраиваемых вокруг ограниченных контекстов. Наши потребительские микросервисы содержат всю логику, связанную с поведением в конкретном контексте.

Когда потребитель хочет внести изменения в данные о клиенте, он отправляет соответствующий запрос в сервис клиентов. Этот сервис, основываясь на своей логике, принимает решение, принять этот запрос или нет. Наш сервис клиентов сам контролирует все события жизненного цикла, связанные с клиентом. Хотелось бы избежать создания немых, безжизненных сервисов, представляющих собой практически простые CRUD-оболочки. Если решение о том, какие изменения разрешено вносить в данные о клиенте, будет вытекать из самого сервиса клиентов, мы утратим зацепление.

Когда жизненный цикл основных понятий заданной области четко смоделирован подобным образом, достигается вполне приемлемый эффект. У нас получается не только одно место для рассмотрения несоответствий состояния (например, когда кто-то пытается обновить данные об уже удаленном клиенте), но также место для придания поведения на основе таких изменений состояния.

Я все-таки полагаю, что REST с использованием HTTP способствует созданию гораздо более практичной технологии интеграции, чем многие остальные решения, но независимо от того, на чем остановится ваш выбор, имейте эти соображения в виду.

Реактивные расширения

Реактивные расширения (reactive extensions, часто сокращается до Rx) представляют собой механизм компоновки результатов нескольких вызовов и запуска операций по их обработке. Сами вызовы могут быть как блокирующими, так неблокирующими. В основном Rx меняют порядок традиционных потоков. Вместо запрашивания каких-либо данных с последующим выполнением в отношении этих данных каких-либо операций изучается исход операции (или набора операций) и происходит реагирование на какие-либо изменения. Некоторые реализации Rx позволяют выполнять какие-либо функции над этими наблюдаемыми результатами, например в RxJava допускается использование таких традиционных функций, как `map` или `filter`.

Различные Rx-реализации очень неплохо прижились в распределенных системах. Они позволяют абстрагироваться от подробностей того, каким образом осуществляются вызовы, и намного проще рассуждать о происходящем. Наблюдается результат вызова какого-либо нижестоящего сервиса. И тут все равно, блокирующий это вызов или нет, ожидается лишь ответ, на который происходит какая-либо реакция. Вся красота в том, что можно компоновать результаты нескольких вызовов, существенно упрощая тем самым обработку конкурирующих вызовов к нижестоящим сервисам.

Если обнаружится, что количество вызовов к сервису возрастает, особенно когда делается несколько вызовов для выполнения одной-единственной операции, присмотритесь к реактивным расширениям для выбранного стека технологий. И вы можете удивиться тому, насколько они смогут упростить вашу жизнь.

DRY и риски повторного использования кода в мире микросервисов

Одним из часто употребляемых акронимов является DRY: don't repeat yourself — «не повторяйтесь». Хотя это определение зачастую упрощенно рассматривается как попытка избежать использования продублированного кода, более точное значение DRY заключается в стремлении избежать продублированности поведения и осведомленности нашей системы. В целом это весьма разумный совет. Наличие большого количества строк кода, выполняющих одно и то же, делает объем исходного кода больше необходимого, затрудняя тем самым его осмысление. При желании

изменить поведение, продублированное во многих частях системы, совсем не трудно забыть о каких-либо местах, в которые нужно вносить изменения, что может привести к появлению ошибок. Поэтому в целом придерживаться DRY-принципа все же стоит.

DRY приводит к созданию кода, пригодного для повторного использования. Повторяющийся код помещается в абстракции, которые затем можно вызывать из нескольких мест. Возможно, мы дойдем и до создания общей библиотеки, которую можно будет использовать повсеместно! Но в архитектуре микросервисов этот подход может оказаться обманчиво опасным.

Обстоятельствами, которых мы избегаем любой ценой, являются излишняя связанность микросервисов и такое их применение, при котором любое, даже самое мелкое изменение самого микросервиса может повлечь за собой ненужные изменения со стороны потребителя. Порой использование общего кода может вылиться в подобную чрезмерную связанность. Например, для одного клиента у нас имелась библиотека общих для заданной области объектов, представляющих основные понятия, используемые в системе. Эта библиотека использовалась для всех имеющихся сервисов. Но когда в один из них были внесены изменения, потребовалось обновление всех сервисов. Обмен данными в системе велся через очередь сообщений, которая также должна опустошаться от теперь уже негодных контекстов, и горе тому, кто это забудет.

Если используется общий код, вечно раскрываемый за пределами границ вашего сервиса, значит, вы ввели потенциальную форму связанности. Использование общего кода наподобие регистрирующих библиотек вполне приемлемо, поскольку они являются внутренними понятиями, невидимыми внешнему миру. В RealEstate.com.au, чтобы помочь в создании с нуля нового сервиса, используется шаблон специализированных сервисов. Вместо того чтобы делать этот код общим, компания копирует его для каждого нового сервиса, чтобы гарантировать невозможность утечки связанности.

Я придерживаюсь следующего практического правила: не нужно навязывать применение DRY-принципов внутри микросервиса, но не стоит опасаться внедрения DRY через все сервисы. Вред от чрезмерной связанности сервисов намного больше вреда от проблем, вызываемых повторяемостью кода. Но все же есть один конкретный сценарий использования, который стоит рассмотреть.

Клиентские библиотеки. Мне приходилось общаться не с одной командой, настаивающей на том, что создание клиентских библиотек для сервисов является наиболее важной частью создания самих сервисов. В качестве подкрепляющих аргументов приводились облегчение использования сервисов и возможность избежать дублирования кода, необходимого для использования сервиса как такового.

Разумеется, есть проблема, связанная с тем, что, если одни и те же люди создают как серверный, так и клиентский API, существует опасность перетекания логики, которая должна присутствовать в сервере, в сторону клиента. Я должен знать: я сделал это сам. Чем больше логики прокрадывается в клиентскую библиотеку, тем сильнее начинает распадаться зацепление, и вы, внедряя исправления в свой сервер, сталкиваетесь с необходимостью внесения изменений в несколько клиентов.

Кроме того, ограничивается простор выбора технологий, особенно если декларативно навязывается применение клиентской библиотеки.

В качестве понравившейся мне модели клиентских библиотек можно назвать Amazon Web Services (AWS). Подразумеваемые SOAP- или REST-вызовы вебсервиса могут быть сделаны напрямую, но каждый из них заканчивается использованием только одного из существующих наборов средств разработки программ (Software Development Kits (SDK)), предоставляющего абстракции поверх основного API. Но эти SDK написаны сообществом разработчиков AWS, а не теми, кто работал над самим API. Похоже, что такая степень разделения сработала и позволила избежать некоторых подводных камней клиентских библиотек. Одна из причин такого успеха заключается в том, что клиент знает, когда происходит обновление. Если вы сами пойдете по пути использования клиентских библиотек, обеспечьте точно такое же развитие событий.

Упор на клиентские библиотеки делается в определенных местах и компанией Netflix, но я подозреваю, что люди смотрят на это только через призму избавления от дублирования кода. Фактически клиентские библиотеки, используемые Netflix, предназначены в основном для обеспечения надежности и масштабируемости систем этой компании. Клиентские библиотеки Netflix занимаются обнаружением сервиса, состояниями отказов, журналированием и другими аспектами, которые не имеют отношения к особенностям самого сервиса. Без этих общих клиентских библиотек было бы сложно обеспечить соответствующее поведение каждой из частей клиент-серверного обмена данными в том крупном масштабе, в котором работает Netflix. Их использование в Netflix, несомненно, облегчает получение работоспособных систем и повышение производительности при обеспечении надлежащего поведения системы. Но, по мнению по крайней мере одного человека из Netflix, со временем это приводит к определенной степени связанности клиента и сервера, вызывающей проблемы.

Если вы задумали воспользоваться подходом с применением клиентской библиотеки, то важным моментом может стать отделение клиентского кода для управления исходным транспортным протоколом, который сможет справляться с обнаружением сервисов и сбоев, от всего, что связано с самим целевым сервисом. Нужно решить, будете ли вы настаивать на применении клиентской библиотеки или же позволите людям использовать другие технологические стеки для вызовов исходного API. И наконец, гарантируйте осведомленность клиентов о необходимости обновления их клиентских библиотек: нам нужно обеспечить сохранение возможности выпуска наших сервисов независимо друг от друга!

Доступ по ссылке

Один из вопросов, которого я хочу коснуться, относится к способу оповещения обо всем, что имеется в нашей области. Мы должны прийти к тому, что микросервис будет охватывать весь жизненный цикл наших основных доменных ресурсов, таких, например, как `Customer`. Мы уже говорили о важности содержания логики, связанной с изменениями ресурса `Customer` в клиентском сервисе, и о том, что при желании внести изменения нужно отправить запрос клиентскому сервису. Но из

этого также следует, что клиентская служба должна рассматриваться как источник истины для ресурсов Customer.

Когда из клиентской службы извлекается заданный ресурс Customer, мы, сделав запрос, получаем возможность увидеть, что он собой представляет. Вполне возможно, что после того, как мы запросили ресурс Customer, кто-то другой внес в него изменения. В результате мы получаем память о том, как когда-то выглядел ресурс Customer. Чем дольше мы держимся за эту память, тем выше шансы, что память будет содержать недостоверную информацию. Разумеется, если мы не станем запрашивать данные чаще, чем это требуется, наши системы станут гораздо эффективнее.

Иногда в памяти будут вполне приемлемые данные, но во всех остальных случаях нужно быть в курсе их изменений. Поэтому, решив обратиться к тому образу ресурса, который был в памяти, нужно также включить ссылку на исходный ресурс, позволяющую извлечь его новое состояние.

Рассмотрим пример обращения к сервису электронной почты на отправку сообщения о том, когда был выслан заказ. Теперь мы можем отправить запрос к сервису электронной почты с подробностями в виде электронного адреса клиента, его имени и заказа. Но если сервис электронной почты уже выстроил очередь из таких запросов или вынул их из очереди, то за время нахождения в ней могли произойти изменения. Может быть, рациональнее было бы просто отправить URI ресурсов Customer и Order и позволить серверу электронной почты просмотреть их, когда настанет срок отправки электронного сообщения.

Отличный контрапункт этому возникает при рассмотрении возможностей совместной работы на основе событий. Работая с событиями, мы говорим о факте случившегося, но нам нужно знать, что именно случилось. Например, если мы получаем обновления, связанные с ресурсом Customer, ценной для нас информацией будет то, на что стал похож Customer, когда событие произошло. Так как мы получили также ссылку на сам ресурс, можно посмотреть на его текущее состояние и взять из обоих миров то, что нам больше подходит.

Разумеется, при получении доступа по ссылке можно пойти и на другие компромиссы. Если при просмотре в ресурсе Customer информации о заданном клиенте мы всегда обращаемся к клиентскому сервису, нагрузка на этот сервис может быть весьма значительной. Если при извлечении ресурса предоставляется дополнительная информация, оповещающая о том, сколько времени ресурс провел в заданном состоянии и, возможно, как долго можно считать эту информацию свежей, то мы можем получить существенные выгоды от кэширования данных и снижения нагрузки на сервис. HTTP предоставляет нам для поддержки всего этого уже готовые решения с широким разнообразием средств управления кэш-памятью, часть из которых более подробно рассматриваются в главе 11.

Еще одна проблема связана с тем, что некоторым сервисам может быть и не нужна информация обо всем ресурсе Customer и, настаивая на том, чтобы они ее искали, мы потенциально усиливаем связанность. Например, может случиться так, что сервис электронной почты должен работать в более простом режиме и ему нужно отправить лишь электронный адрес и имя клиента. Вывести на этот счет какое-либо непреложное правило, конечно, нельзя, но при обходе в запросах тех данных, о степени свежести которых ничего не известно, нужно проявлять крайнюю осмотрительность.

Управление версиями

Практически на каждой лекции о микросервисах мне задавали вопрос о том, *как я справляюсь с управлением версиями*. У людей возникало вполне законное беспокойство о том, что со временем в интерфейс сервиса придется вносить изменения, и они хотели понять, как это можно сделать. Разобьем эту проблему на части и посмотрим на те этапы, которые нужно будет пройти, чтобы с ней справиться.

Откладывание изменений на максимально возможный срок

Наилучшим способом уменьшить влияние внесения критических изменений в первую очередь является отказ от их внесения. Основным способом достижения этого может стать выбор правильной технологии интеграции, о чем и говорилось в данной главе. Интеграция путем использования базы данных является хорошим примером технологии, которая может существенно затруднить отказ от критических изменений. А вот REST помогает достичь желаемого результата, поскольку изменения, вносимые в тонкости внутренней реализации, скорее всего, не приведут к изменениям интерфейса сервиса.

Еще одним ключом к отсрочке внесения критических изменений является содействие правильному поведению ваших клиентов, в первую очередь удержание их от слишком жесткой привязки к вашим сервисам. Рассмотрим сервис электронной почты, чьей задачей является периодическая отправка электронных сообщений клиентам. Он получает задачу на отправку сообщения о высылке заказа клиенту с идентификатором ID 1234. Затем приступает к работе: извлекает данные о клиенте с указанным ID и получает в ответ что-либо подобное показанному в примере 4.3.

Пример 4.3. Пример ответа от клиентского сервиса

```
<customer>
  <firstname>Sam</firstname>
  <lastname>Newman</lastname>
  <email>sam@magpiebrain.com</email>
  <telephoneNumber>555-1234-5678</telephoneNumber>
</customer>
```

Теперь для отправки сообщения по электронной почте нужны только поля `firstname`, `lastname` и `email`. Нам не нужно знать содержимое поля `telephoneNumber`. Требуется просто извлечь те поля, которые нас интересуют, проигнорировав все остальные. Некоторые технологии связывания, в особенности те, которые используются строго типизированными языками, могут попытаться связать все поля независимо от того, нужны они потребителю или нет. Что произойдет, если мы поймем, что поле `telephoneNumber` никто не использует, и решим его удалить? Это может привести к совершенно ненужному нарушению режима работы потребителей.

А что, если нам придет в голову изменить структуру нашего объекта `Customer` так, чтобы она поддерживала более подробные данные, возможно, путем добавления некой дополнительной структуры, как в примере 4.4? А сервису электронной поч-

ты по-прежнему нужны все те же данные и под теми же именами, но, если в коде делаются абсолютно четкие предположения о том, где именно будут храниться данные полей `firstname` и `lastname`, то он опять может стать неработоспособным. В таком случае вместо этого для извлечения нужных нам полей можно воспользоваться XPath, что позволит безразлично относиться к тому, где именно находятся поля, поскольку мы все равно сможем их найти. Именно такая схема, предполагающая создание системы считывания данных, способной проигнорировать те изменения, которые нас не волнуют, получила от Мартина Фаулера название толерантного считывателя (Tolerant Reader).

Пример 4.4. Ресурс Customer с измененной структурой: данные никуда не делись, но сможет ли потребитель их найти?

```
<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, старающегося быть как можно гибче в использовании сервиса, демонстрирует закон Постела, известный также как принцип живучести (robustness principle), который гласит: «Будь требователен к тому, что отсылаешь, и либерален к тому, что принимаешь». Исходной средой для проявления этой мудрости служило взаимодействие сетевых устройств, при котором следует ожидать всевозможных странностей. В контексте же нашего взаимодействия в режиме «запрос — ответ» он может привести нас к стремлению сделать сервис приспособленным к изменениям и не требовать никаких изменений от нас.

Выявление критических изменений на самой ранней стадии

Очень важно гарантировать выявление изменений, способных нарушить работу потребителей как можно раньше, потому что, даже выбрав наилучшую из возможных технологий, мы все равно не будем застрахованы от критических сбоев. Для содействия выявлению этих проблем на ранней стадии я настоятельно рекомендую воспользоваться контрактами, задаваемыми потребителями (consumer-driven contracts), которые рассматриваются в главе 7. Если вы поддерживаете сразу несколько различных клиентских библиотек, то еще одной вспомогательной технологией может стать выполнение тестов с использованием каждой поддерживаемой вами библиотеки в отношении самого последнего сервиса. Как только обнаружится состояние, близкое к нарушению режима работы потребителя, перед вами встает выбор либо попытаться вообще избежать этого нарушения, либо смириться с возникшим состоянием и приступить к переговорам с теми, кто сопровождает сервисы-потребители.

Использование семантического управления версиями

Разве плохо будет, если вы в качестве клиента получите возможность с одного взгляда на номер версии сервиса тут же понять, сможете ли вы интегрироваться с ним или нет? Семантическое управление версиями представляет собой спецификацию, позволяющую получить именно такую возможность. При семантическом управлении версиями у каждой версии есть номер, имеющий форму `MAJOR.MINOR.PATCH` (ВАЖНЫЙ.ВТОРОСТЕПЕННЫЙ.ИСПРАВЛЕНИЕ). Когда происходит увеличение MAJOR-части номера, это означает, что были внесены изменения, исключающие обратную совместимость. Когда увеличивается MINOR-часть номера, это означает, что была добавлена новая функциональная возможность, которая не должна нарушить обратную совместимость. И наконец, когда меняется PATCH-часть номера, это означает, что в существующие функциональные возможности были внесены исправления, устраняющие какие-либо недостатки.

Чтобы убедиться в пользе семантического управления версиями, рассмотрим простой практический пример. Приложение по поддержке клиентов было создано для работы с версией клиентского сервиса, имеющей номер 1.2.0. Если будет добавлено какое-либо новое свойство, которое станет причиной изменения номера версии сервиса на 1.3.0, приложение не заметит никаких изменений в поведении сервиса и от него не будет ожидаться внесения каких-либо изменений в работе. Но мы не можем гарантировать, что будем в состоянии работать с версией 1.1.0 клиентского сервиса, поскольку можем зависеть от наличия тех функциональных возможностей, которые были добавлены при выпуске версии 1.2.0. Мы также можем ожидать необходимости внесения изменений в приложение, если выйдет новый выпуск клиентского сервиса с номером версии 2.0.0.

Решение о применении семантического управления версиями может быть принято как для всего сервиса, так и для его отдельно взятой конечной точки, если вы допускаете сосуществование сразу нескольких конечных точек, подробно рассматриваемое в следующем разделе.

Такая схема управления версиями позволяет помещать всего лишь в три поля достаточный объем информации и предположений. Полные изложения спецификации в очень простой форме обозначают те предположения, которые могут быть сделаны клиентами при изменении представленных номеров частей, и помогают упростить процесс сообщения о том, должны ли изменения каким-либо образом повлиять на потребителей. К сожалению, мне нечасто приходилось наблюдать применение данного подхода к распределенным системам.

Сосуществование различных конечных точек

После того как сделано все возможное, чтобы избежать появления критических изменений интерфейса, следующим заданием станет ограничение влияния. При этом нужно постараться не допустить принуждения потребителей к созданию обновлений вслед за нами, поскольку мы неизменно стремимся обеспечить возможность выпуска микросервисов независимо друг от друга. Одним из подходов, успешно приме-

нявшихся мною при решении этой задачи, являлось сосуществование старого и нового интерфейсов в одном и том же работающем сервисе. То есть, нацелившись на выпуск критических изменений, мы развертываем новую версию сервиса, которая выставляет как старую, так и новую версию конечной точки.

Это позволяет нам получить новый микросервис как можно быстрее и с новым интерфейсом, но дает потребителям время на раскачку. Как только все потребители полностью откажутся от использования старой конечной точки, ее можно будет удалить вместе со всем связанным с ней кодом (рис. 4.5).

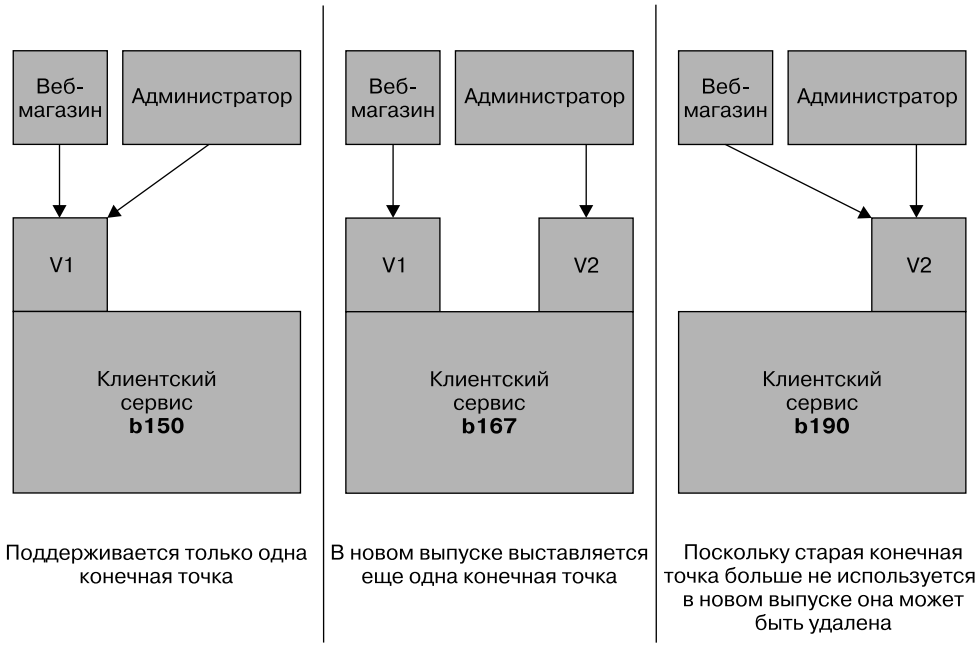


Рис. 4.5. Сосуществование различных версий конечных точек, позволяющее клиентам осуществлять постепенный переход

Когда я в последний раз использовал этот подход, мы слегка запутались с количеством имеющихся у нас потребителей и внесенных критических изменений. Это означало, что у нас фактически сосуществовали три различные версии конечных точек. Я бы такое не стал рекомендовать! Дополнительно нагружать себя поддержкой всего требующегося для этого кода и проведением связанного с ним тестирования, призванного убедить в том, что все это работает, совершенно ни к чему. Чтобы привести все в более управляемое состояние, мы внутренне перевели все запросы к конечной точке V1 в запросы к конечной точке V2, а затем все запросы к V2 — в запросы к конечной точке V3. Это означало, что мы могли четко очертить тот код, который подлежал удалению при выходе из употребления той или иной прежней конечной точки.

По сути, это является примером применения шаблона расширения и свертывания (expand and contract pattern), допускающего постепенный ввод критических

изменений. Мы расширяем предлагаемые возможности, поддерживая как старый, так и новый путь получения какого-либо результата, и как только старые потребители станут работать по-новому, мы свертываем часть нашего API, удаляя старые функциональные возможности.

Если вы намереваетесь допустить сосуществование конечных точек, то для этого потребуется способ соответствующего перенаправления запросов вызывающих сторон. Я видел, как это делается для систем, использующих HTTP, с помощью указания номеров версий как в заголовках запросов, так и в самих URI, например /v1/customer/ или /v2/customer/. Я не могу сказать, какой из этих подходов рациональнее. С одной стороны, мне не нравятся сложные URI-индикаторы, поскольку я не хочу заставлять клиентов пользоваться жестко закодированными URI-шаблонами, но с другой — такой подход делает многое весьма очевидным и может упростить маршрутизацию запросов.

При использовании RPC все может оказаться несколько сложнее. Я справлялся с этой задачей с помощью буферов протокола, помещая свои методы в различные пространства имен, например `v1.createCustomer` и `v2.createCustomer`, но когда делается попытка поддержки различных версий одних и тех же типов, отправляемых по сети, возникают серьезные трудности.

Использование нескольких параллельных версий сервиса

Еще одним часто упоминаемым решением для управления версиями является сосуществование различных версий сервиса, при котором прежние потребители направляют свой трафик к прежней версии, а новые потребители видят новую версию (рис. 4.6). Такой подход весьма умеренно используется в компании Netflix в ситуациях, когда цена внесения изменений в системы прежних потребителей слишком высока, особенно в тех редких случаях, когда устаревшие устройства все еще привязаны к старым версиям API. Лично я такую идею не приветствую и понимаю, почему Netflix пользуется этим приемом крайне редко. Во-первых, если в моем сервисе нужно исправить какой-либо внутренний дефект, я знаю, что исправление и развертывание нужно провести в отношении двух различных наборов сервисов. Весьма вероятно, что для этого понадобится разветвлять исходный код моего сервиса, что всегда вызывает проблемы. Во-вторых, это означает, что необходимо придумать способ направления потребителей к нужному им микросервису. Связанные с этим функции в итоге неизбежно должны попасть в какую-либо связующую программу или в пакет Nginx-сценариев, затрудняя тем самым понимание причин поведения системы. И наконец, следует рассматривать любое постоянное состояние, которым может управлять наш сервис. Клиенты, создаваемые любой из версий сервиса, должны быть сохранены и должны стать видимыми всем сервисам независимо от того, какая из версий была первоначально использована для создания данных. Это может стать дополнительным источником затруднений.

Сосуществование параллельных версий сервисов на непродолжительное время может быть вполне оправданно, особенно при синих и зеленых развертываниях

или канареечных выпусках (более подробно эти схемы рассматриваются в главе 7). В таких ситуациях параллельно работающие версии могут существовать всего несколько минут или, возможно, часов, и обычно это будут только две разные версии сервиса. Чем больше времени займут обновление, выполняемое потребителями до более новой версии, и выпуск этой версии, тем больше следует склоняться к сосуществованию различных конечных точек в одном и том же микросервисе, а не к сосуществованию совершенно разных версий. Я по-прежнему не думаю, что такую работу стоит делать при выполнении обычного проекта.

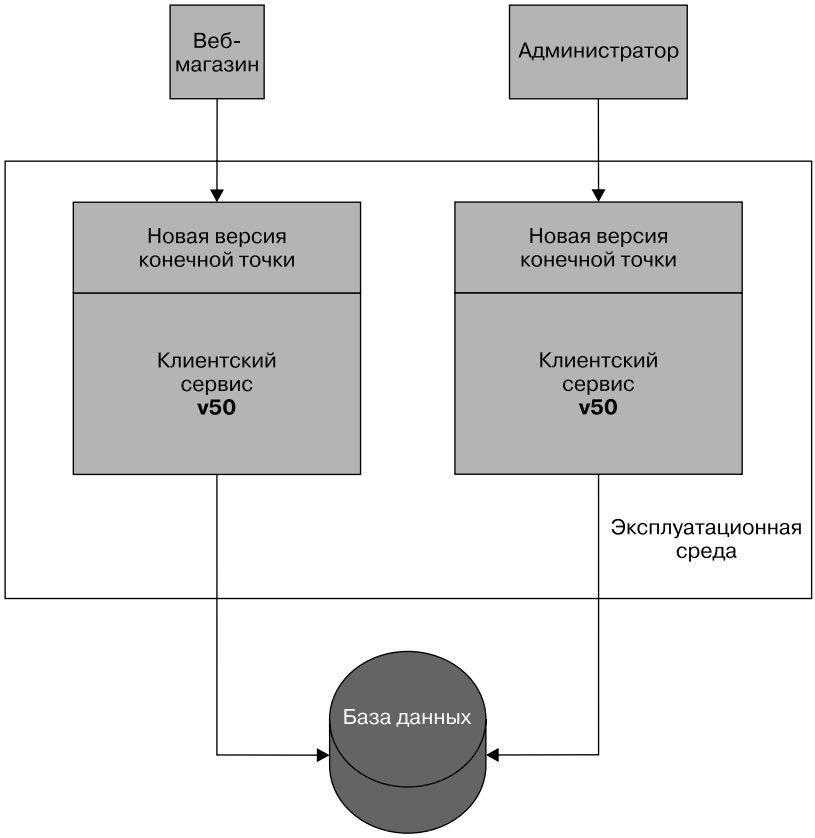


Рис. 4.6. Выполнение нескольких версий одного и того же сервиса с целью поддержки старых конечных точек

Пользовательские интерфейсы

До сих пор мы еще всерьез не касались пользовательского интерфейса. Возможно, некоторые из нас и предоставляют своим клиентам весьма неприглядный, строгий и минималистичный интерфейс, но многие предпочитают создавать красивые и функциональные пользовательские интерфейсы, способные обрадовать

клиентов. Но в действительности мы должны думать о них в контексте интеграции. Все же пользовательский интерфейс является местом сбора всех микросервисов в нечто, имеющее смысл для наших клиентов.

В прошлом, когда я только начал заниматься программированием, разговоры шли главным образом о солидных толстых клиентах, работающих за настольными компьютерами. Я проводил много времени с Motif, а затем со Swing, стараясь максимально украсить свои программы. Зачастую системы предназначались всего лишь для создания локальных файлов и работы с ними, но у многих из них имелся компонент серверной стороны. Моя первая работа в ThoughtWorks заключалась в создании системы электронной торговой точки на основе Swing, которая была всего лишь фрагментом большого количества подвижных частей, многие из которых находились на сервере.

Затем пришло время Интернета. И мы стали продумывать пользовательские интерфейсы в более скромных красках с присутствием основной логики на серверной стороне. Поначалу наши программы на серверной стороне выдавали целиком всю страницу и отправляли ее клиентскому браузеру, работа которого сводилась к минимуму. Любое взаимодействие обрабатывалось на серверной стороне посредством GET- и POST-запросов, инициируемых после щелчка пользователя на ссылках или заполнения форм. Со временем наиболее популярным средством придания динамичности пользовательскому интерфейсу на основе использования браузера стал язык JavaScript, и тогда появилась возможность придать некоторым приложениям прежний солидный вид, который был свойственен старым клиентским приложениям на настольных компьютерах.

Движение в направлении к единым цифровым устройствам

За последние пару лет организации стали изменять свое представление о том, что интернет-системы и мобильные устройства должны рассматриваться в разных ключах, в результате чего появилось некое цельное представление обо всех цифровых устройствах. Так какой же наилучший способ использования сервисов мы можем предложить клиентам? И что для этого нужно сделать с нашей системной архитектурой? Осознание того, что мы не можем в точности предсказать, как в итоге клиент будет взаимодействовать с нашей компанией, привело к принятию на вооружение API с более мелкой структурой, такой, какую могут предоставить микросервисы. Путем объединения различными способами возможностей, которые выставляются нашими сервисами напоказ, мы можем курировать различные виды восприятия наших клиентов, возникающие при использовании ими приложений для настольных компьютеров, мобильных устройств, переносных устройств, или даже предоставлять их в физической форме, если клиенты посетят наш магазин стройматериалов.

Следовательно, пользовательские интерфейсы нужно рассматривать в виде композиционных уровней, то есть мест, в которых мы сплетаем в единое целое различные ветви предлагаемых нами возможностей. Итак, памятуя о сказанном, как же все-таки мы можем сплести все это в единое целое?

Ограничения

Ограничения представляют собой различные формы взаимодействия пользователей с нашей системой. Например, при использовании веб-приложения для настольной системы мы рассматриваем ограничения в виде того, чем пользуются посетители браузера или каким разрешением обладает его экран. А вот мобильные устройства приносят нам целый ряд новых ограничений. Влияние может оказать тот способ, который используется нашими мобильными приложениями для обмена данными с сервером. Свою роль может сыграть даже ограничение пропускной способности мобильной сети, которое заставит задуматься именно о ширине полосы пропускания. Различные виды взаимодействия могут приводить к быстрой разрядке батарей, отсекая тем самым часть потребителей.

Изменяется и характер взаимодействий. Я, к примеру, не могу так же запросто, как на простых компьютерах, щелкнуть правой кнопкой мыши, если работаю на планшете. А для мобильного телефона у меня может возникнуть желание разработать интерфейс под преимущественное управление одной рукой, чтобы большинство операций можно было вызвать с помощью манипуляций большим пальцем. А где-то еще, в местах с очень высокой платой за ширину полосы пропускания, например в странах со средним уровнем развития, где SMS используются в качестве интерфейса весьма часто, я могу позволить людям взаимодействовать с сервисами с помощью SMS.

Итак, несмотря на то, что наши основные сервисы или же основные предложения могут быть одинаковыми, нужен способ их адаптации под различные ограничения, существующие для каждого типа интерфейса. Рассматривая различные стили композиций пользовательского интерфейса, мы должны удостовериться в том, что они отвечают решению этой непростой задачи. Рассмотрим несколько моделей пользовательских интерфейсов, чтобы посмотреть, как можно будет достичь желаемого результата.

API-композиция

Предположим, что наши сервисы уже общаются друг с другом посредством XML или JSON через HTTP и очевидный доступный нам вариант заключается в непосредственном взаимодействии пользовательского интерфейса с теми API, которые показаны на рис. 4.7. В пользовательском интерфейсе на основе веб-технологий для извлечения данных можно воспользоваться написанными на JavaScript GET-запросами, а для изменения этих данных можно воспользоваться POST-запросами. Даже для чисто мобильных приложений инициировать обмен данными по протоколу HTTP довольно легко. Для пользовательского интерфейса затем придется создать различные компоненты, составляющие интерфейс, справляющиеся с синхронизацией состояния и тому подобным с сервером. Если для обмена данными между серверами использовался двоичный протокол, ситуация может усложниться для клиентов, работающих на основе веб-технологий, но при этом может вполне подойти для чисто мобильных устройств.

Рис. 4.7. Использование нескольких API для представления пользовательского интерфейса

Но у этого подхода есть ряд недостатков. В первую очередь это ограниченный круг возможностей для адаптации ответов под устройства различных типов. Например, должен ли я при извлечении записи клиента вытаскивать все те же данные для мобильного магазина, что и для приложения службы поддержки клиентов? При таком подходе одним из решений может стать разрешение пользователям указывать, какие поля извлекать при выдаче ими запроса, но это предполагает, что такой вид взаимодействия должен поддерживаться всеми сервисами.

И еще один ключевой вопрос: кто создает пользовательский интерфейс? Тех, кто приглядывает за сервисами, не беспокоит то, как их сервисы появляются перед пользователями. Например, если пользовательский интерфейс создает другая команда, мы можем вернуться к прежним не самым благополучным временам многоуровневой архитектуры, когда внесение даже самых незначительных изменений вело к перемене требований сразу к нескольким командам.

Этот обмен данными может быть слишком многословным. Открытие множества вызовов непосредственно к сервисам может оказаться слишком обременительным для мобильных устройств и привести к весьма неэффективному расходу мобильного плана клиента! Здесь может помочь наличие API-шлюза, поскольку вы сможете выдать вызовы, объединяющие несколько исходных вызовов, хотя само по себе это может иметь ряд недостатков, которые мы вскоре рассмотрим.

Составление фрагментов пользовательского интерфейса

Вместо того чтобы пользовательский интерфейс занимался API-вызовами и отображал все обратно на свои элементы управления, мы можем сделать так, чтобы сервисы предоставляли части пользовательского интерфейса напрямую, а затем, как показано на рис. 4.8, просто вставить эти фрагменты с целью создания пользовательского интерфейса. Представьте себе, к примеру, что рекомендационный сервис предоставляет рекомендационный виджет, который с целью создания полноценного пользовательского интерфейса объединяется с другими элементами управления или фрагментами этого интерфейса. Он может отображаться на веб-странице наряду с другим ее содержимым в виде блока.

Разновидностью такого подхода, который может вполне достойно справиться со своей работой, является сборка крупномодульных частей пользовательского интерфейса. Здесь вместо создания небольших виджетов собираются вместе целые панели солидного клиентского приложения или, возможно, набор страниц для сайта.

Эти довольно крупные фрагменты подаются приложениями серверной стороны, которые, в свою очередь, делают соответствующие API-вызовы. Эта модель лучше всего работает, когда фрагменты точно распределяются по принадлежности между командами. Например, возможно, команда, которая несет ответственность за управление заказами в музыкальном магазине, обслуживает все страницы, связанные с управлением заказами.

Для того чтобы собрать все эти части вместе, понадобится некий сборочный уровень. Этот вопрос может быть решен простым созданием шаблонов на серверной

Рис. 4.8. Сервисы, непосредственно обслуживающие компоненты пользовательского интерфейса, предназначенные для создания сборки

стороне, или же там, где каждый набор страниц поставляется другим приложением, вам, наверное, потребуется некая интеллектуальная URI-маршрутизация.

Одним из ключевых преимуществ этого подхода является то, что та же команда, которая вносит изменения в сервисы, может также заниматься внесением изменений в соответствующие части пользовательского интерфейса. Это позволяет ускорить получение изменений. Но с этим подходом все же связаны некоторые проблемы.

В первую очередь нужно обратить внимание на обеспечение соответствия пользовательского восприятия. Пользователям хочется получать цельное восприятие, чтобы у них не возникало ощущения, что разные части интерфейса работают по-разному или что они представляют разные языки дизайна. Но существуют технологии, позволяющие обойти эту проблему, например действенные стилевые ориентиры (living style guides), где такие ресурсы, как HTML-компоненты, CSS и изображения, могут использоваться совместно, содействуя тем самым выдерживанию определенного уровня взаимного соответствия.

А вот с другой проблемой справиться сложнее. Что происходит с чистыми приложениями или полноценными клиентами? Мы не можем обслуживать компоненты пользовательского интерфейса. Можно применить гибридный подход и использовать для обслуживания HTML-компонентов чистые приложения, но в этом подходе постоянно обнаруживаются недостатки. Поэтому, если вам требуется получить естественное восприятие, придется вернуться назад, к подходу, при котором интерфейсное приложение самостоятельно осуществляет API-вызовы и управляет пользовательским интерфейсом. Но даже если рассматривать только пользовательские интерфейсы на основе веб-технологий, все равно может потребоваться иметь совершенно разную трактовку для разного типа устройств. Разумеется, помочь в данном вопросе может создание отзывчивых компонентов.

У этого подхода имеется еще одна ключевая проблема, в вероятности решения которой я не уверен. Иногда возможности, предлагаемые сервисом, не вписываются в виджет или страницу. Конечно, мне может потребоваться выдать какие-либо общие рекомендации в блоке страницы нашего сайта, а что, если я захочу создать систему динамических рекомендаций где-либо в другом месте? При поиске я, к примеру, хочу, чтобы набираемый текст автоматически продолжался выводом новых рекомендаций. Чем больше сквозных форм взаимодействия, тем меньше надежд на то, что эта модель подойдет, и больше подозрений, что придется вернуться назад, к простой выдаче API-вызовов.

Внутренние интерфейсы, предназначенные для внешних интерфейсов

Обычным решением проблемы многословных интерфейсов внутренних сервисов, или проблемы, связанной с необходимостью изменения содержимого для разных типов устройств, является использование объединяющей конечной точки на серверной стороне, или API-шлюза. Тем самым можно будет выстраивать несколько внутренних вызовов в случае необходимости изменять и собирать содержимое для

различных устройств и, как показано на рис. 4.9, обслуживать все это. Я видел, что подобные конечные точки на серверной стороне, становясь довольно мощными уровнями со слишком развитым поведением, приводили к катастрофе. Все заканчивалось тем, что ими управляли различные команды разработчиков и они становились еще одним местом, где функциональные изменения приводили к необходимости внесения изменений в логику работы.

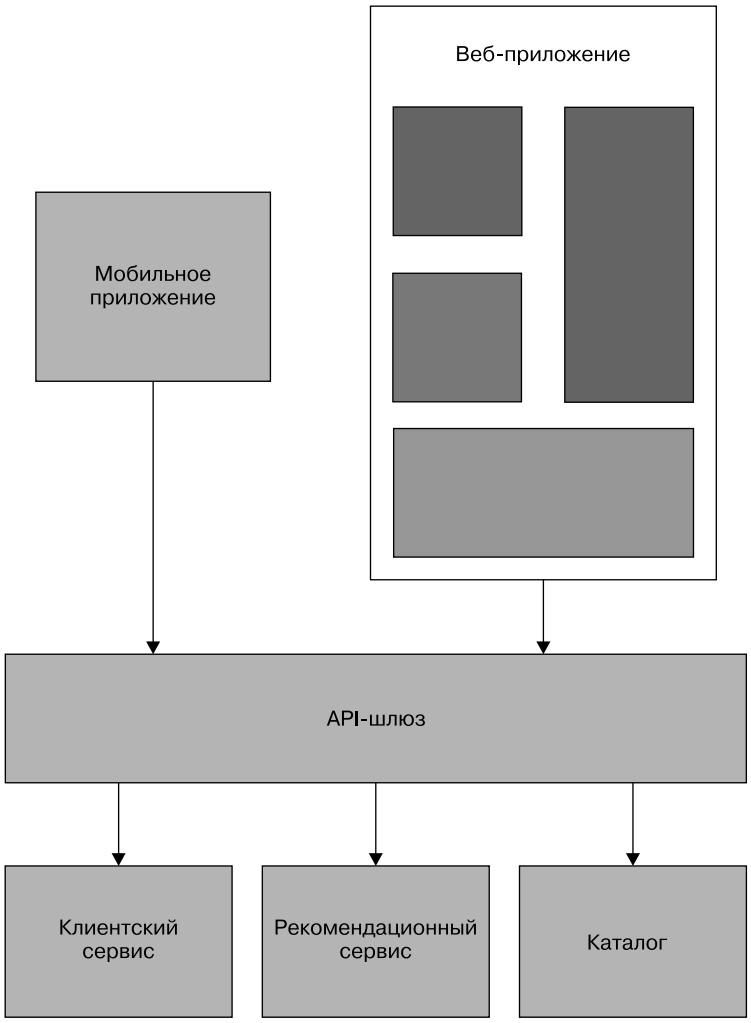


Рис. 4.9. Использование единого монолитного шлюза для управления вызовами из пользовательских интерфейсов и к самим этим интерфейсам

Может возникнуть проблема, связанная с тем, что у нас вполне естественным образом получится просто гигантский уровень для всех наших сервисов. Это приведет к тому, что все будет свалено в кучу и мы внезапно начнем терять изолированность различных пользовательских интерфейсов, что ограничит возмож-

ности независимой реализации. Я отдаю предпочтение работоспособной, на мой взгляд, модели (рис. 4.10), которая заключается в сведении использования таких внутренних интерфейсов к одному конкретному пользовательскому интерфейсу или приложению.

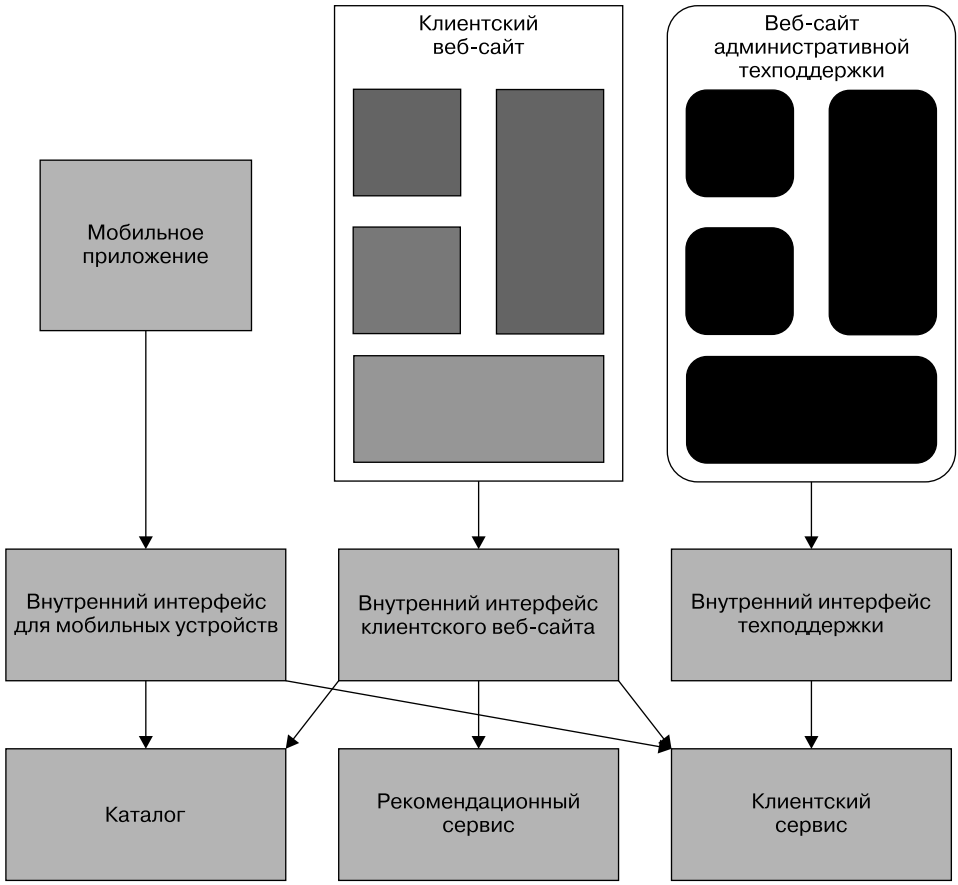


Рис. 4.10. Использование для внешних интерфейсов предназначенных конкретно для них внутренних интерфейсов

Эту схему иногда называют внутренними интерфейсами для внешних интерфейсов (backends for frontends (BFF)). Она позволяет команде, отвечающей за любой отдельно взятый пользовательский интерфейс, вдобавок ко всему обслуживать его собственные компоненты, находящиеся на стороне сервера. Эти внутренние интерфейсы можно рассматривать как часть пользовательского интерфейса, встроенную в сервер. Некоторые типы пользовательских интерфейсов требуют минимальной площади опоры на сервере, а некоторым нужна опора посолиднее. Если нужен уровень API-аутентификации и авторизации, то он может располагаться между BFF-интерфейсами и пользовательскими интерфейсами. Более подробно этот вопрос рассматривается в главе 9.

Опасности, подстерегающие при выборе этого подхода, аналогичны тем, которые связаны с любым объединяющим уровнем: он может содержать логику, которой в нем быть не должно. Бизнес-логика для различных возможностей, используемых этими внутренними интерфейсами, должна содержаться в самих сервисах. Эти BFF-интерфейсы должны обладать только поведением, характерным для создания конкретного пользовательского восприятия.

Гибридный подход

Многие из вышеупомянутых вариантов не должны целиком располагаться только на одной стороне. Мне приходилось видеть организации, взявшие на вооружение для создания сайтов подход, заключающийся в сборке на основе фрагментов, но при этом, когда дело касалось их мобильных приложений, использовалось создание внутренних интерфейсов для внешних интерфейсов. Главное здесь — сохранять единство основных возможностей, предлагаемых пользователям. Нам нужно обеспечить нахождение логики, связанной с заказом музыки или изменением данных о клиентах внутри тех сервисов, которые занимаются этими операциями, и не позволять ей размываться по всей нашей системе. Нужно, соблюдая искусный баланс, избежать ловушки, возникающей при помещении в промежуточные уровни слишком большого объема поведенческой логики.

Интеграция с программами сторонних разработчиков

Мы рассмотрели подходы разбиения на части существующих систем, находящихся в нашем ведении. А как быть с теми системами, в которые мы не можем вносить изменения, но с которыми вынуждены вести информационный обмен? По многим весьма уважительным причинам организации, для которых мы работаем, приобретают готовые коммерческие программы (commercial off-the-shelf software (COTS)) или пользуются программами в виде сервисов (software as a service (SaaS)), предлагающими услуги, управление которыми с нашей стороны имеет весьма ограниченный характер. Так как же провести разумную интеграцию с такими системами?

Если вы читаете эту книгу, то, наверное, работаете в организации, создающей программный код. Вы можете разрабатывать программы для собственных внутренних целей, или для внешнего клиента, или и для того и для другого. Тем не менее, даже если вы представляете организацию, способную создавать существенные объемы заказных программ, вы все равно пользуетесь программными продуктами, предоставляемыми внешними сторонами, будь то коммерческие продукты или программы с открытым кодом. Почему именно так это и происходит?

Во-первых, ваша организация почти наверняка испытывает большие потребности в программных средствах, которые нельзя удовлетворить своими силами. Подумайте обо всех продуктах, которыми пользуетесь, от инструментов, приме-

няемых в офисах, типа Excel до операционных систем и систем начисления заработной платы. Создание всего этого для внутреннего потребления может стать непосильной затеей. Во-вторых, что более важно, это будет экономически невыгодно! Например, стоимость создания собственной системы электронной почты будет значительно больше стоимости использования существующих сочетаний почтового сервера и клиента, даже если это будут коммерческие варианты.

Мои клиенты часто задаются вопросом: «Создавать или покупать?» Обычно, когда происходит подобный разговор с обычной предпринимательской организацией, я и мои коллеги даем совет, который сводится к следующему: «Создавать, если то, что вы сделаете, будет уникальным и может считаться стратегическим активом, и покупать, если нужный инструмент к данной категории не относится».

Например, используемая в обычной организации система начисления заработной платы может не считаться стратегическим активом. Людям во всем мире начисляют зарплату одинаково. Точно так же большинство организаций склоняются к приобретению готовых систем управления контентом (CMSes), если использование подобного инструментария не рассматривается как что-то ключевое по отношению к их бизнесу. Однако в прежние времена меня привлекали к переделке сайта Guardian, и было принято решение создать заказную систему управления контентом, поскольку она была основой газетного бизнеса.

Поэтому намерение временами использовать коммерческие программные продукты сторонних производителей вполне обоснованно и может только приветствоваться. Но многие из нас в конечном счете проклинают некоторые из таких систем. Почему же так происходит?

Отсутствие должного контроля

Одна из проблем, связанных с объединением с COTS-продуктами CMS- или SaaS-инструментов и расширением их возможностей, заключается в том, что многие технические решения, как правило, уже были сделаны для вас. Как интегрироваться с инструментом? Это решение поставщика. Каким языком программирования можно воспользоваться для расширения возможностей инструмента? Это зависит от поставщика. Можно ли сохранить конфигурацию инструмента в системе управления версиями и восстановить ее с нуля, чтобы иметь возможность непрерывной интеграции в настройках? Это зависит от выбора, сделанного поставщиком.

Если вам повезет, то вопрос о том, насколько легко или трудно работать с инструментом с точки зрения разработчика, будет рассмотрен как часть процесса выбора инструмента. Но даже при том вы фактически уступаете некоторый уровень контроля внешней стороне. Вся хитрость заключается в том, чтобы вернуться к проведению интеграции и адаптации на своих условиях.

Адаптация

Многие инструментальные средства, приобретаемые предпринимательскими организациями, продаются с возможностью глубокой адаптации *именно под ваши*

потребности. Осторожно! Зачастую из-за самой природы цепочки инструментов, к которой у вас есть доступ, стоимость адаптации может быть значительно выше создания с нуля какого-нибудь продукта на заказ! Если вы решили приобрести продукт, чьи конкретные возможности не предназначены специально для вас, то, может быть, более разумным решением будет подстроить под него порядок работы организации, чем заниматься сложной адаптацией этого продукта под свои нужды.

Хорошим примером опасности подобного рода могут послужить системы управления контентом. Мне приходилось работать с несколькими CMS, которые по своей конструкции не поддерживали непрерывную интеграцию, у которых были ужасные API и в которых даже несущественное обновление исходного инструментария могло разрушить любые сделанные вами настройки.

Наиболее проблемным в этом смысле является продукт компании Salesforce. На протяжении многих лет компания проталкивала свою платформу Force.com, которая требовала использования языка программирования Apex, существовавшего только внутри экосистемы Force.com!

Тонкости интеграции

Еще одной проблемой является порядок интеграции с инструментальным средством. Как уже говорилось, важно весьма тщательно продумать порядок интеграции между сервисами, а в идеале желательно вывести стандарты в отношении весьма небольшого количества типов интеграции. Но если для одного продукта решено использовать собственный двоичный протокол, для другого предпочтение отдано SOAP, а для третьего выбрана технология XML-RPC, то что делать? Еще хуже те инструменты, которые позволяют вам добираться до их базовых хранилищ данных, что вызывает те же проблемы связанности, о которых уже говорилось.

На наших собственных условиях

Продукты COTS и SaaS занимают свое место по праву, и невозможно, да и неразумно большинству из нас создавать все с нуля. Так как же все-таки решить все эти проблемы? Суть заключается в том, чтобы делать все на своих условиях.

Основная идея состоит в том, чтобы производить всю адаптацию на той платформе, которой вы можете управлять, и в том, чтобы ограничить количество различных потребителей самого инструментального средства. Чтобы подробно исследовать эту идею, рассмотрим два примера.

Пример: CMS в качестве сервиса

Мой опыт свидетельствует: CMS является наиболее часто используемым продуктом, нуждающимся в адаптации или создании с ним интеграции. Причина в том, что, если не нужен простой статичный сайт, обычная предпринимательская организация желает обогатить функциональность своего сайта динамическим содержимым вроде клиентских записей или предложений о приобретении продуктов из самых последних поступлений. Источником этого динамичного содержимого

обычно являются другие сервисы внутри организации, возможно, созданные собственными силами.

Соблазн, а зачастую и привлекательность, возникающие при продаже CMS, состоят в том, что вы можете адаптировать CMS для того, чтобы она вбирала в себя это специализированное содержимое и демонстрировала его всему внешнему миру. Но разработочная среда для обычной CMS оставляет желать лучшего.

То, на чем специализируется обычная CMS и для чего мы ее, возможно, приобретаем, — это создание контента и управление им. Большинство CMS весьма посредственно справляются даже с макетированием страниц, обычно предоставляя инструменты для перетаскивания, которые не подслащивают эту горькую пилюлю. И даже при этом вы сталкиваетесь с необходимостью иметь кого-то, кто разбирается в HTML и CSS для подстройки CMS-шаблонов. Платформа для создания пользовательского кода из них, как правило, никудышная.

Так как же быть? Поставить впереди CMS собственный сервис, представляющий сайт внешнему миру (рис. 4.11). Считайте CMS сервисом, чья роль состоит в том, чтобы позволить ему создавать контекст и возвращать его. В собственном сервисе вы пишете код и интегрируете его с сервисами по своему усмотрению. У вас есть контроль над масштабированием сайта (чтобы справиться с нагрузкой, многие коммерческие CMS предоставляют собственные дополнительные компоненты), и вы можете выбрать систему создания шаблонов, имеющую для вас определенный смысл.

Многие CMS также предоставляют API, позволяющие создавать контент, поэтому у вас есть возможность поставить впереди них фасад из собственного сервиса. В некоторых ситуациях мы даже использовали такой фасад для отвлечения от API, предназначенного для извлечения контента.

В последние несколько лет мы неоднократно использовали данную схему в компании ThoughtWorks, и сам я делал это не один раз. Одним примечательным примером был клиент, искавший возможность выпустить новый сайт для своих продуктов. Сначала он хотел сделать все на CMS, но ему еще предстояло выбрать, на какой именно системе это делать. Вместо этого мы предложили ему рассматриваемый здесь подход и приступили к разработке лицевого сайта. В ожидании выбора CMS мы имитировали такую систему с помощью веб-сервиса, который просто выставлял наружу статический контент. В итоге у нас при использовании сервиса-имитатора контекста, производящего выставляемое для живого сайта содержимое, получился вполне работоспособный сайт еще до выбора CMS. Чуть позже мы смогли просто вставить выбранный наконец-то инструментарий, не внося никаких изменений в лицевое приложение.

Используя данный подход, мы свели к минимуму работу CMS и переместили адаптацию в собственный технологический стек.

Пример: многоцелевая CRM-система

CRM, или Customer Relationship Management, то есть система управления взаимосвязями с клиентами, — это часто встречающийся инструмент, считающийся неким чудовищем, способным вселить страх в душу даже самого отважного архитектора. Этот сектор, судя по характеристикам таких поставщиков, как Salesforce

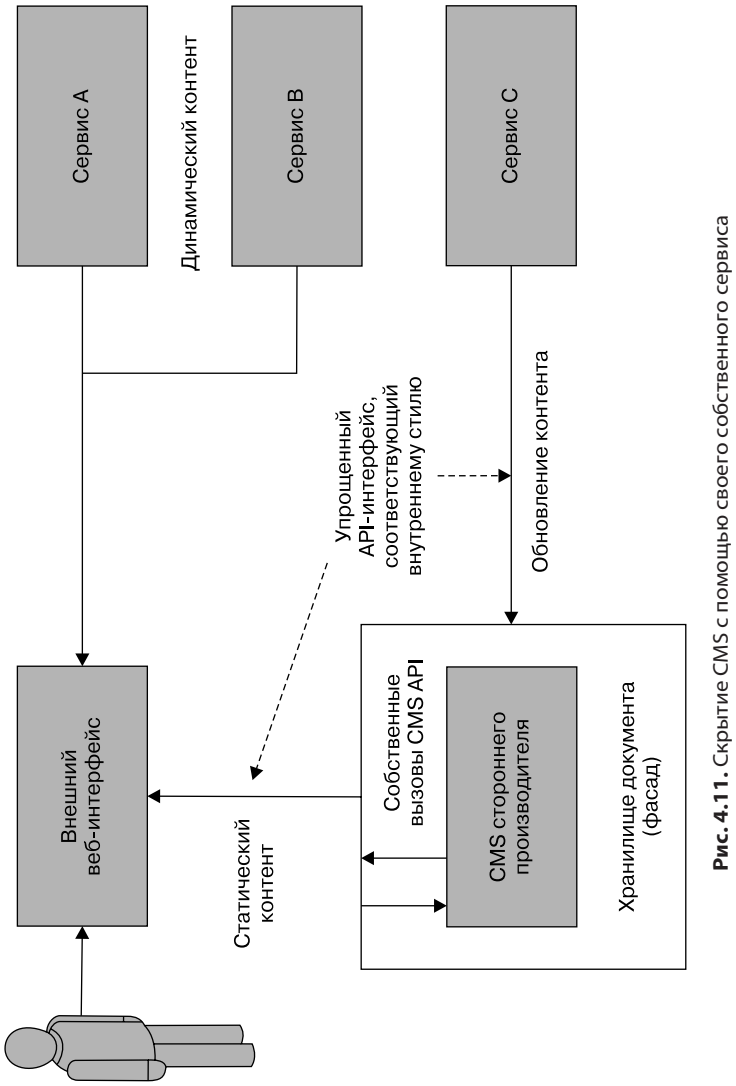


Рис. 4.11. Скрытие CMS с помощью своего собственного сервиса

или SAP, изобилует примерами инструментов, пытающихся все делать за вас. Это может привести к тому, что сам инструмент может стать и единственной точкой сбоя, и запутанным узлом зависимостей. Многие реализации CRM-инструментов, попадавшиеся мне на глаза, являли собой массу наилучших примеров *связанных* (в отличие от сцепленных) сервисов.

Обычно поначалу масштабы применения такого инструмента невелики, но со временем он становится все более важной частью стиля работы вашей организации. Проблема в том, что направления и варианты в рамках этой теперь уже жизненно важной для вас системы зачастую выбираете не вы, а сам поставщик инструмента.

Недавно я участвовал в попытке возвращения управления. Организация, с которой я работал, пришла к выводу, что, хотя CRM-инструментарий использовался в ней для решения многих задач, от увеличения стоимости платформы особой выгоды они не получали. В то же время несколькими внутренними системами для интеграции использовались далеко не идеальные API CRM. Мы хотели переместить архитектуру системы в то место, в котором имелись сервисы, моделирующие нашу область бизнеса, а также заложить основу для потенциальной миграции.

Сначала мы определили для нашей области основные концепции, которыми уже владела CRM-система. Одной из них была концепция проектов, то есть то, что может быть назначено штатному сотруднику. Проектная информация нужна была нескольким другим системам. То, чем мы занимались, заменяло сервис проектов. Этот сервис выставлял проекты в виде RESTful-ресурсов, и внешние системы могли перемещать свои точки интеграции на новый сервис, с которым было легче работать. Внутри сервис проектов представлял собой всего лишь фасад, за которым скрывались детали основной интеграции. Все это можно увидеть на рис. 4.12.

Работа, которая в момент написания этих строк еще продолжалась, заключалась в определении в нужной области других понятий, с которыми справлялась система CRM, и создании скорее фасадов для них. Когда настанет время уйти от базовой CRM, можно будет по очереди пересмотреть каждый фасад, чтобы решить, есть ли соответствующие требованиям внутренние программные решения или что-либо из готовых программных продуктов.

Шаблон Strangler (Дроссель)

Когда дело доходит до устаревших или даже COTS-платформ, которые находятся полностью под вашим контролем, приходится справляться с ситуациями, при которых нужно их удалить или по крайней мере от них отойти. В таком случае пригодится шаблон под названием Strangler Application Pattern. Во многом подобно примеру выстраивания лицевой части CMS с помощью собственного кода, с использованием шаблона Strangler добывают и перехватывают вызовы к старой системе. Это дает возможность принять решение, нужно ли направлять эти вызовы существующему устаревшему коду или же направлять их к новому коду, который вы могли написать. Это позволяет со временем заменить функциональные свойства, не требуя для этого больших переделок кода.

Рис. 4.12. Использование фасадных сервисов в качестве маскировки основного CRM

Когда же дело касается микросервисов, то для выполнения перехвата вместо использования одного монолитного приложения, перехватывающего все вызовы к существующей устаревшей системе, можно воспользоваться серией микросервисов. Захват и перенаправление исходных вызовов может оказаться в этой ситуации намного сложнее, что может потребовать использования прокси-сервиса, делающего все это за вас.

Резюме

Мы рассмотрели ряд вариантов интеграции, и я поделился своими размышлениями о том, какие решения могут скорее всего обеспечить то, что наши микросервисы останутся как можно более разобщенными со всем, с чем они совместно работают.

- Любой ценой избегайте интеграции с помощью баз данных.
- Разберитесь с компромиссами между REST и RPC, но как следует присмотритесь к REST как к хорошей стартовой точке для интеграции по схеме «запрос — ответ».
- Отдавайте предпочтение хореографическому, а не оркестровому принципу.
- Избегайте критических изменений и необходимости прибегать к управлению версиями, разобравшись с законом Постела и использованием толерантных считывателей.
- Подумайте о пользовательских интерфейсах как о композиционных уровнях.

Здесь было рассмотрено множество тем, углубиться в которые мы просто не могли. Тем не менее это может послужить неплохой основой для задания вашему пути верного направления, если вы захотите продолжить изучение.

Мы также потратили время на рассмотрение порядка работы с системами, которые не полностью нами контролируются и относятся к COTS-продуктам. Оказывается, данное описание может быть с легкостью применено и к программам, которые мы пишем сами!

Некоторые из показанных здесь подходов могут с одинаковой эффективностью применяться и к устаревшим программным продуктам, но что делать, если нужно решить такие проблемы, как подчинение устаревших систем своим интересам и их разбиение на более полезные для нас части? Подробно данный вопрос рассматривается в следующей главе.

5 Разбиение монолита на части

Мы уже выясняли, на что похож хороший сервис и почему более мелкие сервисы могут подойти лучше. Также рассмотрели важность получения возможности развития конструкций наших систем. Но как справиться с тем, что уже может существовать большой объем исходных кодов, по своей сути не отвечающих принятым нами схемам? Как справиться с декомпозицией этих монолитных приложений, не ввязываясь в широкомасштабное переписывание кода?

Со временем монолит разрастается. Он с устрашающей скоростью обзаводится новыми функциональными возможностями и новыми строками кода. Вскоре он становится большим, ужасным гигантом, живущим в нашей организации, страшно к нему прикасаться или вносить в него изменения. Но еще не все потеряно! Имея в своем распоряжении нужные инструменты, мы можем убить этого зверя.

Все дело в стыках

В главе 3 мы согласились с тем, что наши сервисы должны обладать слабой связанностью и сильным зацеплением. Проблема монолита в том, что зачастую он обладает прямо противоположными качествами. Вместо стремления к сильному зацеплению и группировке, вместо всего того, что обычно изменяется вместе, мы получаем и сцепляем всевозможный неродственный код. Слабая связанность также практически отсутствует: как только понадобится внести изменения в строку кода, это можно будет сделать довольно легко, но я не могу выполнить развертывание этого изменения без потенциального распространения влияния на основную часть монолита, и мне, несомненно, придется заново развертывать всю систему.

Майкл Физерс (Michael Feathers) в своей книге *Working Effectively with Legacy Code* (Prentice-Hall) дал определение понятия *стыка* как порции кода, которая не может рассматриваться изолированно и работать, не влияя на весь остальной исходный код. Нам также нужно дать определение стыкам. Но вместо поиска определения для более четкого понимания исходного кода нужно определить стыки, которые могут превратиться в границы сервисов.

Итак, по каким же критериям можно определить хороший стык? Как уже говорилось, превосходными стыками могут послужить ограниченные контексты, поскольку по определению они представляют собой сильно зацепленные, но все же

слабо связанные границы внутри организации. Следовательно, первым шагом должно стать определение этих границ в нашем коде.

В большинстве языков программирования имеется понятие пространства имен, позволяющее группировать вместе соответствующий код. Понятие из пакета Java представляет собой, конечно, довольно слабый пример, но, по большому счету, соответствует нашим потребностям. Все остальные широко распространенные языки программирования имеют сходные встроенные понятия, и только JavaScript, вероятно, является исключением.

Разбиение MusicCorp на части

Представим себе, что имеется большой внутренний монолитный сервис, определяющий основное поведение онлайн-систем MusicCorp. Для начала в соответствии с приемами, рассмотренными в главе 3, нужно определить границы ограниченных контекстов высокого уровня, которые, как мы понимаем, имеются в организации. Затем нужно будет попытаться понять, на какие ограниченные контексты отображается монолит. Представим, что изначально были определены четыре контекста, которые охватывает монолитный внутренний сервис.

- **Каталог.** Все, что касается метаданных товарных позиций, предлагаемых на продажу.
- **Финансы.** Отчеты по счетам, платежам, возмещению убытков и т. д.
- **Товарный склад.** Отправка и возвращение заказов клиентов, управление уровнем запасов и т. д.
- **Рекомендации.** Ожидающая патентования революционная система выдачи рекомендаций, представляющая собой весьма сложный код, написанный командой, в которой больше кандидатов наук, чем в обычной научной лаборатории.

Сначала нужно создать пакеты, представляющие эти контексты, а затем переместить в них существующий код. Используя современные IDE-среды, переместить код можно автоматически посредством рефакторинга, и сделать это пошагово, занимаясь другими делами. Но, чтобы отловить повреждения, возникающие в связи с перемещением кода, нужно все же проводить тестирование, особенно если используется язык с динамической типизацией, в котором IDE-средам выполнять рефакторинг довольно трудно. Со временем мы начинаем замечать, какой код поддается этому легче, а какой совершенно непригоден для данной процедуры. Этот оставшийся код зачастую будет определять, возможно, пропущенные нами ограниченные контенты!

В ходе этого процесса можно также воспользоваться кодом для анализа зависимостей между пакетами. Код должен представлять организацию, поэтому пакеты, представляющие ограниченные контенты, в организации должны взаимодействовать точно так же, как взаимодействуют между собой настоящие подразделения организации в данной области бизнеса. Например, такой инструмент, как Structure 101, позволяет увидеть графический образ зависимостей между пакетами. Если будет замечено что-то неправильное, например что пакет товарного склада зависит от кода

в финансовом пакете, хотя в реальной организации такой зависимости нет, мы сможем понять суть проблемы и попытаться ее решить.

Этот процесс может занять целый день при небольшом объеме исходного кода или несколько недель и даже месяцев, когда придется работать с миллионами строк кода. Вам может не понадобиться сортировка всего кода по ориентированным на отдельные области пакетам перед выделением своего первого сервиса, и даже более того, может оказаться полезнее сконцентрироваться на одном месте. Эта работа не должна представлять собой стремительный процесс. Ее можно сделать пошагово, день за днем, и в нашем распоряжении имеется множество инструментов для отслеживания процесса.

Итак, мы организовали исходный код по стыкам. Что же делать дальше?

Мотивы для разбиения монолита на части

Для начала подойдет такое решение: вам хотелось бы, чтобы монолитный сервис или приложение имели меньший объем. Я бы настоятельно рекомендовал урезать эти системы. По ходу дела вы постепенно изучите микросервисы, и это поможет ограничить влияние неверных шагов на всю работу (а таких шагов вам просто не избежать!). Подумайте о нашем монолите как о куске мрамора. Мы могли бы сразу взорвать его, но это редко заканчивается хорошо. Намного разумнее обтесывать кусок постепенно.

Итак, если мы собрались разбивать монолит по кусочку, то с чего начать? Теперь у нас есть стыки, но какой из них нужно вынуть первым? Нужно подумать о том, где вы собираетесь получить наибольшую выгоду от части исходного кода, подлежащей выделению, а не просто разбивать ради самого разбиения. Рассмотрим ряд определяющих аспектов, которые помогут управлять долотом.

Темпы изменений

Возможно, мы знаем, что находимся на пороге больших изменений в способах управления запасами. Если теперь мы сделаем скол по стыку товарного склада и представим отколовшийся кусок в виде сервиса, то сможем внести изменения в этот сервис быстрее, поскольку теперь он станет автономной единицей.

Структура команды

Команда доставки MusicCorp фактически разделена между двумя географическими регионами. Одна команда находится в Лондоне, а другая на Гавайях (везет же людям!). Было бы здорово выделить код, с которым работает преимущественно гавайская команда, чтобы он перешел в ее полное владение. Эта идея рассматривается в главе 10.

Безопасность

Компания MusicCorp проверила систему безопасности и решила ужесточить меры защиты конфиденциальной информации. На данный момент все управляется ко-

дом, связанным с финансовыми операциями. Если вычленить этот сервис, то можно будет обеспечить для него дополнительные меры защиты в плане мониторинга, защиты передаваемых данных и защиты содержащихся данных. Эти меры защиты подробнее рассматриваются в главе 9.

Технология

Команда, присматривавшая за нашей системой рекомендаций, столкнулась с трудностями применения новых алгоритмов с использованием библиотеки логического программирования на языке Clojure. Ее члены посчитали, что это сможет принести пользу клиентам, повысив качество того, что мы им предлагаем. Если бы можно было вычленить код системы выдачи рекомендаций в отдельную службу, то вопрос о ее альтернативной реализации с возможностью тестирования решался бы намного проще.

Запутанные зависимости

Другой вопрос, который нужно рассмотреть, когда определены несколько стыков для разделения монолита, касается переплетения этого кода со всей остальной системой. Нам по возможности нужно выявить такой стык, у которого меньше всего зависимостей. Если есть возможность просмотреть различные стыки в виде непосредственного ациклического графа зависимостей (иногда для этого отлично подходят ранее упомянутые мною пакеты средств моделирования), это может помочь в выявлении тех стыков, которые, скорее всего, будет сложнее освободить от зависимостей.

Это подводит нас к тому, что часто служит источником запутанных зависимостей, — к базе данных.

База данных

Проблемы использования баз данных в качестве средства интеграции нескольких сервисов подробно обсуждались ранее. Как я абсолютно ясно дал понять, я не сторонник этого способа интеграции! Это означает, что нам нужно найти стыки и в базе данных, чтобы по ним можно было провести четкое разбиение. Но базы данных — весьма хитрые звери.

Решение проблем

Для начала нужно посмотреть на сам код и понять, какие его части занимаются чтением из базы данных и записью в нее. Обычно для привязки кода к базе данных и облегчения отображения объектов или структур данных на базу данных и обратно используется уровень хранилища, поддерживаемый какой-либо средой вроде Hibernate. Если до сих пор вы следовали нашим предписаниям, то у вас должен быть код, сгруппированный в пакеты, являющиеся представлениями наших ограниченных контекстов.

С кодом доступа к базам данных мы хотим сделать то же самое. Для этого может потребоваться разбиение уровня хранилища на несколько частей (рис. 5.1).

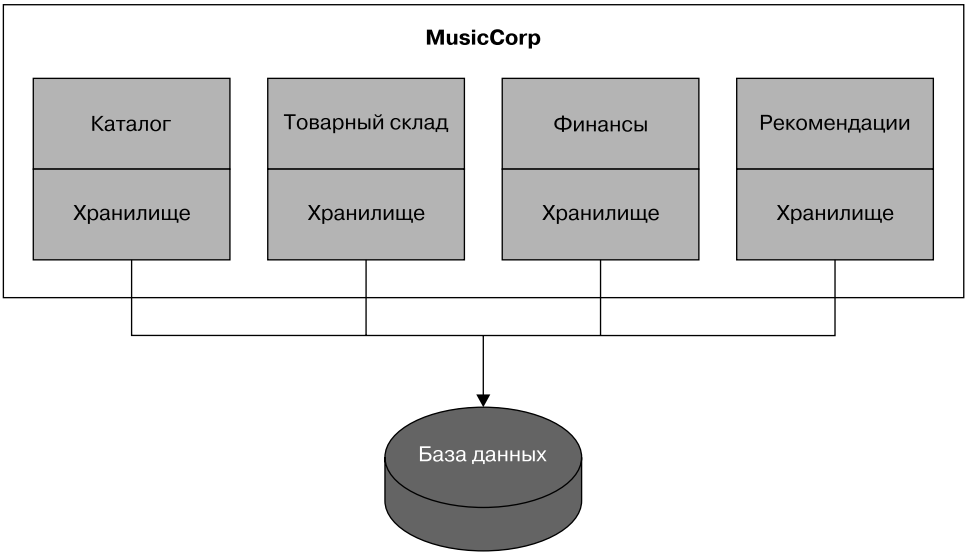


Рис. 5.1. Разбиение уровней хранилищ

Наличие кода отображения на базу данных, расположенного внутри кода для заданного контекста, может помочь разобраться в том, какие части базы данных используются тем или иным фрагментом кода. Например, среда Hibernate может прояснить ситуацию, если вы используете что-либо вроде файла отображения для каждого ограниченного контекста.

Но полной картины мы, конечно же, не получим. Например, мы можем получить возможность определения того, что код финансов использует таблицу главной бухгалтерской книги, а код каталога — таблицу товарных позиций, но при этом может быть не выяснено, что база данных использует внешний ключ, связывающий первую таблицу со второй. Чтобы на уровне базы данных увидеть такие ограничения, на которых можно споткнуться, нужно воспользоваться другим инструментальным средством визуализации данных. Для начала было бы неплохо воспользоваться таким свободно распространяемым средством, как SchemaSpy, которое может сгенерировать графическое представление взаимоотношений между таблицами.

Все это помогает разобраться в связях между таблицами, которые могут перекрывать то, что со временем станет границами сервисов. Но как разорвать эти связи? И что делать в том случае, когда одни и те же таблицы используются из нескольких ограниченных контекстов? Разобраться с подобными проблемами не так-то просто, и на эти вопросы есть масса ответов, но все же это выполнимо.

Возвращаясь к конкретным примерам, еще раз рассмотрим наш музыкальный магазин. Мы определили четыре ограниченных контекста и хотим пойти дальше и сделать на их основе четыре различных, совместно работающих сервиса. Мы со-

бираемся рассмотреть несколько конкретных примеров тех проблем, с которыми могли бы столкнуться, а также потенциальные решения этих проблем. И хотя некоторые из этих примеров относятся именно к тем сложностям, которые встречаются в ходе работы со стандартными реляционными базами данных, сходные проблемы могут возникнуть и во время работы с другими магазинами, в программных средствах которых используется язык SQL.

Пример 1: разрыв взаимоотношений, использующих внешние ключи

В этом примере код каталога использует типичная таблица товарных позиций, хранящая информацию об альбоме. А для отслеживания финансовых транзакций код финансов использует таблицу главной бухгалтерской книги. В конце каждого месяца нам нужно составлять отчеты для различных должностных лиц организации, чтобы они могли видеть состояние наших дел. Хочется сделать отчеты красивыми и легкими для чтения, поэтому вместо того, чтобы сообщить: «Мы продали 400 копий SKU 12345 и выручили на этом 1300 долларов», есть желание добавить дополнительную информацию о том, что именно было продано (то есть «Мы продали 400 копий Bruce Springsteen's Greatest Hits и выручили на этом 1300 долларов»). Чтобы добиться желаемого результата, код составления отчетов в финансовом пакете должен добраться до таблицы товарных позиций и извлечь заголовок для SKU. В нем, как показано на рис. 5.2, могут существовать ограничения, связанные с использованием внешнего ключа от таблицы главной бухгалтерской книги к таблице товарных позиций.

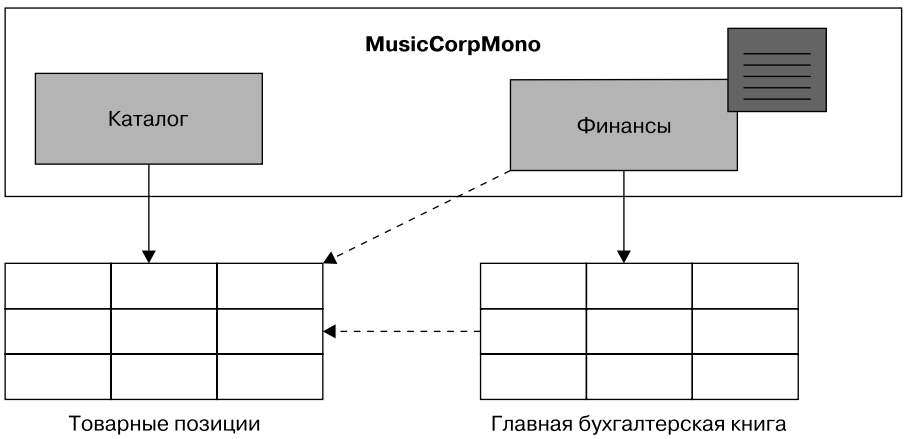


Рис. 5.2. Взаимоотношения, обусловленные наличием внешнего ключа

Итак, как же здесь можно исправить положение? Нужно внести изменения в двух местах. Следует прекратить доступ финансового кода к таблице товарных позиций, поскольку эта таблица принадлежит коду каталога, а мы не хотим, чтобы при вступивших в свои права сервисах каталога и финансов происходила интеграция

посредством базы данных. Быстрее всего решить эту проблему, заменив тот код в финансах, который обращался к таблице товарных позиций, выставлением данных через обработку в пакете каталога API-вызова, совершаемого кодом финансов. Как показано на рис. 5.3, этот API-вызов может быть предвестником того вызова, который мы сделаем по сети.

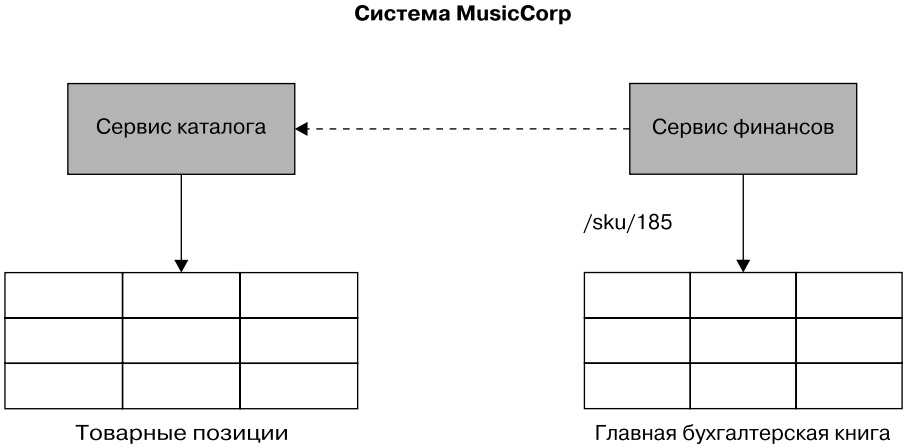


Рис. 5.3. Ситуация после отказа от использования внешних ключей

Теперь уже понятно, что для составления отчета мы можем обойтись двумя вызовами, направляемыми к базе данных. И это правильно. То же самое произойдет при наличии двух отдельных сервисов. Обычно разговор о производительности при этом не идет. На это у меня есть довольно простой ответ: насколько быстрой должна быть ваша система? И насколько быстро она работает сейчас? Если есть возможность протестировать ее текущую производительность и разобраться в том, что значит высокая производительность, тогда можно почувствовать уверенность в правильности вносимых изменений. Иногда намеренно допускается замедление работы каких-либо компонентов, чтобы взамен получить какие-то другие преимущества, особенно если такое замедление вполне приемлемо.

А как же насчет взаимоотношений, обусловленных наличием внешних ключей? Мы их просто теряем. Теперь обязанность управления вменяется получающимся у нас сервисам и снимается с уровня базы данных. Это может означать, что нам придется постоянно проверять согласованность сервисов или предпринимать иные активные действия для очистки взаимосвязанных данных. Вопрос о необходимости таких действий зачастую не относится к выбору, осуществляемому технологом. Например, если прежний сервис содержал перечень идентификаторов для элементов каталога, то что произойдет, если элемент каталога удален и теперь заказ ссылается на неверный идентификатор каталога? Должны ли мы допускать подобную ситуацию? Если да, то как это должно быть представлено в заказе при выводе его на экран? Если нет, то как мы можем проверить отсутствие нарушений? На эти вопросы должны ответить те люди, которые определяют порядок поведения системы по отношению к ее пользователям.

Пример 2: совместно используемые статичные данные

Наверное, мне попадалось столько же много кодов стран в базах данных (рис. 5.4), сколько я написал классов StringUtils для собственных Java-проектов. Это позволяет предположить, что мы планируем вносить изменения в страны, поддерживаемые нашей системой, чаще, чем будет развертываться новый код, но какой бы ни была реальная причина, в этих примерах совместного использования статичных данных, хранящихся в базах данных, придумано много нового. Итак, что же нам предпринять для музыкального магазина, если все потенциальные сервисы считывают данные из одной и той же таблицы?

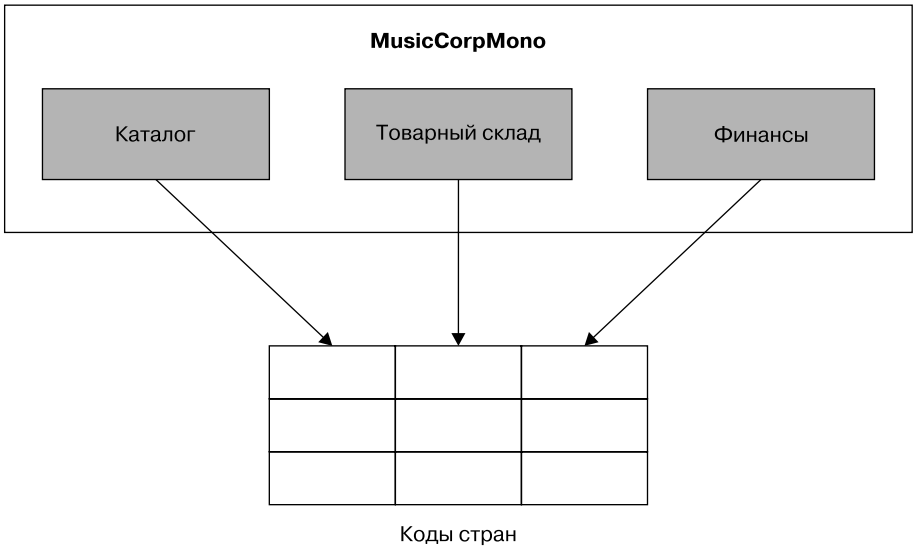


Рис. 5.4. Коды стран в базе данных

Итак, у нас есть несколько вариантов. Один из них предполагает дублирование этой таблицы для каждого из наших пакетов с тем, чтобы в долгосрочной перспективе она была продублирована также в каждом сервисе. Разумеется, это приводит к потенциальным осложнениям с согласованностью данных: что будет, если обновить одну таблицу с целью отображения создания некой страны Ньюмантопии на восточном побережье Австралии, оставив другие таблицы без изменений?

Второй вариант состоит в том, чтобы рассматривать совместно используемые статичные данные как код. Возможно, он мог бы содержаться в файле свойств, развернутом в виде части сервиса, или может использоваться в виде простого перечисления. Проблема с согласованностью данных остается, но опыт подсказывает, что намного проще поместить изменения в конфигурационные файлы, чем вносить их в действующие таблицы баз данных. Зачастую такой подход считается вполне разумным.

Третий, возможно, экстремальный вариант заключается в том, чтобы поместить статические данные в отдельный полноправный сервис. В двух ситуациях, с которыми мне приходилось сталкиваться, объема, сложности и количества правил, связанных со статическими ссылочными данными, было достаточно для того, чтобы считать такой подход оправданным, а вот когда дело касается просто кодов стран, он, вероятнее всего, будет излишним!

Лично я в большинстве ситуаций стараюсь помещать эти данные в конфигурационные файлы или непосредственно в код, поскольку чаще всего этот вариант оказывается самым простым.

Пример 3: совместное использование данных

Теперь углубимся в более сложный пример из разряда решений типичных проблем, возникающих при попытке препарировать независимые системы, — пример совместного использования изменяющихся данных. Наш финансовый код отслеживает платежи, осуществляемые клиентами за сделанные ими заказы, а также отслеживает возврат средств клиентам при возврате ими товара. Тем временем код товарного склада обновляет записи, чтобы показать отправку заказов клиентам или их возврат от клиентов. Все эти данные отображаются в одном удобном месте на сайте, позволяя клиентам наблюдать за всем происходящим с их учетной записью. Чтобы избежать усложнения, вся эта информация хранилась в универсальной таблице клиентских записей (рис. 5.5).

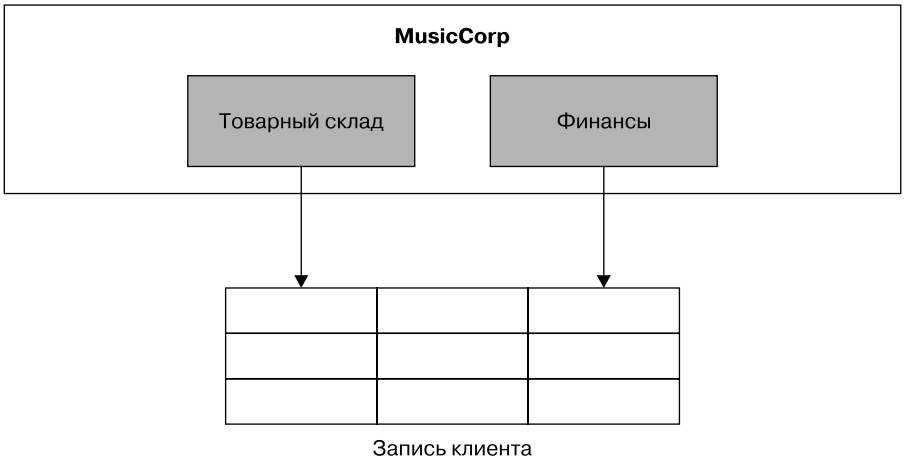


Рис. 5.5. Доступ к клиентским данным: мы ничего не упустили?

Как финансовый, так и складской код ведет запись и, возможно, время от времени осуществляет чтение из одной и той же таблицы. Как можно препарировать ее на части? Здесь мы имеем то, что вам будет попадаться довольно часто, — понятие области, не промоделированной в коде и фактически полностью смоделированной в базе данных. В этом случае пропущенным понятием области является `Customer` (Клиент).

Нам нужно превратить текущее абстрактное понятие клиента в конкретное. В качестве промежуточного этапа мы создаем новый пакет под названием `Customer`. Затем можно будет воспользоваться API для открытия кода `Customer` другим пакетам, например финансовому или складскому. Проделав все это, мы можем в итоге получить отдельный клиентский сервис (рис. 5.6).

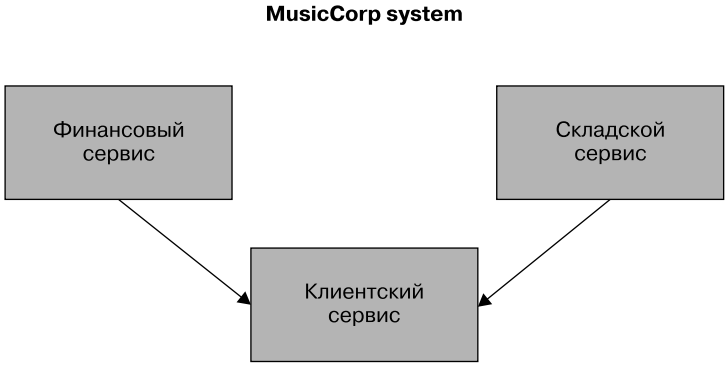


Рис. 5.6. Распознавание ограниченного контекста клиента

Пример 4: совместно используемые таблицы

На рис. 5.7 показан последний пример. Каталогу нужно сохранять название и цену продаваемых музыкальных записей, а товарному складу — вести электронный учет материально-технических ресурсов. Мы решили содержать и то и другое в одном и том же месте — в универсальной таблице товарных позиций. Раньше, когда весь код составлял единое целое, нам не было понятно, что мы фактически объединяем интересы, но теперь можно увидеть, что действительно есть два различных понятия, которые должны сохраняться по-разному.

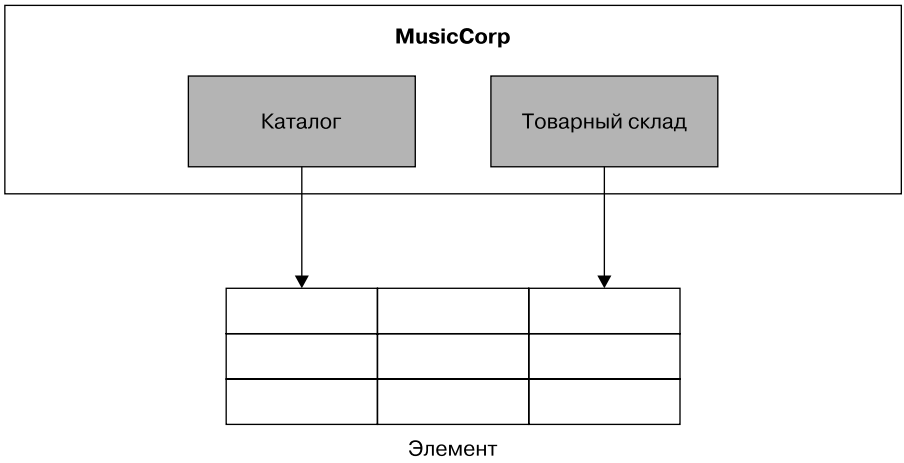


Рис. 5.7. Таблицы, совместно используемые различными контекстами

Ответ заключается в разбиении таблицы и получении двух таблиц (рис. 5.8), возможно, с созданием таблицы товарных позиций для склада и таблицы записей каталога для подробностей, необходимых сервису каталогов.

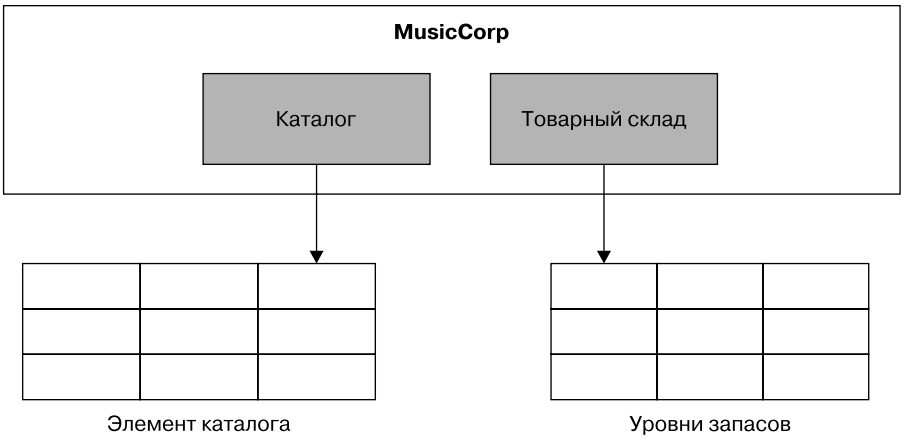


Рис. 5.8. Разбиение совместно используемой таблицы

Перестройка баз данных

То, что было рассмотрено в предыдущих примерах, относится к перестройкам баз данных, способствующих разделению ваших схем. Для более подробного изучения предмета можно обратиться к книге Скотта Дж. Амблера (Scott J. Ambler) и Прамода Садаладжа (Pramod J. Sadalage) *Refactoring Databases* (Addison-Wesley).

Поэтапное разбиение. Итак, уже найдены стыки в коде приложения, код сгруппирован вокруг ограниченных контекстов, все это использовано для нахождения стыков в базе данных и приложены все силы для ее разбиения. А что же дальше? Нужно ли выполнять радикальное разбиение, переходя от одного монолитного сервиса с единой схемой к двум сервисам, каждый из которых имеет собственную схему? Я настоятельно рекомендую разбить схему, но не разделять сервис до разбиения кода приложения на два отдельных микросервиса (рис. 5.9).

При отдельной схеме число потенциальных вызовов для выполнения одного действия будет потенциально увеличиваться. Там, где прежде можно было получать все нужные данные при выполнении одной инструкции `SELECT`, теперь придется извлекать данные из двух мест и объединять их в памяти. Кроме того, в результате перехода к двум схемам получается нарушение целостности транзакции, которое может существенно повлиять на наше приложение, о чем мы поговорим в следующем разделе. При разбиении схемы и неразбитом коде приложения мы оставляем для себя возможность вернуться к прежней схеме или продолжить настройки, никак не влияя на потребителей сервиса. Как только мы удостоверимся в том, что разделение базы данных имеет смысл, можно будет подумать и о разбиении кода приложения на два сервиса.

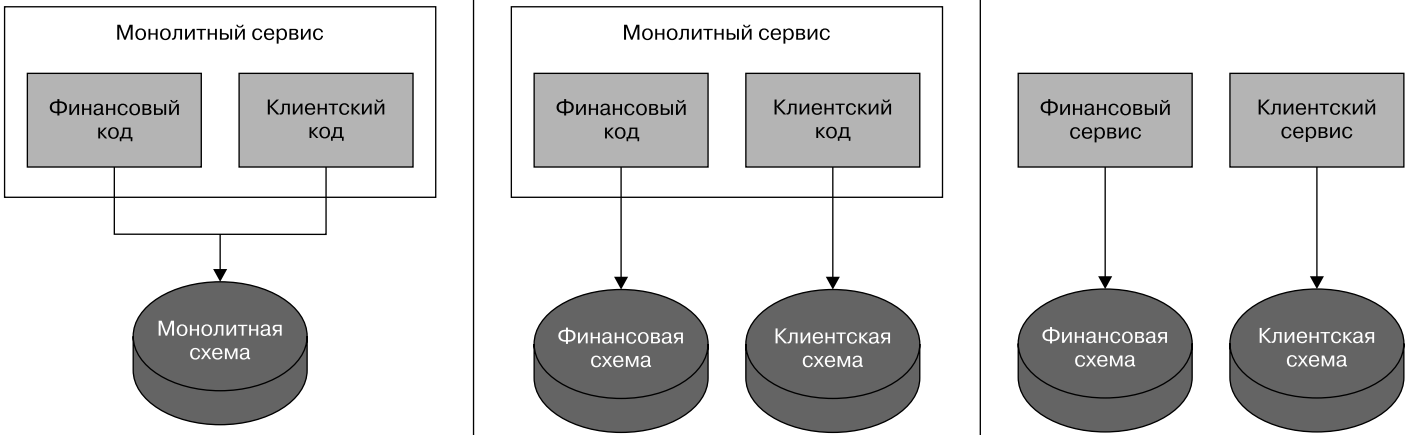


Рис. 5.9. Поэтапное разбиение сервиса

Транзакционные границы

Транзакции — вещь полезная. Они позволяют быть уверенным в том, что либо данные события произойдут вместе, либо не случится ни одно из них. Особую пользу они приобретают при вставке данных в базу, давая возможность одновременно обновлять сразу несколько таблиц и знать при этом, что в случае сбоя произойдет полный откат к прежнему состоянию, что исключит ситуацию, при которой данные будут находиться в несогласованном состоянии. Проще говоря, транзакции позволяют группировать вместе несколько различных действий, которые переносят нашу систему из одного согласованного состояния в другое, при этом либо все срабатывает, либо ничего не изменяется.

Транзакции применимы не только к базам данных, хотя наиболее часто они используются именно в их контексте. Например, брокеры сообщений уже давно позволяют вам публиковать и получать сообщения также внутри транзакций.

При монолитной схеме все операции по созданию или изменению, скорее всего, будут проводиться в рамках единой транзакционной границы. Когда мы разбиваем на части наши базы данных, то утрачиваем ту безопасность, которая обеспечивается при наличии единой транзакции. Рассмотрим простой пример в контексте MusicCorp. При создании заказа мне нужно обновить таблицу заказов, утвердив тем самым создание клиентского заказа, а также поместить запись в таблицу для команды товарного склада, чтобы оповестить ее о существовании заказа, который нужно скомплектовать для отгрузки. Мы добрались до распределения кода приложения на отдельные пакеты, в достаточной степени разделили клиентскую и складскую части схемы и приготовились поместить эти части в их собственные схемы, предваряя тем самым разделение кода приложения.

В имеющейся у нас монолитной схеме создание заказа и вставка записи для складской команды производились в рамках одной транзакции (рис. 5.10).

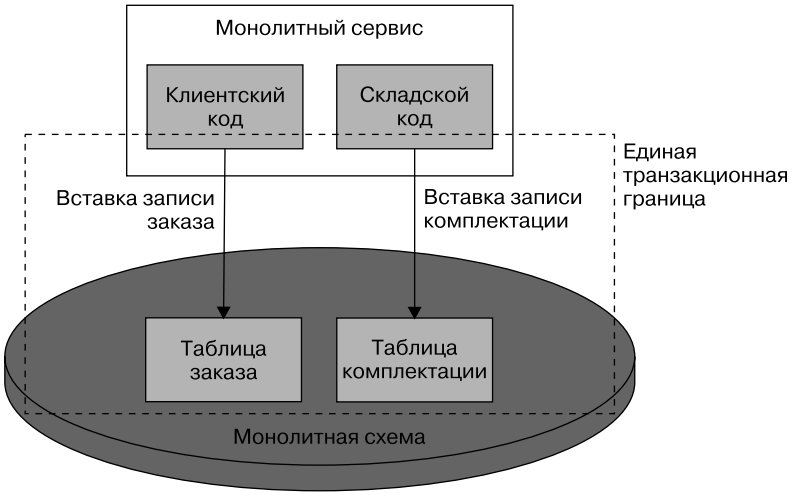


Рис. 5.10. Обновление двух таблиц в рамках одной транзакции

Но если мы разбили схему на две отдельные схемы: одну для данных, связанных с клиентом, а другую для склада, — мы утратили транзакционную безопасность. Процесс размещения заказа теперь охватывает две обособленные транзакционные границы (рис. 5.11). Если при вставке в таблицу заказов произойдет сбой, то мы, конечно же, можем все остановить, сохраняя согласованное состояние. Но что получится, если вставка в таблицу заказов пройдет успешно, а при вставке в таблицу комплектации произойдет сбой?

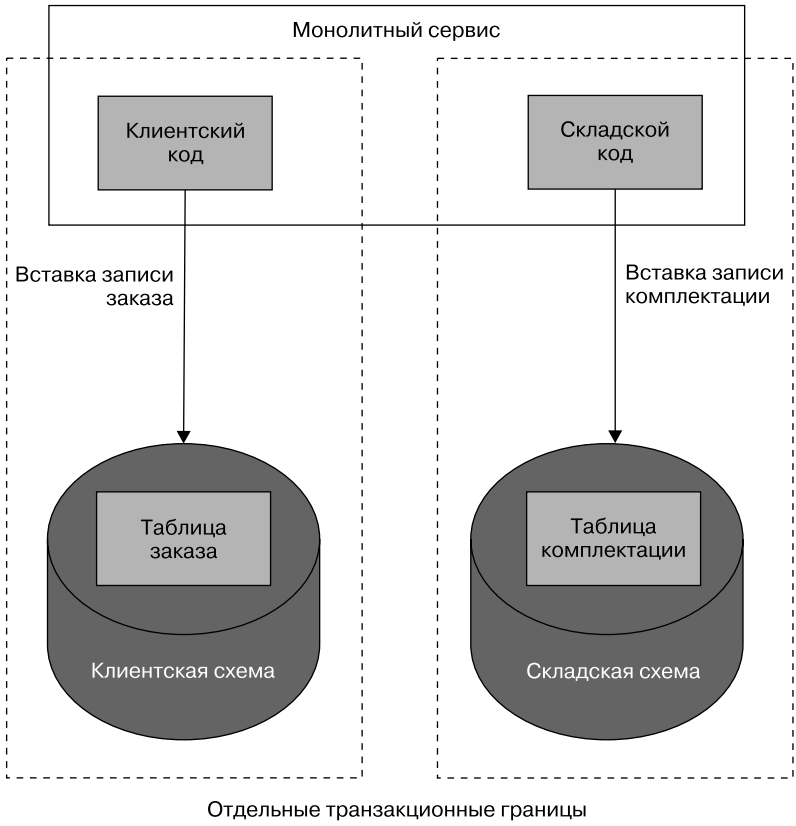


Рис. 5.11. Распространение границ транзакций для единой операции

Повторная попытка

Самого факта получения и размещения заказа может быть для нас достаточно, и позднее мы можем принять решение о повторной вставке записи комплектации в складскую таблицу. Эту часть операции можно будет поставить в очередь или занести в файл журнала и повторить попытку чуть позже. Для некоторых видов операций в этом есть определенный смысл, но мы должны гарантировать, что повторная попытка исправит ситуацию.

Во многих смыслах это еще одна форма того, что называется *возможной согласованностью*. Вместо использования транзакционной границы как гарантии согласованного состояния по окончании транзакции мы допускаем, что система сама приведет себя в согласованное состояние в какой-то будущий момент времени. Такой подход особенно хорош для продолжительных бизнес-операций. Более подробно он будет рассмотрен в главе 11 при изучении особенностей масштабируемых шаблонов.

Отмена всей операции

Еще один вариант заключается в отмене всей операции. В этом случае систему нужно вернуть в прежнее согласованное состояние. С таблицей комплектации все просто, поскольку вставка дала сбой, но в таблице заказов мы имеем уже зафиксированную транзакцию. Поэтому нужно сделать откат. Необходимое действие выполняется в рамках *компенсационной транзакции*, то есть запуска новой транзакции для отката всего, что только что случилось. В нашем случае все может свестись к простой выдаче инструкции удаления `DELETE`, предназначенной для удаления заказа из базы данных. Затем нужно будет отчитаться в пользовательском интерфейсе о сбое операции. В монолитной системе наше приложение может справиться с обоими аспектами, а вот когда код приложения уже разбит на части, нужно призадуматься о том, что делать. Где именно должна находиться логика управления компенсационной транзакцией, в клиентском сервисе или где-то еще?

А как быть, если произойдет сбой компенсационной транзакции? Вероятность этого не исключена. Тогда у нас в таблице заказов будет заказ, не имеющий соответствующей ему инструкции по комплектации. В такой ситуации нужно либо провести компенсационную транзакцию повторно, либо позволить какому-нибудь внутреннему процессу убрать несогласованность чуть позже. Можно было бы просто воспользоваться экраном обслуживания с доступом только со стороны административного персонала или же использовать автоматизированный процесс.

А теперь подумайте о том, что будет, если у нас не одна или две операции, согласованности которых нужно придерживаться, а три, четыре или пять операций. Проведение компенсационных транзакций для каждого сбойного режима очень трудно не то что реализовать, но даже осмыслить.

Распределенные транзакции

Альтернативой ручной организации компенсационных транзакций является использование *распределенной транзакции*. Распределенные транзакции пытаются объединить в себе сразу несколько транзакций, используя для управления различными транзакциями, проводимыми в базовых системах, общий управляющий процесс, называемый диспетчером транзакций. Точно так же, как и обычная транзакция, распределенная транзакция старается гарантировать пребывание всего в согласованном состоянии, только она пытается сделать это в рамках нескольких систем, запущенных в различных процессах, связь между которыми зачастую осуществляется через сетевые границы.

Наиболее распространенный алгоритм управления распределенными транзакциями — особенно теми, которые носят кратковременный характер, как в случае с нашим клиентским заказом, — заключается в использовании двухфазной фиксации. При этом сначала следует фаза голосования, при которой каждый участник (также называемый в данном контексте партнером) распределенной транзакции сообщает диспетчеру транзакций о том, считает ли он, что его локальная транзакция может начинаться. Если диспетчер транзакций получит положительный ответ от всех участников, он дает им команду на начало транзакций и выполняет их фиксацию. Для того чтобы совершить откат всех транзакций, диспетчеру транзакций хватает единственного отрицательного ответа.

Такой подход предполагает, что все участники останавливаются, пока центральный координационный процесс не даст команду на продолжение работы. Это означает, что мы не застрахованы от остановки работы. Если диспетчер транзакций зависнет, отложенные транзакции никогда не завершатся. Если партнер не ответит в процессе голосования, все будет заблокировано. И неизвестно, что произойдет, если фиксация даст сбой после голосования. В этом алгоритме есть безусловное предположение о том, что такого никогда не случится: если партнер сказал «да» при голосовании, значит, мы должны предполагать, что его транзакция будет зафиксирована. Партнерам нужен способ, позволяющий заставить фиксацию происходить в нужный момент. Это означает, что данный алгоритм не защищен от сторонних сбоев, вернее, он предусматривает попытку обнаружения большинства случаев сбоев.

Этот координационный процесс предусматривает также установку блокировок, то есть отложенная транзакция должна удерживать блокировку ресурсов. Блокировка ресурсов может привести к конкуренции, существенно усложняя масштабируемые системы, особенно в контексте распределенных систем.

Распределенные транзакции были реализованы для конкретных технологических стеков, таких как Transaction API в Java, что позволяет таким разрозненным ресурсам, как база данных и очередь сообщений, участвовать в одной и той же всеобъемлющей транзакции. Разобраться в различных алгоритмах довольно трудно, поэтому я советую отказаться от попытки создания собственных алгоритмов. Если вы считаете, что нужно пойти именно этим путем, лучше досконально исследуйте данную тему и посмотрите, можно ли воспользоваться какой-либо из уже имеющихся реализаций.

Так что же делать?

Все эти решения усложняют систему. Как видите, разобраться в распределенных транзакциях довольно трудно и фактически они могут воспрепятствовать масштабированию. О системах, которые в конечном итоге сводятся к компенсационной логике повторов, труднее рассуждать, и для устранения несогласованности данных они могут нуждаться в ином компенсационном поведении.

Когда вам встречаются бизнес-операции, проводимые в данный момент в рамках единой транзакции, задайте себе вопрос, действительно ли они должны это делать. Не могут ли они проводиться в различных локальных транзакциях и полагаться

на концепцию возможной согласованности? Создавать такие системы и заниматься их масштабированием намного проще (более подробно этот вопрос рассматривается в главе 11).

Если попадется такое состояние, необходимость в согласованности которого не вызывает никаких сомнений, то в первую очередь сделайте все возможное, чтобы избежать разбиения. Приложите для этого все усилия. Если же разбиения будет не избежать, подумайте об изменении чисто технического взгляда на процесс (например, транзакции в базе данных) и создайте конкретные понятия, представляющие саму транзакцию. Это даст вам возможность зацепиться за запуск других операций, подобных компенсационным транзакциям, а также за способ отслеживания этих более сложных понятий в вашей системе и управления ими. Например, можно прийти к идее незавершенного заказа, которая даст вам реальное место для концентрации всей логики вокруг сквозной обработки заказа (и работы с исключениями).

Создание отчетов

Как мы уже видели, при разбиении сервиса на более мелкие части нужно также в потенциале разбить на части и способы хранения этих данных. Но это создает проблему, когда дело доходит до жизненно важного и весьма распространенного случая — создания отчетов.

Такие фундаментальные изменения в архитектуре, как переход к микросервисам, вызовет множество разрушений, но это не означает, что нужно отказываться от всего, что мы делаем. Аудиторию наших систем отчетности, как и любых других систем, составляют пользователи, и мы должны учитывать их запросы. Фундаментальная перестройка архитектуры была бы преувеличением наших возможностей, поэтому ее нужно просто приспособить под новые нужды. Я, конечно, не берусь утверждать, что пространство создания отчетов не должно подвергаться разрушению, — это неизбежно, и тут важно сначала определить порядок работы с существующими процессами. Иногда нам придется выбирать пути борьбы.

База данных для создания отчетов

Создание отчетов обычно требует группировать данные, поступающие из нескольких подразделений организации, с целью генерации полезных выходных данных. Например, нам нужно расширить данные из главной бухгалтерской книги описанием того, что было продано, и взять это описание из каталога. Или же отследить интересы, которые проявляют при покупках конкретные особо ценные покупатели, которым может потребоваться информация из истории их покупок и клиентского профиля.

При стандартной монолитной архитектуре сервиса все данные хранятся в одной большой базе данных. Это означает, что все они находятся в одном месте, поэтому создание отчетов по всей информации выполняется довольно легко и мы можем просто объединить данные в SQL-запросах или чем-то подобном. Обычно создание отчетов не запускается на основной базе данных из опасения того, что нагрузка на

них, создаваемая запросами, повлияет на производительность основной системы, поэтому зачастую системы создания отчетов привязывают к копии базы данных, предназначенной для чтения (рис. 5.12).

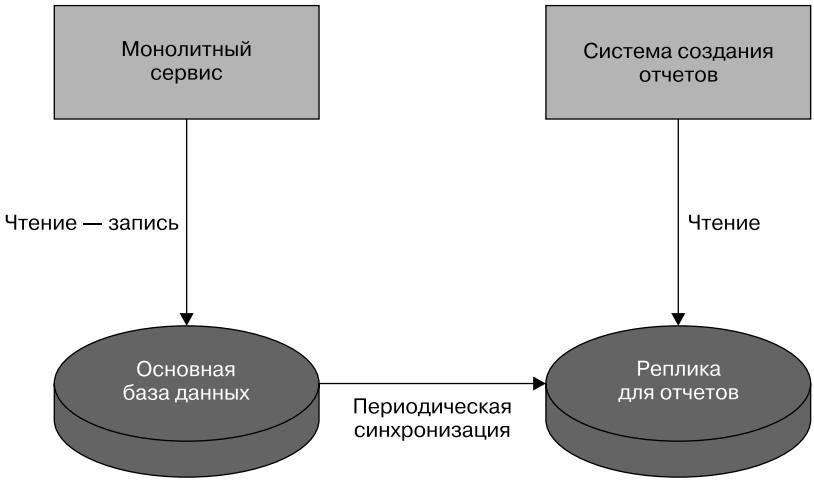


Рис. 5.12. Стандартная копия для чтения данных

При таком подходе мы получаем весьма большое преимущество, заключающееся в том, что все данные уже находятся в одном месте. Это позволяет воспользоваться весьма простым инструментарием для их запроса. Но есть также пара недостатков. Во-первых, схема базы данных теперь фактически представляет собой API, совместно используемый работающими монолитными сервисами и любой системой создания отчетов. Поэтому изменение, вносимое в схему, должно быть тщательно отрегулировано. В действительности мы получаем еще одно препятствие, уменьшающее шансы любого желающего взяться за решение задачи внесения такого изменения и его согласования.

Во-вторых, у нас весьма ограничен выбор вариантов того, как может быть оптимизирована база данных для обоих случаев ее использования — поддержки основной системы и системы создания отчетов. Некоторые базы данных позволяют проводить оптимизацию копий, предназначенных для чтения, чтобы ускорить создание отчетов и повысить его эффективность. Например, MySQL позволит запускать различные виды внутренней обработки, не создающие издержек при управлении транзакциями. Но мы не можем структурировать данные по-разному с целью ускорения создания отчетов, если сделанные для этого изменения в структуре данных плохо повлияют на рабочую систему. Часто случается, что схема хороша для одного сценария использования и нехороша для другого или же она становится наименьшим общим знаменателем, не обеспечивающим наилучший вариант ни для одной из целей ее использования.

И наконец, доступные нам варианты баз данных недавно уже были отвергнуты. Поскольку стандартные реляционные базы данных выставляют на всеобщее обозрение интерфейсы SQL-запросов, которые работают со многими инструментами

создания отчетов, они не всегда являются наилучшим вариантом хранения данных для работающих у нас сервисов.

Что, если данные нашего приложения лучше моделируются как граф в Neo4j? Или если нам лучше будет воспользоваться таким хранилищем документов, как MongoDB? А что, если для нашей системы создания отчетов захочется присмотреться к использованию основанной на понятии столбцов базе данных Cassandra, которая упрощает масштабирование более существенных объемов данных? Ограничивая себя необходимостью использования одной базы данных в обеих целях, мы часто лишаемся возможности подобного выбора и исследования новых вариантов.

Итак, пусть это далеко от совершенства, но оно работает (чаще всего). А что нам делать, если информация хранится в нескольких различных системах? Есть ли способ собрать все данные вместе для запуска системы создания отчетов? И можем ли мы потенциально отыскать способ избавления от ряда недостатков, присущих стандартной модели создания отчетов на основе использования базы данных?

Оказывается, в нашем распоряжении есть сразу несколько весьма жизнеспособных альтернатив этому подходу. Какое из решений будет наиболее подходящим именно для вас, зависит от ряда факторов, но мы изучим несколько вариантов, которые мне встречались на практике.

Извлечение данных посредством служебных вызовов

Существует множество вариантов этой модели, но все они основаны на извлечении запрошенных данных из исходной системы посредством API-вызовов. Для очень простой системы создания отчетов, подобной панели мониторинга, на которой может быть нужно всего лишь показывать количество заказов, размещенных за последние 15 минут, это может оказаться вполне подходящим вариантом. Чтобы создать отчет на основе данных от двух и более систем, нужно для сбора этих данных сделать несколько вызовов.

Но при вариантах, требующих более крупных объемов данных, этот подход быстро становится непригодным. Представьте такой вариант использования, при котором нужно составить отчет о предпочтениях в покупках клиента нашего музыкального магазина за последние 24 месяца с рассмотрением различных тенденций в его поведении и того, как это влияло на выручку. Для этого необходимо извлечь большие объемы данных как минимум из клиентской и финансовой систем. Сохранять локальную копию этих данных в системе создания отчетов опасно, поскольку мы будем не в курсе происходящих изменений (задним числом могут изменяться даже исторические данные), поэтому для создания точного отчета нужны все финансовые и клиентские записи за последние два года. Даже при скромном числе клиентов можно будет заметить, что вскоре эта операция станет проводиться очень медленно.

Системы создания отчетов часто зависят от программных инструментов сторонних производителей, ожидающих получения данных вполне определенным

способом, и предоставление SQL-интерфейса в данном случае является самым быстрым путем обеспечения наиболее простой интеграции с ними вашей цепочки средств создания отчетов. Разумеется, мы могли бы воспользоваться этим подходом для периодического получения данных в базе данных SQL, но это все же будет создавать для нас ряд проблем.

Одной из ключевых проблем является то, что API, доступные в различных микросервисах, могут быть не предназначены для использования их в сценариях создания отчетов. Например, сервис клиентов может позволить найти клиента по его идентификатору или же отыскать его по тем или иным полям, но не факт, что он раскроет API для извлечения данных обо всех клиентах. Это может привести к выдаче множества вызовов для извлечения всех данных. Например, чтобы последовательно перебрать список всех клиентов, придется делать отдельный вызов для каждого клиента. Это не только может сделать неэффективной работу системы создания отчетов, но и создаст большую нагрузку на задействованный сервис.

Можно, конечно, ускорить извлечение данных путем добавления к ресурсам, раскрываемым нашим сервисом, заголовков кэша и кэшированием этих данных где-нибудь вроде прокси-сервера, который будет выглядеть для клиента как исходный сервер, но особенность создания отчетов предполагает получение доступа к длинной веренице данных. Это означает, что мы можем запросить такие ресурсы, которые прежде не запрашивались никогда (или по крайней мере довольно долго), а это приведет к потенциально дорогостоящему непопаданию в кэш.

Для упрощения создания отчетов данную проблему можно решить путем раскрытия пакетных API. Например, наш сервис клиентов может позволить прохождение по списку идентификаторов пользователей для извлечения данных о них в пакетах или даже раскрыть интерфейс, позволяющий пролистывать данные обо всех клиентах. Более экстремальным вариантом этого действия будет моделирование пакетного запроса в качестве самостоятельного ресурса. Например, клиентский сервис может раскрыть что-то вроде конечной точки ресурса `BatchCustomerExport`. Вызывающая система будет выдавать POST-запрос `BatchRequest`, возможно, передавая сведения о месте, где может находиться файл со всеми данными. Клиентский сервис в качестве ответа вернет код HTTP 202, показывающий, что запрос был принят, но еще не обработан. Затем вызывающая система может опрашивать ресурс, ожидая возвращения от него статуса выполнения 201 Created, свидетельствующего о выполнении запроса и о том, что вызывающая система может перейти к извлечению данных. Это может позволить потенциально большим файлам данных экспортироваться без издержек, связанных с их отправкой по HTTP. Вместо этого система может просто сохранить CSV-файл в место совместного доступа.

Мне приходилось видеть предыдущий подход, используемый для пакетной вставки данных, с чем он вполне справлялся. Но он представляется мне менее подходящим для системы создания отчетов, поскольку я чувствую, что есть и другие, потенциально более простые решения, поддающиеся более эффективному масштабированию, чем при работе с традиционными отчетными потребностями.

Программы перекачки данных

Вместо того чтобы заставлять системы создания отчетов извлекать данные, можно перемещать данные в эти системы. Одним из недостатков извлечения данных посредством стандартных HTTP-вызовов являются издержки HTTP при отправке большого количества вызовов наряду с издержками, связанными с необходимостью создания API, которые могут пригодиться только для создания отчетов. Альтернативным вариантом может послужить использование отдельной программы, имеющей прямой доступ к базе данных сервиса, являющегося источником данных, и перекачивающей данные в отчетную базу данных (рис. 5.13).

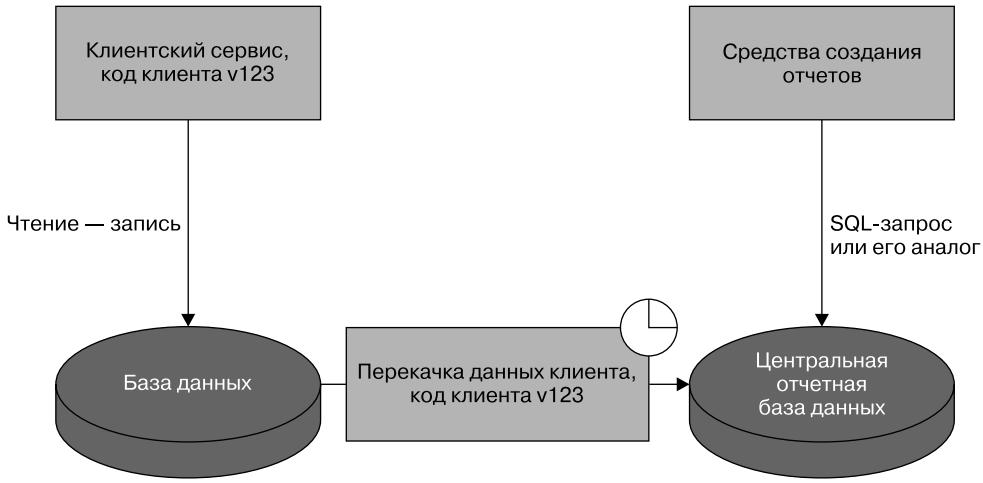


Рис. 5.13. Использование программы перекачки данных для их периодического перемещения в центральную отчетную базу данных

И вот здесь вы можете возразить: «Сэм, ты же говорил, что лучше не использовать программы, интегрированные посредством базы данных!» По крайней мере, я надеюсь на такое ваше замечание, учитывая настойчивость, с которой я высказывал возражения по данному вопросу! Этот подход при надлежащей реализации является весьма интересным исключением, в котором недостатки связанности более чем компенсированы легкостью создания отчетов.

Для начала программы перекачки данных должны создаваться и управляться той же командой, которая управляет сервисом. Она должна быть такой же простой, как программа командной строки, запускаемая с помощью Cron. Эта программа должна довольно много знать и о внутренней базе данных для сервиса, и о схеме для создания отчетов. Задача перекачки данных заключается в отображении одного на другое. Мы постараемся сократить проблемы со связанностью за счет того, что перекачкой и сервисом будет управлять одна и та же команда. Фактически я хочу предложить, чтобы управление версиями этих средств велось совместно и чтобы сборки программы перекачки данных создавались в виде дополнительного побочного продукта и являлись частью сборки самого сервиса в предположении,

что при развертывании одной из этих сборок происходит развертывание и другой сборки. Поскольку об их совместном развертывании и о недопустимости открытия доступа к схеме где-либо за пределами команды сервиса мы сделали явное заявление, многие из традиционных проблем интеграции на основе использования баз данных значительно смягчаются.

Связанность в самой схеме создания отчетов остается, но мы должны считать ее публикуемым API, который трудно изменить. Некоторые базы данных предоставляют технологии, позволяющие и дальше снижать цену за использование такого подхода. На рис. 5.14 показан пример, касающийся реляционных баз данных, где можно иметь одну схему в отчетной базе данных для каждого сервиса, используя нечто вроде материализованных представлений для создания агрегированного представления. В этом случае для программы перекачки данных клиента раскрывается только схема создания отчетов для клиентских данных. Но будет ли это чем-нибудь, что можно сделать в произвольном стиле, зависит от возможностей выбранной для отчетов базы данных.

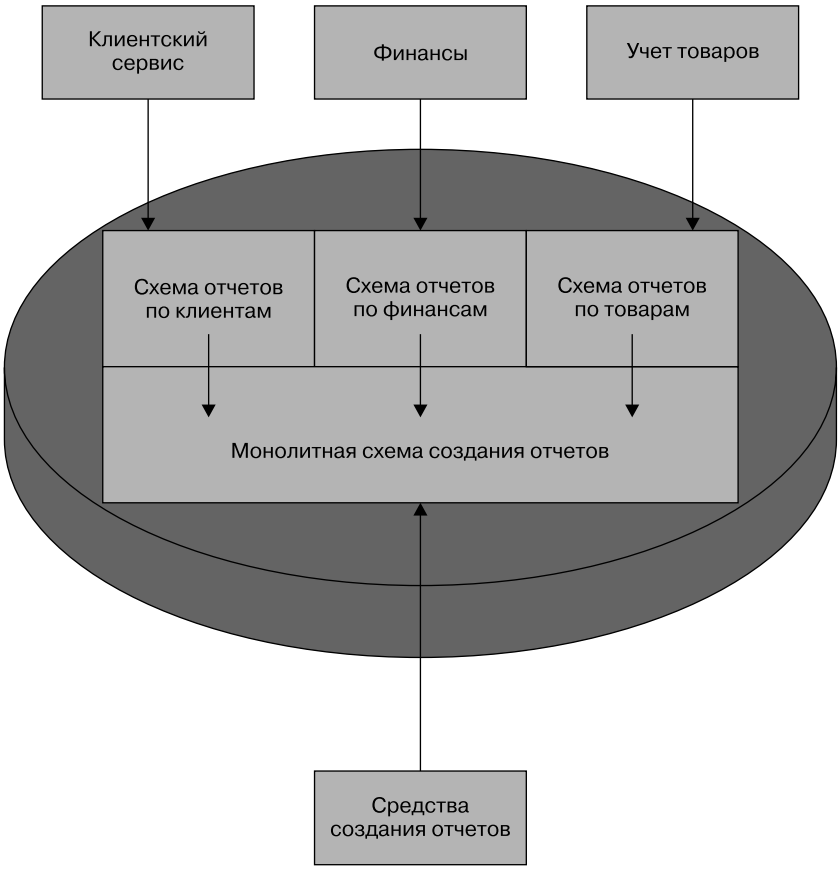


Рис. 5.14. Использование материализованных представлений для создания единой монолитной схемы создания отчетов

Разумеется, здесь сложность интеграции упрятана глубже в схему и будет зависеть от возможностей базы данных добиться от такой структуры высокой производительности. Хотя в целом я считаю программы перекачки данных весьма разумным и работоспособным предложением, у меня остаются сомнения в том, что сложность сегментированной схемы сможет себя оправдать, особенно если учитывать проблемы в управлении изменениями в базе данных.

Альтернативные направления

В одном из проектов с моим участием мы использовали серии программ перекачки данных для заполнения JSON-файлов в AWS S3, эффективно применяя S3 для маскировки огромной ярмарки данных! Этот подход работал весьма исправно до тех пор, пока нам не потребовалось расширить данное решение, и на момент написания книги мы искали, чем заменить программы перекачки, чтобы вместо этого заполнить куб, который мог быть интегрирован со стандартными средствами создания отчетов, такими как Excel и Tableau.

Перекачка данных на основе событий

В главе 4 упоминалась идея микросервисов, выдающих события на основе изменения состояния тех объектов, которыми они управляли. Например, наш клиентский сервис может выдавать событие при создании, или обновлении, или удалении клиента. Для микросервисов, выставляющих напоказ выдачу событий, имеется вариант написания своего подписчика на события, который перекачивает данные в отчетную базу данных (рис. 5.15).

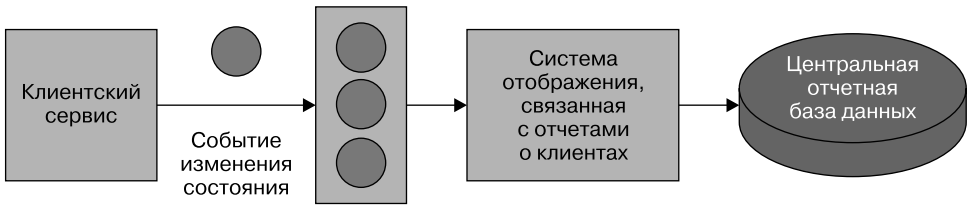


Рис. 5.15. Программа перекачки данных на основе событий, использующая события изменения состояния для наполнения отчетной базы данных

Связанности в используемой базе данных исходного микросервиса теперь удалось избежать. Взамен мы просто привязались к событиям, выдаваемым сервисом, который разработан открытым для внешних потребителей. При условии, что у этих событий временный характер, мы получаем также более простую возможность проявить интеллект в том, какие данные отправлять центральному хранилищу отчетности. Мы можем отправить данные системе создания отчетов, как только увидим событие, позволяя данным быстрее перетекать в отчетную систему, и не быть зависимыми от регулярного графика, как при обычной перекачке данных.

Кроме того, если сохранять информацию о том, какие события уже были обработаны, мы сможем просто обработать новое событие сразу же по его поступлении,

предполагая при этом, что старые события уже были отображены на систему создания отчетов. Это означает, что внедрение будет более эффективным, поскольку нам нужно лишь отправить различия. Аналогичные действия можно выполнить и с программой перекачки данных, но управлять этим придется самостоятельно, принимая во внимание абсолютно временный характер потока событий (*x* случается с меткой времени *y*), что существенно нам поможет.

Поскольку программа перекачки данных на основе событий имеет меньшую связанность с внутренними механизмами сервиса, будет также проще рассматривать вопрос управления всем этим отдельной группой той команды, которая присматривает за самим микросервисом. Поскольку по своей природе поток событий не слишком связывает подписчиков в их возможностях внесения изменений в сервис, эта система отображения событий может развиваться независимо от сервиса, который на нее подписан.

Основной недостаток такого подхода состоит в том, что вся необходимая информация должна передаваться в виде событий и может масштабироваться под большие объемы данных не так широко, как при использовании программы перекачки данных, у которой есть преимущество работы непосредственно на уровне базы данных. Тем не менее более слабая связанность и более свежие данные, доступные благодаря этому подходу, делают его весьма привлекательным для рассмотрения, если соответствующие события выставляются на всеобщее обозрение.

Перекачка данных на основе систем резервного копирования

Этот вариант основан на подходе, используемом в Netflix, в котором применяются существующие решения по созданию резервных копий, а также решаются некоторые проблемы масштабирования, с которыми приходится сталкиваться компании. Отчасти его можно рассматривать в качестве особого средства перекачки данных, но, похоже, такое интересное решение вполне заслуживает включения в наш арсенал.

Компания Netflix решила использовать базу данных Cassandra в качестве стандартного резервного хранилища своих многочисленных сервисов. Netflix потратила немало времени на создание средств, облегчающих работу с Cassandra, многими из которых компания делится с остальным миром посредством ряда проектов с открытым кодом. Конечно же, необходимость резервного копирования тех данных, которые хранятся в Netflix, вполне очевидна. Для резервного копирования данных, хранящихся в базе данных Cassandra, стандартным подходом является создание копий поддерживающих ее файлов и сохранение их в безопасном месте. Netflix сохраняет эти файлы, известные как SSTables, в принадлежащем компании Amazon хранилище объектов S3, твердо гарантирущем долговечность хранения данных.

Netflix нуждается в отчетах по всем этим данным, но с учетом задействованных масштабов решить эту задачу нелегко. В выбранном компанией подходе применяется среда Hadoop, которая использует резервные копии SSTable в качестве источника для своих заданий. Напоследок компания Netflix завершила реализацию конвейера, способного с использованием рассматриваемого подхода обрабатывать

большой объем данных, которые она затем превратила в проект с открытым кодом под названием Aegisthus. Но как и при использовании программ перекачки данных, при применении этой модели мы по-прежнему сталкиваемся с наличием связанности с целевой схемой составления отчетов (или с целевой системой).

Возможно, применение аналогичного подхода, то есть систем отображения, создающих резервные копии, позволит выработать вполне работоспособное решение и в других контекстах. А если вы уже используете базу данных Cassandra, то компания Netflix сделала основную часть работы за вас!

Переход к реальности

Многие из ранее выделенных моделей представляют собой различные способы получения большого объема данных из множества различных мест и помещения их в одно место. Но неужели идея создания всех отчетов из одного места по-прежнему имеет право на существование? У нас имеются отчетные данные, выводимые на панель управления, разного рода предупреждения, финансовые отчеты, аналитика, связанная с пользовательской активностью, — и все это предъявляет различные требования к точности и своевременности, что может найти выражение в выборе различных технических приемов для их получения. В соответствии с уточнениями, которые даются в главе 8, мы перемещаем все больше и больше данных по направлению к универсальным системам обработки событий, способным направлять данные в несколько разных мест в зависимости от наших потребностей.

Цена внесения изменений

В книге приводится немало причин, по которым я поддерживаю необходимость внесения незначительных, постепенных изменений, но одним из основных стимулов является понимание влияния каждого вносимого изменения и корректировка их направления, если она потребуется. Это позволяет более эффективно снижать цену ошибок, но не может полностью исключить вероятность их совершения. Мы можем и будем делать ошибки, и нужно принимать это как должное. Но, кроме этого, мы должны понимать, как наилучшим образом можно уменьшить цену этих ошибок.

Как мы уже поняли, цена перемещения кода в его исходном источнике относительно невелика. В нашем распоряжении имеется множество вспомогательных средств, и если возникнет проблема, ее, как правило, можно быстро устранить. Но разбиение на части базы данных требует намного большего объема работы, а откат изменений, вносимых в базу данных, является довольно-таки непростой задачей. Точно так же весьма нелегким делом может оказаться распутывание излишней связующей интеграции сервисов или необходимость полного переписывания API, используемого несколькими потребителями. Высокая цена изменений означает, что эти операции имеют все более высокую степень риска. Как можно управлять степенью риска? Мой подход заключается в допущении тех ошибок, отрицательное воздействие которых будет наименьшим.

Я стараюсь все рассматривать в том месте, где цена изменений и цена ошибок будут наименьшими: на доске в лекционной аудитории. Изобразите краткое представление предлагаемой конструкции. Посмотрите, что получится, когда вы запускаете варианты использования через предполагаемые границы сервиса. Например, можно представить, каковы будут для музыкального магазина последствия того, что клиент ищет запись, регистрируется на сайте или приобретает альбом. Какие для этого делаются вызовы? Замечаете ли вы случайные циклические ссылки? Замечаете ли вы два сервиса, ведущих между собой слишком интенсивный обмен данными, который может быть признаком того, что они должны составлять единое целое?

Здесь неплохо было бы внедрить подход, который более типичен при обучении созданию объектно-ориентированных систем: применение карт событийного взаимодействия классов (CRC). При использовании CRC-карт создается одна индексная карта с именем класса, на которой указывается, за что он отвечает и с чем взаимодействует. При проработке предложенной конструкции для каждого сервиса перечисляется все, за что он отвечает в понятиях предоставляемых им возможностей, также на схеме указываются совместно работающие с ним сервисы. По мере проработки все большего количества вариантов использования вы начинаете понимать, насколько правильно все это сообразуется друг с другом.

Умение разбираться в основных причинах

Мы рассмотрели способы разбиения крупных сервисов на более мелкие, но в чем первопричина того, что сервисы разрослись до таких больших размеров? Сначала нужно понять, что разрастание сервиса до определенного объема, требующего его разбиения, — это вполне нормальное явление. Нам нужно, чтобы архитектура системы со временем изменялась. Главное — разобраться в том, что она требует разбиения еще до того, как такое разбиение станет обходиться слишком дорого.

Однако на практике многие из нас видели, как сервисы разрастаются, приобретая размеры, абсолютно не отвечающие здравому смыслу. Несмотря на то что нам известно, что с меньшим набором сервисов проще работать, чем с тем огромным чудовищем, которое у нас получилось, мы по-прежнему занимаемся выращиванием чудовища. Почему?

Часть проблемы заключается в том, чтобы знать, с чего начать, и я надеюсь, что эта глава помогла вам в этом разобраться. Но другой проблемой являются затраты, связанные с разбиением сервисов на части. Нелегкими задачами будут поиск среды для запуска сервиса, раскрутка нового стека сервисов и т. д. Как же со всем этим справиться? Если дело нужное, но сложное, мы должны попытаться все упростить. Снизить затраты, связанные с созданием нового сервиса, может ставка на применение библиотек и облегченных сред сервисов. Упростить предоставление и тестирование систем может обеспечение людям доступа к самообслуживаемым виртуальным машинам или даже создание платформы в качестве услуги (PaaS). В следующих главах будет рассмотрен ряд способов, позволяющих снизить эти затраты.

Резюме

Мы разбиваем систему на части путем поиска стыков, по которым могут проходить границы сервисов, и применение этого подхода может носить поэтапный характер. Совершенствуя в первую очередь поиск этих стыков и работу по снижению стоимости разбиения сервисов, мы можем продолжить наращивание и развитие систем, реагируя на все встречающиеся на этом пути требования. Как вы уже могли заметить, часть этой работы требует особого усердия. Но сам по себе тот факт, что это можно делать постепенно, означает, что этой работы не нужно бояться.

Итак, мы можем разбивать сервисы, но при этом проявляются некоторые новые проблемы. Теперь у нас намного больше движущихся частей, требующих доводки до работы в производственном режиме! Следовательно, настало время погрузиться в мир развертывания микросервисов.

6 Развертывание

Монолитные системы развертываются довольно просто. А вот с развертыванием микросервисов с их взаимозависимостями придется повозиться. Если выбрать неверный подход к развертыванию, оно настолько усложнится, что ваша жизнь превратится в сплошные мучения. В этой главе будет рассматриваться ряд приемов и технологий, призванных помочь в развертывании микросервисов в мелкоструктурных архитектурах.

Но начнем мы с того, что рассмотрим вопросы непрерывной интеграции и непрерывной поставки. Эти родственные, но различные понятия помогут наметить контуры других решений, которые будут приниматься в ходе размышлений о том, что именно создавать, как создавать и как развертывать созданное.

Краткое введение в непрерывную интеграцию

Понятие непрерывной интеграции (continuous integration (CI)) существует уже несколько лет. Но на объяснение ее основ все же стоит потратить немного времени, особенно при обдумывании отображений между микросервисами, сборок и хранилищ управления версиями, требующем рассмотрения нескольких разных вариантов.

При использовании CI основной целью является поддержка всеобщего синхронизированного состояния, которая достигается путем проверки того, что только что введенный в эксплуатацию код интегрируется с существующим кодом должным образом. Для достижения этой цели CI-сервер определяет переданный код и проводит ряд проверок, убеждаясь в том, что код скомпилирован и тесты с ним проходят без сбоев.

В качестве части данного процесса часто создается артефакт (или артефакты), используемый для дальнейшей проверки, например развертывания работающего сервиса с целью запуска для него ряда тестов. В идеале нам потребуется создавать эти артефакты только один раз и использовать их для всех развертываний той или иной версии кода. Это делается для того, чтобы избежать многократного повторения одних и тех же действий и чтобы можно было подтвердить, что развернутый нами артефакт уже прошел тестирование. Чтобы допустить повторное

использование этих артефактов, мы помещаем их в особое хранилище, либо предоставляемое самим CI-инструментарием, либо находящееся в отдельной системе.

Вскоре мы рассмотрим, какого именно сорта артефакты можно будет использовать для микросервисов, а о тестировании более подробно поговорим в главе 7.

CI предоставляет ряд преимуществ. Мы получаем некое довольно быстро формируемое представление о качестве кода. Непрерывная интеграция позволяет автоматизировать создание двоичных артефактов. Весь код, требующийся для создания артефакта, проходит самостоятельное управление версиями, поэтому при необходимости артефакт можно создать заново. Мы также получаем некоторую возможность отслеживания, проходящего от развернутого артефакта назад к его коду, и в зависимости от возможностей CI-инструментария можем наблюдать, какие тесты запущены в отношении как кода, так и артефакта. Именно эти обстоятельства и обусловили успешное применение CI.

А вам приходилось этим заниматься? Подозреваю, что вы уже использовали непрерывную интеграцию в своей организации. Если нет, то пора приступить к ее применению. Дело в том, что это основной способ, позволяющий быстрее и легче вносить изменения, и без него путь к микросервисам будет слишком труден. Мне приходилось работать с командами, которые, несмотря на заявления о применении CI, вообще ею не пользовались. Они путали применение CI-инструментария с внедрением практического использования CI. Инструментарий — лишь средство, позволяющее применить сам подход.

Мне нравятся три вопроса, которые Джез Хамбл (Jez Humble) задает людям, чтобы проверить, правильно ли они понимают, что такое CI.

- *Вы проходите ежедневную проверку в основной программе?*

  Вам нужно убедиться в том, что ваш код интегрируется в систему. Если не проводить частые проверки своего кода вместе с чьими-либо изменениями, можно существенно усложнить будущую интеграцию. Даже при использовании для управления изменениями недолговечных ответвлений интеграцию с единой основной ветвью нужно проводить как можно чаще.

- *Имеется ли у вас набор тестов для проверки правильности вносимых вами изменений?*

  Без тестов мы просто знаем, что синтаксически интеграция работает, но не знаем, не нарушим ли поведение системы. CI без ряда проверок, подтверждающих ожидаемое поведение кода, — это не CI.

- *Служит ли исправление неисправной сборки главным приоритетом команды?*

  Проход зеленой сборки означает, что изменения были безопасно интегрированы. Красная сборка означает, что последнее изменение, возможно, не прошло интеграцию. Нужно остановить все последующие проверки, не применяемые для исправления сборок, чтобы пройти тесты повторно. Если нагромоздить большое количество изменений, время исправления сборки существенно возрастет. Мне приходилось работать с командами, у которых сборка оставалась неисправной по нескольку дней, что приводило к необходимости прикладывать существенные усилия для получения в итоге проходящей тесты сборки.

Отображение непрерывной интеграции на микросервисы

Размышляя о микросервисах и непрерывной интеграции, нужно подумать о том, как средство CI создает отображение на отдельно взятые микросервисы. Как я уже не раз говорил, нужно убедиться в том, что мы можем внести изменение в отдельный микросервис и развернуть его независимо от остальных микросервисов. Как, памятуя об этом, мы отображаем отдельно взятые микросервисы на CI-сборки и исходный код?

Если начать с самого простого варианта, то можно было бы свалить все в кучу. У нас есть одно огромное хранилище, в котором находится весь код (рис. 6.1). Любая проверка этого исходного кода запустит сборку, где мы запустим все этапы верификации, связанные с микросервисами, и создадим несколько артефактов, имеющих обратную связь с той же самой сборкой.

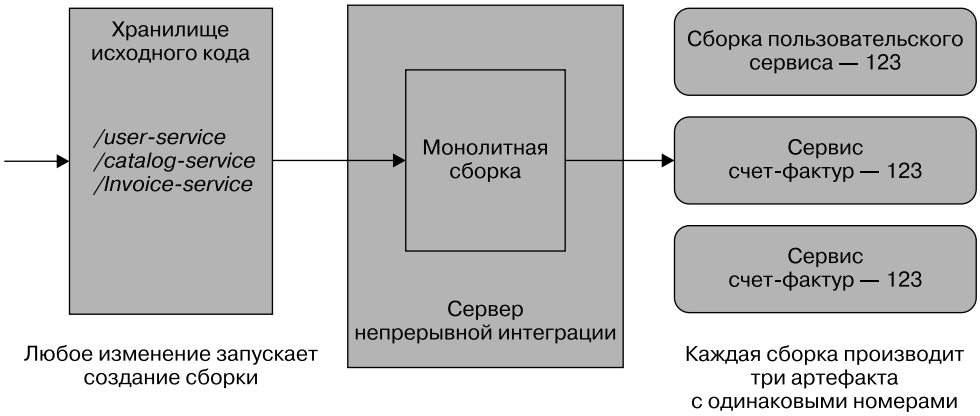


Рис. 6.1. Использование единого хранилища исходного кода и CI-сборки для всех микросервисов

На первый взгляд все кажется намного проще других подходов: меньше приходится беспокоиться о хранилищах, да и создание сборок концептуально выглядит проще. С точки зрения разработчика, все также выглядит довольно просто. Нужно лишь проверять находящийся внутри код. Если приходится работать сразу с несколькими сервисами, следует беспокоиться только об одном совершаемом действии.

Такая модель может работать весьма успешно при условии, что вы придерживаетесь замысла жестко регламентированных выпусков, где вас не смущает одновременное развертывание сразу нескольких сервисов. Вообще-то это именно тот шаблон, которого следует всячески избегать, но на самой ранней стадии проекта, особенно если над проектом работает одна команда, его кратковременное применение имеет определенный смысл.

У данной модели есть ряд весьма существенных недостатков. При внесении самых незначительных изменений в какой-либо сервис, например при изменении

поведения пользовательского сервиса, показанного на рис. 6.1, должны быть собраны и верифицированы все остальные сервисы. На это может уйти больше времени, чем требуется в других обстоятельствах, поскольку придется ожидать прохождения тестов там, где в тестировании нет никакого смысла. Это влияет на продолжительность цикла и скорость, с которой мы можем пройти путь от разработки до внедрения отдельного изменения. Но большее беспокойство вызывает мысль о том, что нужно знать, какие артефакты должны быть развернуты, а какие — нет. Знаю ли я о том, нужно ли развертывать все собранные сервисы для того, чтобы запустить мои небольшие изменения в производство? Ответ на этот вопрос может вызвать затруднения, нелегко бывает догадаться, какие сервисы действительно подверглись изменениям, в процессе простого чтения сообщений о совершении действий. Организации, использующие такой подход, часто склоняются к одновременному развертыванию всех сервисов, чего нужно избегать.

Более того, если вносимое мной в пользовательский сервис незначительное изменение испортит сборку, то, пока неисправность не будет устранена, вносить какие-либо иные изменения в другие сервисы будет невозможно. И подумайте о сценарии, при котором эту гигантскую сборку совместно используют сразу несколько команд разработчиков. Какая из них будет главной?

Разновидность этого подхода (рис. 6.2) предусматривает наличие единого дерева для всего исходного кода и нескольких CI-сборок, отображаемых на части этого дерева. При четко заданной структуре можно без особых усилий отобразить сборки на конкретные части дерева исходного кода. Вообще-то я не сторонник данного подхода, поскольку эта модель может вызывать смешанные чувства. С одной стороны, проверочный или наладочный процесс может упроститься, поскольку беспокоиться приходится только об одном хранилище. С другой — проверка исходного кода сразу для нескольких сервисов может легко войти в привычку, что может так же легко перейти в привычку внесения изменений сразу в несколько сервисов. Но этот подход по сравнению с подходом, предусматривающим создание единой сборки для нескольких сервисов, я считаю предпочтительным.

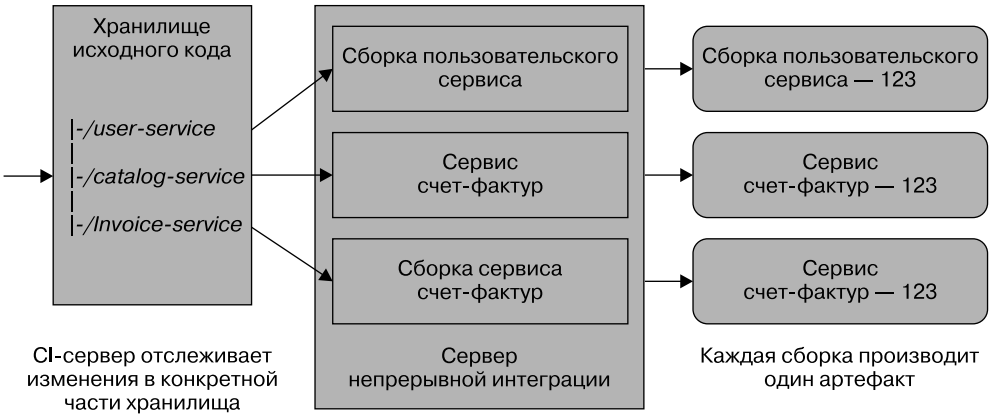


Рис. 6.2. Единое хранилище исходного кода с подкаталогами, отображаемыми на независимые сборки

А есть ли еще одна альтернатива? Предпочитаемый мною подход предполагает наличие одной CI-сборки для каждого микросервиса, что позволяет быстро вносить изменение и проверять его работоспособность еще до развертывания в производственной среде (рис. 6.3). Здесь у каждого микросервиса имеется собственное хранилище исходного кода, отображаемое на его собственную CI-сборку. При внесении изменения запускается и тестируется только нужная мне сборка. И для развертывания я получаю только один артефакт. Группировка по командной принадлежности также становится понятнее. Кто владеет сервисом, тот владеет и хранилищем и сборкой. Правда, при этом немного усложняется внесение изменений, затрагивающих сразу несколько хранилищ, но я считаю, что с этим справиться намного проще (например, с помощью сценариев командной строки), чем с теми недостатками, которые присущи монолитному процессу управления исходным кодом и сборками.

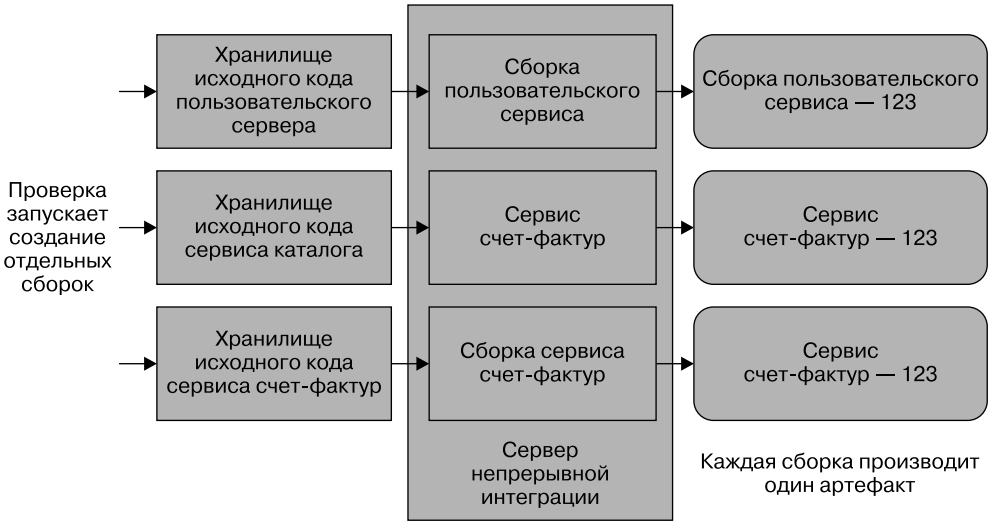


Рис. 6.3. Использование для каждого микросервиса одного хранилища и одной CI-сборки

Наряду с исходным кодом микросервисов в системе управления версиями должны находиться тесты для конкретного микросервиса, чтобы мы всегда знали, какие тесты для какого конкретного сервиса должны запускаться.

Итак, у каждого микросервиса будут собственное хранилище исходного кода и собственный процесс CI-сборки. Этот процесс в полностью автоматическом режиме будет использоваться и для готовых к развертыванию артефактов. Теперь отвлечемся от CI, чтобы увидеть, как в общую картину вписывается непрерывная доставка.

Сборочные конвейеры и непрерывная доставка

В самом начале использования непрерывной интеграции мы поняли важность наличия временами внутри сборки нескольких стадий. В качестве весьма распространенного

проявления этой особенности можно назвать тесты. Может использоваться множество быстрых, весьма ограниченных по области применения тестов и небольшое количество широкомасштабных медленных тестов. Если запустить сразу все тесты, то в режиме ожидания завершения широкомасштабных медленных тестов можно не дождаться быстрого получения ответного результата на сбой при проведении быстрых тестов. А если быстрые тесты не будут пройдены, то запускать медленные тесты не будет никакого смысла! Решением этой проблемы является использование различных стадий сборки путем создания того, что называется сборочным конвейером. Одна стадия будет для быстрых тестов, а другая — для медленных.

Концепция конвейерной сборки предоставляет отличный способ отслеживания хода обработки программы по мере прояснения ситуации на каждой стадии, помогая выяснить качество программы. Мы осуществляем сборку нашего артефакта, и этот артефакт используется на всем протяжении конвейера. По мере прохождения артефакта через все его стадии растет уверенность в том, что программа будет работать в производственной среде.

Непрерывная доставка (CD) построена на этой и некоторых других концепциях. Как подчеркивается в книге с таким же названием, авторами которой являются Джез Хамбл и Дэйв Фарли (Dave Farley), непрерывная доставка представляет собой такой подход, при котором мы получаем постоянный отклик о производственной готовности абсолютно каждой проверяемой сборки и, кроме того, каждую проверяемую сборку рассматриваем в качестве сдаточной версии.

Чтобы полностью усвоить суть данной концепции, нужно промоделировать все процессы, вовлеченные в перевод программы из проверочного в производственное состояние, и знать, на какой стадии находится каждая заданная версия программы с точки зрения уверенности в ее готовности к выпуску. При применении CD это делается путем расширенного толкования идеи многоступенчатого конвейера сборки для моделирования каждой стадии, через которую должна пройти программа, как в ручном, так и в автоматическом режиме. Простой, возможно, уже знакомый вам конвейер показан на рис. 6.4.

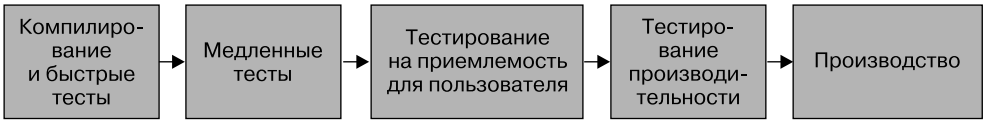


Рис. 6.4. Стандартный процесс выпуска, смоделированный в виде сборочного конвейера

И теперь нам очень нужен инструмент, включающий в себя CD в качестве одной из главных концепций. Мне встречались многие, кто пытался внедриться в CI и заняться ее расширением до получения CD, что часто заканчивалось созданием сложных систем, которые не могли считаться простыми в использовании средствами, встраиваемыми в CD с самого начала. Средства, полностью поддерживающие CD, позволяют определять и визуализировать конвейеры и моделировать весь путь программы к производству. Во время продвижения версии кода по конвейеру, пройдя одну из автоматизированных стадий верификации, она переходит к следующей стадии. Остальные стадии могут запускаться вручную. Например, для

моделирования процесса ручного тестирования на приемлемость для пользователя (UAT) при его наличии можно будет воспользоваться CD-средством. Я смогу увидеть следующую доступную сборку, готовую к развертыванию в UAT-среде, осуществить ее развертывание и, если она пройдет ручные проверки, отметить эту стадию как успешно пройденную, позволив двигаться дальше.

Благодаря моделированию всего пути программы к производству мы существенно улучшаем возможность оценить ее качество, а также можем значительно сократить время между выпусками, поскольку наблюдение за процессом сборки и выпуска ведется из одного места и у нас имеется вполне определенный координационный центр для внесения улучшений.

В мире микросервисов, где требуется обеспечить возможность выпуска сервисов независимо друг от друга, получается, что, как и в случае с CI, нужен отдельный конвейер для каждого сервиса. В наших конвейерах находится создаваемый артефакт, который нужно провести по всему пути к производству. Как всегда, оказывается, что артефакты могут иметь разнообразные размеры и формы. Вскоре мы рассмотрим некоторые наиболее распространенные и доступные варианты.

Неизбежные исключения. Как и в каждом хорошем правиле, здесь существуют исключения, также требующие рассмотрения. Подход «один микросервис на каждую сборку» должен стать вашей абсолютной целью, но есть ли случаи, когда имеет смысл воспользоваться чем-то другим? Когда команда приступает к разработке нового проекта, особенно в новом для нее ключе, когда работа начинается с чистого листа бумаги, высока вероятность того, что на пути встретится большая мешанина с точки зрения работы за пределами проложенных границ сервиса. Есть весьма веские причины задавать сервисам в начальной стадии их разработки больший размер, пока не появится понимание того, что они приобрели достаточную стабильность.

Пока все перемешано, высока вероятность внесения изменений, не соблюдающих границ сервисов, а также необходимости частого изменения того, что входит или не входит в сервисы. В этот период может быть целесообразнее содержать все сервисы в одной сборке, чтобы сократить затраты на те изменения, которые не соблюдают границ сервисов.

Но из этого не следует, что нужно позволить всем сервисам собраться в пучок. Этот период должен иметь сугубо переходный характер. Как только стабилизируются API сервисов, нужно приступать к их перемещению в собственные сборки. Если по истечении нескольких недель (или в крайнем случае пары месяцев) вам не удастся добиться стабильности в отношении границ сервисов, определяемых с целью их приемлемого разделения, снова объедините их в более монолитный сервис, сохраняя модульное разделение в пределах границ, и выделите время для того, чтобы разобраться с областью их применения. В таком подходе отражается опыт нашей собственной команды SnapCI, уже упоминавшийся в главе 3.

Артефакты для конкретных платформ

У многих технологических стеков имеются как своеобразные высококачественные артефакты, так и инструментальные средства, поддерживающие их создание и установку. У Ruby есть Gem-пакеты, у Java — JAR- и WAR-файлы,

а у Python — egg-пакеты. Разработчики с опытом работы с одним из стеков будут неплохо разбираться в работе с этими артефактами (и, надеюсь, в их создании).

Но с точки зрения создателя микросервисов, зависящей от выбранного вами технологического стека, этот артефакт может быть несамодостаточным. Наряду с тем, что в Java JAR-файл может быть создан исполнительным и способным запускать встроенный HTTP-процесс, для Ruby- и Python-приложений ожидается применение диспетчера процессов, запускаемого внутри Apache или Nginx. Поэтому нам может понадобиться некий способ установки и настройки других программ, необходимых для развертывания и запуска наших артефактов. В этом случае можно обратиться за помощью к таким средствам автоматизированного управления настройками, как Puppet и Chef.

Другим недостатком в данном случае является то, что эти артефакты относятся только к определенному технологическому стеку, что может еще больше усложнить развертывание при задействовании набора технологий. Оцените складывающуюся ситуацию с точки зрения того, кто пытается развернуть сразу несколько сервисов. Вполне возможно, что какому-нибудь разработчику или тестировщику захочется протестировать работу некоторых функций или же понадобится кто-то, берущий на себя управление развертыванием производственных сборок. А теперь представьте, что в имеющихся сервисах используются три совершенно разных механизма развертывания. Возможно, это Ruby Gem, JAR-файл и nodeJS NPM-пакет. Кто-нибудь скажет вам спасибо?

Существенную помощь в скрытии различий в механизмах развертывания имеющихся артефактов может оказать автоматизация. Кроме того, несколько различных широко распространенных сборок артефактов на основе конкретных технологий поддерживаются такими средствами, как Chef, Puppet и Ansible. Но есть несколько различных типов артефактов, работать с которыми можно еще проще.

Артефакты операционных систем

Одним из способов избежать проблем, связанных с артефактами на основе конкретных технологий, является создание артефактов, характерных для используемой операционной системы. Например, для систем на основе RedHat или CentOS можно создать RPM-пакеты, для Ubuntu — deb-пакет или для Windows — MSI-пакет.

Преимущество использования артефактов, характерных для той или иной операционной системы, заключается в том, что по отношению к развертыванию не нужно обращать внимания на то, какая именно технология положена в основу артефакта. Для установки пакета просто применяются средства, характерные для используемой операционной системы. Эти средства могут помочь также в удалении пакетов или получении о них нужной информации или даже предоставить хранилища пакетов, в которые наши CI-средства могут помещать свои данные. Основная часть работы, проделываемая диспетчером пакетов операционной системы, может также содержать ту работу, которую иначе пришлось бы выполнять в таких средствах, как Puppet или Chef. К примеру, на всех Linux-платформах, которыми я пользовался, можно определить зависимости

ваших пакетов от других пакетов, и средства операционной системы автоматически установят для вас и эти пакеты.

В качестве недостатка можно в первую очередь назвать сложность создания пакетов. Что касается Linux, диспетчер пакетов FPM предоставляет весьма неплохую абстракцию для создания пакетов этой операционной системы, а преобразование из развертывания на основе tarball в развертывание на основе средств операционной системы может проводиться без особого труда. С Windows дела обстоят немного сложнее. Возможности присущей ей системы создания пакетов в виде MSI-установщиков и т. п. по сравнению с возможностями, предоставляемыми Linux, оставляют желать лучшего. Помочь в решении возникающих сложностей, по крайней мере в плане содействия управлению библиотеками, используемыми при разработках, призвана система создания пакетов NuGet. В последнее время этот замысел был расширен — выпущено средство Chocolatey NuGet, предоставляющее диспетчер пакетов для Windows, разработанный для развертывания инструментальных средств и сервисов, который намного больше похож на диспетчеры пакетов в Linux-среде. Конечно же, это шаг в правильном направлении, хотя идиоматический стиль в Windows, попрежнему связанный с развертыванием каких-либо средств в IIS, означает, что такой подход может стать непривлекательным для некоторых команд, ведущих разработки под Windows.

Другим недостатком может быть, конечно же, развертывание, проводимое в нескольких различных операционных системах. Издержки, связанные с управлением артефактами для различных операционных систем, могут быть слишком велики. Если создаются приложения, предназначенные для установки другими людьми, может создаться безвыходное положение. Но если программа устанавливается на машины, находящиеся в вашем распоряжении, то я бы предложил вам присмотреться к унификации или, по крайней мере, сокращению числа используемых операционных систем. Тем самым можно существенно уменьшить колчество вариантов поведения при переходе от машины к машине и упростить решение задач развертывания и обслуживания.

В общем, встречавшиеся мне команды, перешедшие на управление пакетами на основе средств операционных систем, упростили свой подход к развертыванию и стремятся не попадать в ловушку больших и сложных сценариев развертывания. Это может стать хорошим способом упрощения развертывания микросервисов при использовании разнородных технологических стеков, особенно если работа ведется под управлением Linux.

Настраиваемые образы

Одной из проблем использования таких автоматизированных систем управления настройками, как Puppet, Chef и Ansible, может стать время, затрачиваемое на выполнение can-сценариев на машине. Рассмотрим простой пример сервера, предназначенного и настроенного для развертывания Java-приложения. Предположим, что для предоставления сервера, использующего стандартный образ Ubuntu, применяется AWS. Первое, что нужно сделать, — установить Oracle JVM для запуска

Java-приложения. Я видел, что этот простой процесс занимал около пяти минут, при этом две минуты затрачивались предоставляемой машиной и чуть больше времени требовалось на установку JVM-машины. Затем можно было подумать о размещении на ней нашей программы.

Вообще-то это весьма тривиальный пример. Чаще всего приходится устанавливать другие объемы часто востребуемых программных средств. Например, может потребоваться использование collectd для сбора статистических сведений об операционной системе, logstash — для регистрации собранных данных, а также, возможно, придется устанавливать соответствующие объемы nagios для мониторинга систем (более подробно эта программа рассматривается в главе 8). Со временем может добавляться что-нибудь еще, что станет требовать все большего времени, затрачиваемого на предоставление всех этих зависимостей.

Puppet, Chef, Ansible и им подобные средства могут быть достаточно интеллектуальными, чтобы не устанавливать уже имеющиеся программы. Это не означает, что выполнение сценариев на существующих машинах всегда будет осуществляться быстро, к сожалению, требует времени и выполнение всех проверок. Хотелось бы также избежать слишком долгого задействования машин и не хотелось бы допускать слишком широкого разброса настроек (этот вопрос вскоре будет рассмотрен более подробно). И если используется компьютерная платформа, выделяемая по требованию, то мы могли бы ежедневно, если не чаще, выключать и запускать новые экземпляры, поэтому заявленные особенности этих средств управления настройками могут иметь весьма ограниченное применение.

Со временем наблюдение за одним и тем же инструментарием, устанавливающимся снова и снова, может превратиться в зеленую тоску. Если пытаться сделать это по нескольку раз в день, возможно, как часть разработки или CI, то в плане обеспечения быстрой отдачи это становится настоящей проблемой. Это может привести и к увеличению простоя при развертывании производственной версии, если ваша система не позволяет выполнять развертывание без простоев, поскольку перед установкой программы придется ждать установки на машины всех предварительно востребованных средств. Смягчить условия могут модели синего или зеленого развертывания (рассматриваются в главе 7), позволяющие проводить развертывание новой версии сервиса, не переводя старую версию в автономный режим работы.

Один из подходов, позволяющий сократить подготовительный период, заключается в создании образа виртуальной машины, в котором зафиксирован ряд наиболее часто встречающихся из используемых нами зависимостей (рис. 6.5). Все платформы виртуализации, которыми мне приходилось пользоваться, позволяли создавать собственные образы, и средства, предназначенные для решения этой задачи, намного совершеннее тех, что существовали всего несколько лет назад. Это меняет наши взгляды. Теперь можно зафиксировать самые распространенные средства в нашем собственном образе. Когда потребуется развернуть программу, нужно будет запустить экземпляр этого заранее настроенного образа, и тогда останется только установить самую последнюю версию сервиса.

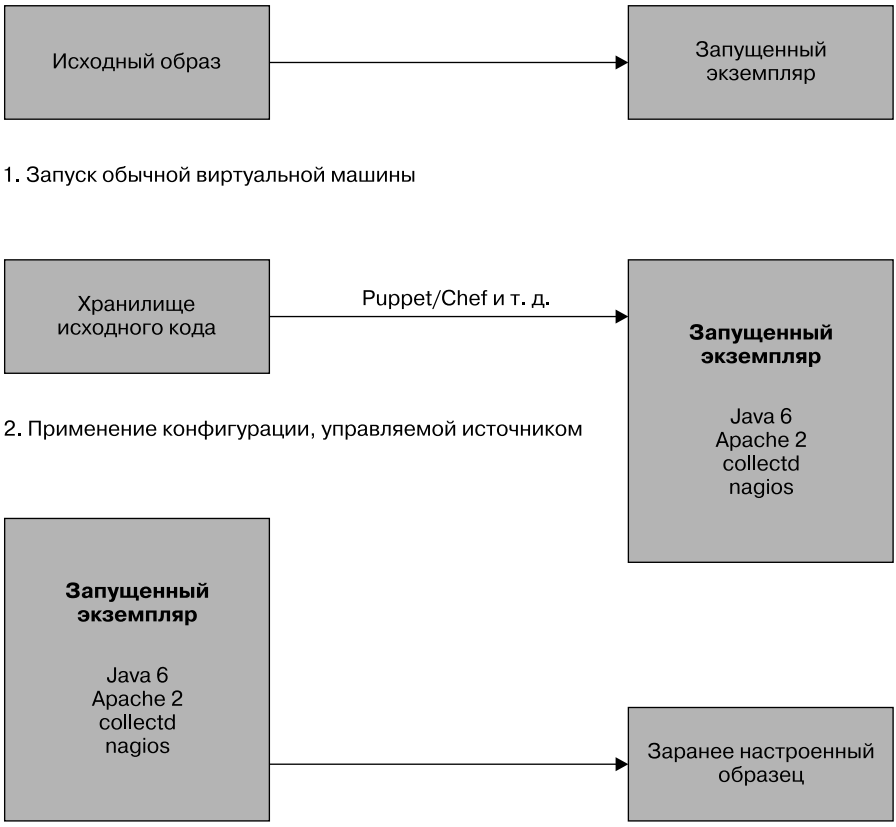


Рис. 6.5. Создание заранее настроенного VM-образа

Образ создается всего один раз, поэтому при последующих запусках его копий не нужно тратить время на установку зависимостей, поскольку они уже есть в образе. Это может привести к существенной экономии времени. Если основные зависимости не меняются, новые версии сервиса могут продолжать использовать исходный образ.

Но у этого подхода имеются некоторые недостатки. Создание образов может занять довольно много времени. Это означает, что для разработчиков можно применять другие способы развертывания сервисов, чтобы не заставлять их ждать по полчаса только для того, чтобы создать развертывание двоичной среды. Второй недостаток заключается в том, что получающиеся образы могут быть слишком большими. При создании собственных VMWare-образов это может стать реальной проблемой, поскольку переместить по сети, к примеру, образ объемом 20 Гбайт не так-то просто. Вскоре будет рассмотрена контейнерная технология, в частности такое средство, как Docker, позволяющее избавиться от некоторых упомянутых здесь недостатков.

Исторически сложилось так, что одной из проблем является изменение от платформы к платформе цепочки инструментов, требующейся для создания нужного нам образа. Создание VMWare-образа отличается от создания AWS AMI, Vagrant- или Rackspace-образа. Конечно, если у вас повсеместно одна и та же платформа, то волноваться не о чем, но этим могут похвастаться далеко не все организации. И даже если это так, с инструментами в этом пространстве зачастую трудно работать и они не всегда хорошо вписываются в тот инструментарий, который может использоваться для конфигурации машины.

Чтобы значительно упростить создание образа, было создано средство Packer. Используя настроечные сценарии по вашему выбору (поддерживаются Chef, Ansible, Puppet и многие другие средства), Packer позволяет создавать образы для различных платформ из одной и той же конфигурации. На момент написания книги это средство поддерживало VMWare, AWS, Rackspace Cloud, Digital Ocean и Vagrant, и мне встречались команды, успешно использующие его для создания образов под Linux и Windows. Это означает, что из одной и той же конфигурации можно создавать образ для развертывания в вашей производственной среде AWS и соответствующего Vagrant-образа для локального развертывания и тестирования.

Образы как артефакты

Итак, мы можем создать образы виртуальной машины с зафиксированными зависимостями для ускорения отдачи, но стоит ли на этом останавливаться? Можно ведь пойти дальше и зафиксировать сервис в самом образе, приспособить модель артефакта сервиса в качестве образа. Теперь, запуская образ, мы получаем готовый к работе сервис. Этот действительно быстрый способ подготовки к работе является причиной того, что в Netflix была принята модель фиксирования его собственных сервисов в качестве AWS AMI-образов.

Как и в случае применения пакетов, характерных для конкретной операционной системы, эти VM-образы становятся отличным способом абстрагирования от различий в технологических стеках, используемых для создания сервисов. Разве нас волнует, что сервис, запущенный в образе, написан на Ruby или Java и использует gem-пакет или JAR-файл? Нам интересно только то, что все это работает. Тогда мы можем сконцентрировать усилия на автоматизации создания и развертывания этих образов. К тому же это становится отличным способом реализации другой концепции развертывания — неизменяемого сервера.

Неизменяемые серверы

При хранении всех наших настроек в системе управления версиями мы стремимся обеспечить возможность автоматизированного воспроизводства сервисов и, надеюсь, при желании — всей среды. Но что случится, если мы запустили процесс развертывания, а кто-то войдет в систему и внесет изменения независимо от того, что находится в системе управления версиями? Эту проблему часто называют дрейфом конфигурации — код в системе управления версиями больше не является отражением конфигурации на работающем хосте.

Во избежание этого можно обеспечить невозможность внесения изменений в работающий сервис. Любое, даже самое незначительное, изменение должно пройти через сборочный конвейер, чтобы была создана новая машина. Эту схему можно реализовать без использования развертывания на основе образа, но оно также является логическим расширением использования образов в качестве артефактов. Например, в ходе создания образа можно отключить SSH, гарантируя тем самым, что никто не сможет даже войти в систему для внесения какого-либо изменения!

Разумеется, ранее высказанные предостережения относительно времени цикла применимы и в данном случае. И нужно также обеспечить, чтобы интересующие нас данные, сохраненные в компьютере, хранились бы в другом месте. Наряду с этими сложностями мне приходилось наблюдать случаи, когда использование этой схемы приводило к более простым развертываниям и созданию более обоснованных сред. И, как уже было сказано, нужно делать все, чтобы добиться упрощения!

Среды

При проходе программы через стадии CD-конвейера она также будет развернута в средах различных типов. Если обратиться к примеру сборочного конвейера (см. рис. 6.4), то нужно, наверное, рассмотреть как минимум четыре различные среды: для запуска медленных тестов, тестирования на приемлемость для пользователя, тестирования на производительность и работы в производственном режиме. Наш микросервис должен быть везде одинаков, но среды будут разными. По крайней мере, они должны быть обособленными, с отдельными наборами конфигураций и хостов. Но зачастую они могут отличаться друг от друга еще значительнее. Например, среда, предназначенная для работы сервиса в производственном режиме, может состоять из нескольких сбалансированных по нагрузке хостов, разбросанных по двум дата-центрам, а все компоненты среды, предназначенной для тестирования, могут быть запущены всего на одном хосте. Такие различия в средах могут вызвать ряд проблем.

Мне приходилось испытывать это на себе много лет назад. Мы развертывали веб-сервис Java в производственном кластерном контейнере WebLogic-приложения. Этот WebLogic-кластер реплицировал состояние сессии между несколькими узлами, обеспечивая некоторую степень устойчивости при сбое отдельного узла. Но лицензии на использование WebLogic, под которыми было развернуто приложение, а также машины, на которых оно было развернуто, обходились довольно дорого. Поэтому среда для тестирования и программы были развернуты на одной машине в некластерной конфигурации.

И во время одного из выпусков это обстоятельно нанесло нам существенный вред. Чтобы у WebLogic была возможность копировать состояние сессии между узлами, данные сессии должны поддаваться сериализации. К сожалению, одна из наших фиксаций разрушала этот процесс, поэтому при развертывании для работы в производственном режиме репликация сессии дала сбой. В итоге мы решили проблему, приложив большие усилия для имитации кластеризованной настройки в нашей среде тестирования.

Предназначенный для развертывания сервис одинаков для различных сред, но каждая из сред служит для разных целей. На разработочном ноутбуке мне нужно быстро развернуть сервис, имея в потенциале заглушки, имитирующие его сотрудников, чтобы запустить тесты или выполнить ручную проверку поведения, в то время как при развертывании в производственной среде мне может понадобиться развернуть несколько копий сервиса для соблюдения сбалансированности нагрузки, возможно, с разбрасыванием по одному или нескольким дата-центрам из соображений живучести.

По мере перехода с ноутбука на сборочный сервер, с него — в среду тестирования на приемлемость для пользователя, а дальше — в производственную среду нужно обеспечивать, чтобы среды постепенно приобретали все больше и больше свойств производственной среды, чтобы можно было отловить проблемы, связанные с разницей этих сред, как можно раньше. Этот баланс будет носить постоянный характер. Иногда затраты времени и средств на воспроизводство сред, сходных с производственными, бывают непомерно высокими, поэтому приходится идти на компромиссы. Кроме того, иногда использование сред, сходных с производственными, может замедлить циклы получения ответных результатов, поскольку ожидание того, пока на 25 машинах установится ваша программа в AWS, может быть намного продолжительнее простого развертывания сервиса, к примеру на локальном Vagrant-экземпляре.

Баланс между средами, подобными производственной среде, и быстрым получением ответных результатов не будет статичным. Следите за количеством ошибок, выявляемых по мере прохождения сред, и за показателями сроков получения обратных ответов и перенастраивайте этот баланс по мере необходимости.

Управление средами для монолитных систем с одним артефактом может даваться весьма нелегко, особенно если у вас нет доступа к легко автоматизируемым системам. А если подумать о нескольких средах для каждого микросервиса, то все может показаться еще страшнее. Вскоре мы рассмотрим ряд различных платформ развертывания, которые могут существенно упростить складывающуюся ситуацию.

Конфигурация сервиса

Наши сервисы нуждаются в настройках. В идеале их должно быть немного и они должны быть ограничены теми свойствами, которые меняются от одной среды к другой, например ответами на вопросы о том, какими именем пользователя и паролем нужно воспользоваться для подключения к базе данных. Настройка этих изменений при переходе от одной среды к другой должна сводиться к абсолютному минимуму. Чем больше настроек изменяет основное поведение сервиса и чем больше эти настройки варьируются от одной среды к другой, тем больше будет возникать проблем, касающихся только определенных сред и проявляющихся в экстремальных ситуациях.

Следовательно, если у нас есть некая настройка для сервиса, изменяющаяся от одной среды к другой, как мы можем справиться с этим как с частью процесса развертывания? Одним из вариантов является создание по одному артефакту для

каждой среды с конфигурацией внутри самого артефакта. Поначалу этот вариант представляется весьма разумным. Конфигурация встроена, нужно лишь развернуть артефакт, и все будет работать. Но так ли это? Не так-то все просто. Вспомните о концепции непрерывной доставки. Нам нужно создать артефакт, представляющий сдаточную версию, и провести его по конвейеру, убеждаясь, что он вполне подходит для передачи в производство. Представим, что мною созданы артефакты для тестирования клиентской службы — Customer-Service-Test и для ее работы в производственном режиме — Customer-Service-Prod. Если артефакт Customer-Service-Test пройдет тесты, а я фактически развертываю артефакт Customer-Service-Prod, могу ли я быть уверенным в том, что проверена была программа, которая окажется в производстве?

Существуют и другие трудности. Во-первых, на создание этих артефактов затрачивается дополнительное время. Во-вторых, на время сборки нужно знать, какие среды существуют. А как справиться с важными конфиденциальными данными? Мне не нужна информация о паролях производственного режима, проверяемая вместе с исходным кодом, но она нужна во время создания сборки для создания всех этих артефактов, и этого очень трудно избежать.

Более подходящим подходом будет создание одного-единственного артефакта и раздельное управление конфигурацией. Это могут быть файл свойств, имеющийся для каждой среды, или различные параметры, передаваемые в процесс установки. Еще одним широко распространенным вариантом, особенно при работе с большим количеством микросервисов, является использование для предоставления конфигурации специально выделенной системы, которая подробнее исследуется в главе 11.

Отображение сервиса на хост

Когда речь заходит о микросервисах, в самом начале возникает вопрос: «А сколько сервисов может быть на каждой машине?» Перед тем как дать ответ, нужно подобрать более подходящее понятие, чем «машина» или даже использованное мною ранее более универсальное понятие «стойка». В нашу эру виртуализации отображение между отдельно взятым хостом, на котором работает операционная система, и используемой физической инфраструктурой может варьироваться в самых широких пределах. Я, например, склонен говорить о хостах, используя их как универсальную единицу изолированности, а именно как операционную систему, под управлением которой могу установить и запустить свои сервисы. Если развертывание ведется вами непосредственно на физические машины, тогда один физический сервер отображается на один хост (что, возможно, будет не вполне корректной терминологией в данном контексте, но за неимением лучшей может считаться подходящей). Если используется виртуализация, то отдельная физическая машина может отображаться на несколько независимых хостов, каждый из которых может содержать один сервис и более.

Следовательно, когда мы думаем о различных моделях развертывания, речь идет о хостах. Итак, сколько же у нас должно быть сервисов на хост?

У меня есть вполне определенный взгляд на то, какой из моделей отдать предпочтение, но есть несколько факторов, которые нужно рассмотреть, определяя,

какая из моделей будет самой подходящей именно для вас. Кроме того, важно понять, что некоторые варианты, которые мы выбираем в связи с этим, будут ограничивать круг доступных нам вариантов развертывания.

Несколько сервисов на каждый хост

Размещение на каждом хосте нескольких сервисов (рис. 6.6) по ряду причин является весьма привлекательным решением. Во-первых, из соображений более простого управления хостом. Там, где инфраструктурой управляет одна команда, а программой другая, рабочая нагрузка той команды, которая управляет инфраструктурой, зачастую зависит от количества управляемых хостов. Если на отдельном хосте будет размещено больше сервисов, то с увеличением их количества рабочая нагрузка, связанная с управлением хостами, не увеличится. Во-вторых, нужно принять во внимание расходы. Даже при наличии доступа к виртуализированной платформе, позволяющего и предоставлять виртуальные хосты, и изменять их размеры, виртуализация может повлечь за собой накладные расходы, сокращающие исходные ресурсы, доступные вашим сервисам. На мой взгляд, обе эти проблемы могут быть решены с помощью новых рабочих приемов и технологий, которые вскоре будут рассмотрены.

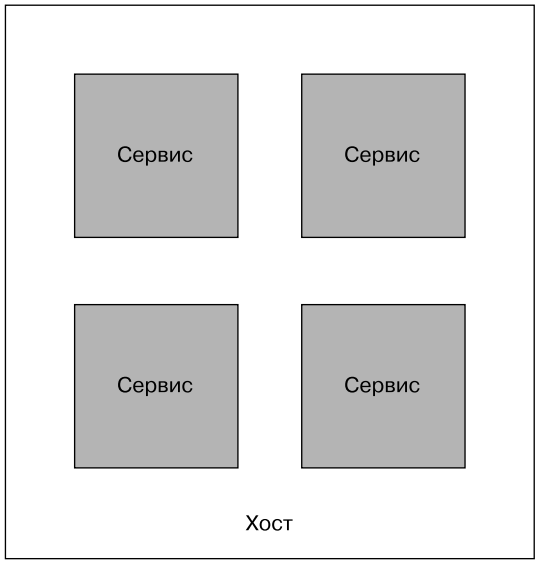


Рис. 6.6. Размещение на каждом хосте нескольких сервисов

Эта модель знакома и тем, кто вел развертывание в каких-нибудь разновидностях приложений-контейнеров. В некотором роде использование приложений-контейнеров является особым случаем модели нескольких сервисов на каждом хосте, поэтому мы рассмотрим этот вопрос отдельно. Эта модель может упростить жизнь разработчика. Развертывание нескольких сервисов на одном хосте, работающем в производственном режиме, равносильно развертыванию нескольких

сервисов на локальной рабочей станции или ноутбуке. Если нужно посмотреть на альтернативные модели, следует найти способ сохранения концептуальной простоты для разработчиков.

Но у этой модели есть ряд проблем. Так, она может затруднить проведение мониторинга. Например, нужно ли при отслеживании загруженности центрального процессора отслеживать загруженность процессора одним сервисом независимо от других? Или нужно ли рассматривать центральный процессор компьютерной стойки как единое целое? Трудно будет также избежать побочных эффектов. Если один сервис находится под большой нагрузкой, это может выразиться в сокращении объемов ресурсов, доступных другим частям системы. Компания Gilt столкнулась с этой проблемой при увеличении количества запущенных сервисов. Изначально на одной стойке сосуществует множество сервисов, но при неравномерной нагрузке на один из сервисов мы получим вредное влияние на работу всех остальных сервисов, запущенных на хосте. Ко всему прочему, это еще больше затруднит анализ влияния сбоев хоста — выход одного узла из строя может вызвать большой резонансный эффект.

Может также стать более трудным развертывание сервисов, поскольку забот прибавляет необходимость обеспечить отсутствие отрицательного влияния одного развертывания на другие. Например, если для подготовки хоста используется Puppet, но у каждого сервиса имеются различные (потенциально несовместимые) зависимости, то как можно справиться с этой работой? Мне встречались специалисты, которые при наихудшем развитии событий в попытке упростить развертывание нескольких сервисов на одном хосте связывали вместе развертывание сразу нескольких сервисов и развертывали на этом хосте несколько различных сервисов в один прием. На мой взгляд, незначительные упрощения не перевесят отказа от одного из ключевых преимуществ микросервисов — стремления к независимым выпускам программных средств. Если внедрять модель размещения нескольких сервисов на одном хосте, нужно убедиться в том, что у вас есть идея, как каждый сервис может быть развернут независимо от всех остальных.

Эта модель может также воспрепятствовать автономии команд. Если сервисы, поддерживаемые разными командами, установлены на один и тот же хост, то кто получит возможность настройки хоста под свои сервисы? По всей вероятности, это закончится получением управления централизованной командой, которая будет иметь больше возможностей для координации развертывания сервисов.

Еще одна проблема состоит в том, что этот вариант может ограничить варианты развертывания артефактов. Если несколько различных серверов не будут связаны друг с другом в единый артефакт, чего нужно всеми силами избегать, то развертывания на основе образов будут недоступны, равно как и неизменяемые серверы.

Факт наличия нескольких сервисов на одном хосте означает, что для масштабирования особо нуждающегося в этом сервиса нужно будет приложить огромные усилия. Аналогично этому, если один микросервис обрабатывает данные и проводит операции с весьма высокой степенью конфиденциальности, для этого может потребоваться иная настройка используемого хоста или даже помещение самого хоста в отдельный сегмент сети. Присутствие абсолютно всего на одном хосте означает, что мы можем в конечном счете прийти к тому, что ко всем сервисам нужно будет относиться одинаково, даже если их потребности отличаются друг от друга.

Как говорит мой коллега Нил Форд (Neal Ford), многие наработанные нами приемы развертывания и управления хостом представляют собой попытку оптимизировать дефицит ресурсов. В прошлом при необходимости получения другого хоста у нас был единственный выбор: купить или арендовать другую физическую машину. Зачастую время выполнения заказа было весьма продолжительным, что приводило к возникновению долгосрочных финансовых обязательств. Поставка новых серверов лишь через каждые два-три года была для клиентов, с которыми мне приходилось работать, вполне обычным явлением, и получить дополнительные машины вне этого графика было очень трудно. Но компьютерные платформы, предоставляемые по требованию, резко снизили цены на вычислительные ресурсы, а развитие технологий виртуализации означало даже для внутренней инфраструктуры на основе использования хостов придание ей большей гибкости.

Приложения-контейнеры

Если вы знакомы с развертыванием .NET-приложений за IIS- или Java-приложениями в контейнер сервлетов, то должны были хорошо ознакомиться с моделью, в которой несколько отличающихся друг от друга сервисов или приложений находятся внутри единого приложения-контейнера, которое, в свою очередь, находится на отдельном хосте (рис. 6.7). Замысел состоит в том, что приложение-контейнер, в котором находятся ваши сервисы, дает вам преимущества с точки зрения повышения управляемости, например поддержку кластеризации для обработки группировок из нескольких собранных вместе экземпляров, инструменты для проведения мониторинга и т. п.

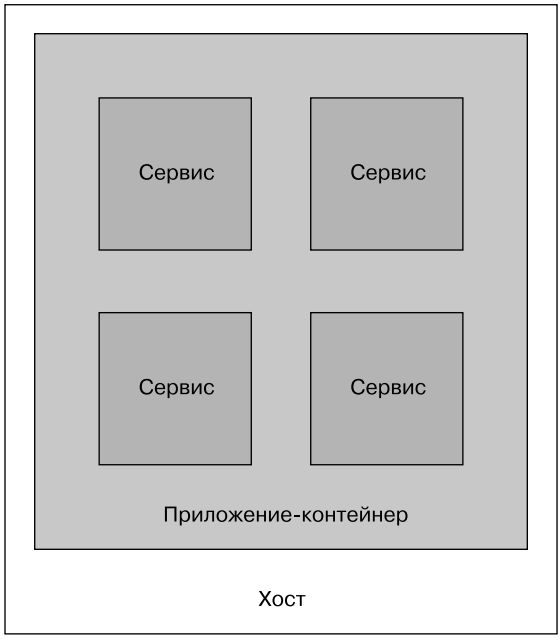


Рис. 6.7. Размещение на каждом хосте нескольких сервисов

Эта настройка может выразиться также в преимуществах по сокращению издержек, присущих средам выполнения программ на тех или иных языках программирования. Рассмотрим выполнение пяти Java-сервисов в одном контейнере сервлетов Java. Все издержки сводятся к одной-единственной JVM-машине. Сравним это с выполнением пяти независимых JVM-машин на том же самом хосте при использовании встроенных контейнеров. Тем не менее я все же чувствую, что эти приложения-контейнеры имеют довольно много недостатков, и вам нужно хорошенько подумать, прежде чем решить, что без них вам никак не обойтись.

Первым из недостатков нужно отметить то, что они неизбежно ограничивают выбор технологий. Вам приходится использовать технологический стек. Это может ограничить не только выбор технологий для реализации самого сервиса, но и варианты автоматизации ваших систем и управления ими. В соответствии с тем, что вскоре будет рассматриваться, одним из способов, позволяющих избежать издержек управления несколькими хостами, является автоматизация, при которой ограничение доступных вариантов разрешения проблемы может нанести двойной ущерб.

Я также усомнился бы в некоторых полезных свойствах контейнера. Многие из них рекламируют в качестве возможностей управлять кластерами для поддержки совместно используемого хранящегося в памяти состояния сессии, то есть того, чего мы, безусловно, стремимся избежать в любом случае из-за сложностей, возникающих при масштабировании сервисов. Недостаточно будет и обеспечиваемых ими контролирующих возможностей, поскольку, как будет показано в главе 8, нам в мире микросервисов нужен присоединяемый мониторинг. Многие из контейнеров имеют довольно длительный период подготовки к работе, отрицательно влияющий на циклы получения разработчиками ответной информации.

Имеются и другие проблемы. Попытки получения приемлемого управления жизненным циклом приложений поверх таких платформ, как JVM, могут вызвать ряд проблем и представляются более сложными, нежели простой перезапуск JVM. Анализ использования ресурсов и потоков также дается намного сложнее, поскольку приходится иметь дело с несколькими приложениями, совместно использующими один и тот же процесс. И запомните, даже если вы действительно видите ценность в контейнерах, создаваемых по конкретным технологиям, бесплатно они вам не обойдутся. Кроме того что многие из них носят коммерческий характер и поэтому имеют определенную стоимость, сами по себе они добавляют издержки на ресурсы.

В конечном счете этот подход является еще одной попыткой оптимизировать дефицит ресурсов, которые могут просто больше не вмещаться. Если вы примете решение в качестве модели развертывания разместить несколько сервисов на одном хосте, я бы настоятельно рекомендовал присмотреться к независимо развертываемым микросервисам как к артефактам. Применяя .NET-технологию, этого можно добиться с помощью использования таких средств, как Nancy, а в Java эта модель поддерживается на протяжении многих лет. Например, имеющий довольно почтенный возраст встроенный контейнер Jetty создается для весьма облегченного автономного HTTP-сервера, составляющего ядро стека Dropwizard. Известно, что в Google встроенные Jetty-контейнеры весьма успешно используются для непосредственного обслуживания статического контента и, следовательно, что все это может работать в широких масштабах.

Размещение по одному сервису на каждом хосте

При использовании модели размещения по одному сервису на каждом хосте (рис. 6.8) мы избавляемся от побочных эффектов, характерных для нескольких сервисов, размещаемых на одном хосте, существенно упрощая возможности контроля и внесения исправлений. Потенциально сокращается и количество критических точек сбоя. Выход из строя одного хоста повлияет только на один сервис, хотя это не всегда так, когда используется виртуализированная платформа. Более подробно создание разработок с прицелом на масштабирование и устойчивость к сбоям рассматривается в главе 11. Кроме того, упрощается масштабирование одного сервиса независимо от всех других, а сосредоточение внимания только на сервисе и требующем этого внимания хосте упрощают решение вопросов обеспечения безопасности.

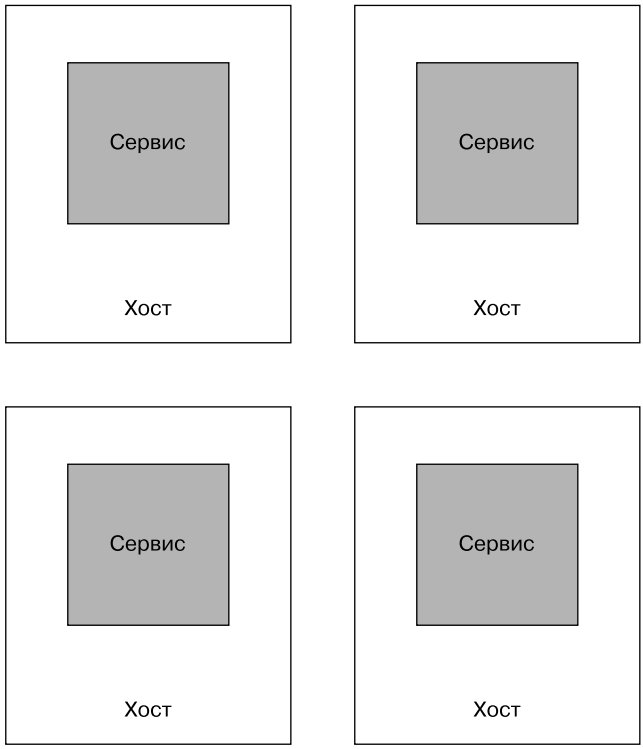


Рис. 6.8. Размещение по одному сервису на каждом хосте

Не менее важно и то, что мы открыли потенциал для использования альтернативных технологий развертывания, например рассмотренных ранее технологий развертывания на основе применения образов или технологий развертывания с использованием неизменяемого сервера.

При внедрении архитектуры микросервисов все сильно усложняется. Выискивать источники всевозможных сложностей хотелось бы в последнюю очередь.

Я считаю, что если у вас нет доступной и жизнеспособной PaaS-платформы, то эта модель хорошо справляется с сокращением общесистемных сложностей. Модель размещения по одному сервису на каждом хосте существенно легче поддается обоснованию и может содействовать упрощению системы. Если вы все же не сможете принять эту модель, я не стану говорить, что микросервисы не для вас. Но при этом хочу предложить, чтобы вы присмотрелись к переходу на эту модель в будущем, поскольку она позволяет уменьшить сложности, которые могут быть привнесены архитектурой микросервисов.

Но в увеличивающемся количестве хостов также кроются потенциальные недостатки. Растут число серверов, требующих управления, и затраты, связанные с запуском большего количества отличных друг от друга хостов. Несмотря на эти проблемы, все же это модель, которой я отдаю предпочтение при создании микросервисных архитектур. И вскоре мы рассмотрим ряд мероприятий, которые можно выполнить для снижения издержек управления большим количеством хостов.

Платформа в качестве услуги

При использовании платформы в качестве услуги (PaaS) работа ведется на более высоком уровне абстракции, чем когда дело касается отдельного хоста. Большинство таких платформ полагаются на принятие артефактов с конкретными технологиями, такими как Java WAR-файл или gem-пакет Ruby, и автоматически предоставляют и запускают их для вас. Некоторые из этих платформ будут самостоятельно пытаться подгонять под ваши нужды масштаб системы вверх и вниз, хотя более распространенный (и, исходя из моего опыта, менее подверженный ошибкам) способ позволяет сохранить за вами контроль над количеством узлов, на которых может быть запущен сервис, справляясь со всем остальным самостоятельно.

На момент написания этих строк большинство самых лучших, наиболее совершенных решений PaaS уже было принято. Когда думаешь о первоклассных средствах в этой области, на память приходит Heroku. Эта платформа не только справляется с запуском вашего сервиса, но и с легкостью поддерживает такие сервисы, как базы данных.

В этой области есть и самостоятельно принимаемые решения, хотя они и менее зрелые, чем те решения, которые уже были приняты.

Когда PaaS-решения работают хорошо, то качества их работы вполне достаточно. Но когда они подходят вам не в полной мере, зачастую вы испытываете дефицит контроля в области внесения каких-либо внутренних исправлений. В этом заключается часть принимаемых нами компромиссов. Хочу заметить, что, исходя из моего опыта, чем более интеллектуальными пытаются быть PaaS-решения, тем чаще допускают ошибки. Мне приходилось пользоваться несколькими PaaS-платформами, которые пытались выполнять автоматическое масштабирование на основе востребованности приложением, но делали это крайне неудачно. Конечно же, эвристика, являющаяся движущим механизмом этого интеллекта, старается подстроиться под некое среднее приложение, а не под ваш конкретный случай. Чем более нестандартный характер имеет приложение, тем выше вероятность того, что оно не сможет идеально сработаться с PaaS-платформой.

Поскольку качественные PaaS-решения справляются с решением довольно многих задач за вас, они могут стать замечательным способом избавления от растущих издержек, возникающих с появлением все большего количества движущихся частей. И тем не менее я все еще не уверен, что в это пространство укладываются абсолютно все модели, а ограниченные, самостоятельно принимаемые варианты означают, что этот подход для решения ваших задач может не сработать. Но я ожидаю, что в ближайшее десятилетие мы станем скорее нацеливаться на развертывание PaaS-платформы, чем развертывать отдельные сервисы на самоуправляемых хостах.

Автоматизация

Ответ на столь большое количество вопросов, заданных до сих пор, сводится к автоматизации. При небольшом количестве машин всем можно управлять вручную. Для меня это дело привычное. Помню, когда в работе был небольшой набор производственных машин, я собирал регистрируемые данные, развертывал программные средства и контролировал процессы, заходя в вычислительную систему вручную. Казалось, что моя продуктивность была ограничена количеством окон терминала, открываемых одновременно, а второй монитор стал огромным шагом вперед. Но все это рухнуло очень быстро.

Один из негативных взглядов на размещение по одному сервису на каждом хосте основывался на впечатлении, что объем накладных расходов на управление этими хостами будет увеличиваться. И это действительно так, если все делать вручную. Удваивается количество серверов, удваивается и объем работы! Но если автоматизировать управление хостами и развертывание сервисов, тогда не станет причин для линейного роста объема рабочей нагрузки в зависимости от количества добавляемых хостов.

Но даже при сохранении небольшого количества хостов мы все же собираемся манипулировать большим количеством сервисов. Это означает, что нужно справляться с несколькими развертываниями, отслеживать работу нескольких сервисов и собирать регистрируемые данные. И автоматизация в этом деле играет весьма важную роль.

Автоматизация может обеспечить также поддержание высокой продуктивности разработчиков. Ключом для облегчения жизни разработчиков является предоставление им возможности обеспечения самообслуживания отдельных сервисов и групп сервисов. В идеале разработчики должны иметь доступ абсолютно к такой же цепочке инструментов, которая используется для развертывания сервисов в производственном режиме, чтобы можно было обнаруживать проблемы на самых ранних стадиях. В данной главе будут рассмотрены многие технологии, включающие это представление.

Выбор технологии автоматизации играет весьма важную роль. Он начинается со средств управления хостами. Можете ли вы написать строку кода для запуска виртуальной машины или ее выключения? Можете ли провести автоматическое развертывание написанной вами программы? Можете ли развернуть изменения, внесенные в базу данных без ручного вмешательства? Если вы хотите справиться

со сложностями архитектур микросервисов, то без усвоения культуры автоматизации вам не обойтись.

Два конкретных примера изучения эффективности автоматизации. Наверное, объяснить эффективность качественной автоматизации полезнее всего на двух конкретных примерах. Одним из наших клиентов является компания RealEstate.com.au (REA) из Австралии. Кроме всего прочего, она предоставляет объекты недвижимости для розничной торговли и коммерческих клиентов в Австралии и в других местах Азиатско-Тихоокеанского региона. В течение ряда лет компания переводит свою платформу на распределенную микросервисную модель. В самом начале этого пути ей пришлось потратить много времени на получение подходящих для сервисов инструментов, облегчающих разработчикам предоставление машин, развертывание их кода или отслеживание его работы. Для запуска проекта пришлось сконцентрировать все усилия с самого начала.

За первые три месяца REA смогла перевести в производственный режим использования только два новых микросервиса, при этом команда разработчиков взяла на себя полную ответственность за их сборку, развертывание и поддержку. В следующие три месяца удалось запустить в том же режиме еще 10–15 сервисов. К концу 18-месячного периода в REA работали свыше 70 сервисов.

Действенность подобной схемы подтверждается и опытом компании Gilt, с 2007 года занимающейся продажей через Интернет модной одежды. Использовавшееся в Gilt монолитное Rails-приложение стало испытывать трудности с масштабированием, и в 2009 году компания решила приступить к декомпозиции системы на микросервисы. И опять автоматизация, особенно оснащенная инструментарием для помощи разработчикам, стала ключевым обоснованием активного перехода компании Gilt к использованию микросервисов. Год спустя у Gilt работали десять микросервисов, к 2012-му — уже свыше 100, а в 2014 году, по собственному подсчету Gilt, — свыше 450 микросервисов. Иными словами, на каждого разработчика в Gilt приходилось примерно по три сервиса.

От физического к виртуальному

Одним из ключевых инструментов, доступных нам при управлении большим количеством хостов, является поиск способов разбиения существующих физических машин на более мелкие части. Огромные преимущества в сокращении издержек на управление хостами могут быть получены с помощью таких традиционных средств виртуализации, как VMWare, или с помощью использования AWS. Но в данной области есть и новые достижения, которые стоит исследовать, поскольку они могут открыть еще более интересные возможности для работы с нашей архитектурой микросервисов.

Традиционная виртуализация

Почему использование большого количества хостов обходится так дорого? Когда для каждого хоста нужен физический сервер, ответ вполне очевиден. Если вы работаете именно в такой обстановке, то вам, наверное, подойдет модель размещения

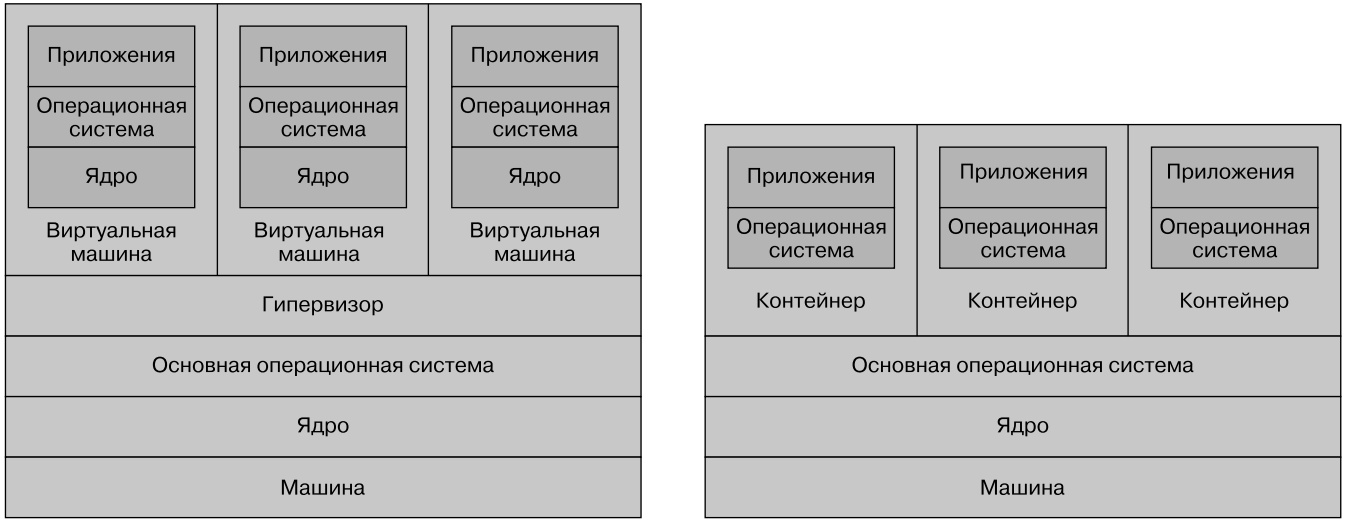
нескольких сервисов на одном хосте, хотя не удивлюсь, если это станет еще более сложным препятствием. Но я подозреваю, что большинство из вас использует какую-либо виртуализацию. Виртуализация позволяет разбить физический сервер на несколько обособленных хостов, на каждом из которых могут запускаться разные программные средства. Следовательно, если нам нужно разместить по одному сервису на каждом хосте, можем ли мы просто разбить физическую инфраструктуру на все более мелкие и мелкие куски?

Ну кто-то это, вероятно, и может сделать. Но разбиение машины на постоянно увеличивающееся количество виртуальных машин не дается бесплатно. Представим нашу физическую машину ящиком для носков. Если поставить в ящике много деревянных перегородок, то больше или меньше носков мы сможем в нем хранить? Правильный ответ — меньше: разделители также занимают место! Нужно, чтобы с нашим ящиком было проще работать и проще наводить в нем порядок, и, возможно, теперь мы решим положить в него не только носки, но еще и футболки, но чем больше разделителей, тем меньше будет общее пространство.

В мире виртуализации у нас есть такие же издержки, как и перегородки в ящике для носков. Чтобы понять природу возникновения этих издержек, посмотрим, как чаще всего выполняется виртуализация. На рис. 6.9 для сравнения представлены два типа виртуализации. Слева показано использование нескольких различных уровней так называемой виртуализации второго типа, которая реализуется в AWS, VMWare, VSphere, Xen и KVM. (К виртуализации первого типа относятся технологии, при которых виртуальные машины запускаются непосредственно на оборудовании, а не в виде надстройки над другой операционной системой.) В нашей физической инфраструктуре имеется основная операционная система. Под управлением этой операционной системы запускается так называемый гипервизор, имеющий две основные задачи. Во-первых, он отображает ресурсы, подобные ресурсам центрального процессора и памяти, с виртуального хоста на физический хост. Во-вторых, он работает в качестве уровня управления, позволяющего манипулировать самими виртуальными машинами.

Внутри виртуальных машин мы получаем то, что похоже на совершенно различные хосты. На них можно запустить их собственные операционные системы с их собственными ядрами. Их можно рассматривать почти как загерметизированные машины, изолированные гипервизором от используемого физического хоста и от других виртуальных машин.

Проблема в том, что для выполнения своей работы гипервизору нужно выделять ресурсы. Это отнимает ресурсы центрального процессора, системы ввода-вывода и памяти, которые могли бы использоваться в других местах. Чем большим количеством хостов управляет гипервизор, тем больше ресурсов ему требуется. К определенному моменту эти издержки станут препятствовать дальнейшему разбиению физической инфраструктуры. На практике это означает, что часто при нарезке физической машины на все более мелкие части убывает отдача ресурсов, поскольку пропорционально этому все больше и больше этих ресурсов уходит на издержки гипервизора.



Стандартные виртуализации

Виртуализации на основе контейнеров (LXC)

Рис. 6.9. Сравнение стандартной виртуализации второго типа и облегченных контейнеров

Vagrant

Vagrant представляет собой весьма полезную платформу развертывания, которая используется скорее для разработки и тестирования, чем для работы в производственном режиме. Vagrant предоставляет вам на вашем ноутбуке виртуальное облако. Снизу она использует стандартную систему виртуализации (обычно VirtualBox, хотя может использовать и другие платформы). Она позволяет определять набор виртуальных машин в текстовом файле, а также определять, как виртуальные машины связаны по сети и на каких образах они должны быть основаны. Этот тестовый файл может проверяться и совместно использоваться сотрудниками команды.

Это упрощает для вас создание среды подобной той, что используется в производственном режиме на вашей локальной машине. Одновременно можно запускать несколько виртуальных машин, останавливать работу отдельных машин для тестирования сбойных режимов и отображать виртуальные машины на локальные каталоги, чтобы можно было вносить изменения и тут же отслеживать реакцию на них. Даже для тех команд, которые используют такие облачные платформы, выделяющие ресурсы по запросу, как AWS, более быстрая реакция от применения Vagrant может стать большим подспорьем в разработке.

Один из недостатков заключается в том, что запуск большого количества виртуальных машин может привести к перегрузке разработочной машины. Если у нас на каждой виртуальной машине имеется по одному сервису, то может исчезнуть возможность переноса всей системы на свою локальную машину. Это может привести к тому, что вам, чтобы добиться управляемости, придется ставить заглушки на некоторые зависимости, и это станет еще одной задачей, с которой нужно будет справиться, чтобы обеспечить качественное проведение развертывания и тестирования.

Контейнеры Linux

У пользователей Linux имеется альтернатива виртуализации. Вместо использования гипервизора для сегментирования и управления отдельными виртуальными хостами Linux-контейнеры создают обособленное пространство для процессов, в котором проходят остальные процессы.

В Linux процесс запускается конкретным пользователем и имеет конкретные возможности, основанные на особенностях настройки прав пользователей. Процессы могут порождать другие процессы. Например, если я запущу процесс в терминале, то этот процесс обычно считается дочерним процессом процесса терминала. Обслуживанием дерева процессов занимается ядро Linux.

Linux-контейнеры являются расширением этого замысла. Каждый контейнер, по сути, является поддеревом общесистемного дерева процессов. У этих контейнеров могут быть выделенные им физические ресурсы, и этим ядро управляет для нас. Этот общий подход был воплощен во многих формах, таких как Solaris Zones и OpenVZ, но наибольшую популярность снискал LXC-контейнер. LXC в готовом виде доступен во многих современных ядрах Linux.

Если посмотреть на блок-схему для хоста с работающим LXC-контейнером (см. рис. 6.9), можно увидеть несколько различий. Во-первых, нам не нужен гипер-

визор. Во-вторых, хотя каждый контейнер может запускать собственный дистрибутив операционной системы, он должен совместно использовать одно и то же ядро (поскольку процесс дерева запускается именно в ядре). Это означает, что в качестве основной операционной системы может быть запущена Ubuntu, а в контейнерах — CentOS, поскольку обе они могут совместно использовать одно и то же ядро.

Но мы не только получаем преимущества от ресурсов, сэкономленных за счет ненужности гипервизора. Мы также выигрываем из-за ускоренной обратной реакции. Linux-контейнеры предоставляются намного быстрее, чем полноценные виртуальные машины. Для виртуальных машин вполне в порядке вещей тратить несколько минут на запуск, а при использовании Linux-контейнеров запуск может занять всего несколько секунд. Вы приобретаете также более тонкий контроль над самими контейнерами в плане выделения им ресурсов, что существенно упрощает тонкую подстройку под получение большей отдачи от использующегося оборудования.

Благодаря более легким характеристикам контейнеров мы можем получить от них намного большую отдачу при запуске на одном и том же оборудовании, которую возможно было бы получить при использовании виртуальных машин. За счет развертывания по одному сервису в каждом контейнере (рис. 6.10) мы получаем определенную степень изолированности от других контейнеров (хотя ее нельзя назвать идеальной) и можем получить более высокий экономический эффект, чем тот, который был бы возможен, если бы нам захотелось запустить каждый сервис на его собственной виртуальной машине.

Контейнеры могут использоваться также с полноценной виртуализацией. Мне приходилось видеть не один проект с предоставлением довольно крупного экземпляра AWS EC2 и запуском на нем LXC-контейнеров для получения наилучших качеств, присущих обоим мирам: предоставляемую по запросу эфемерную вычислительную платформу в виде EC2 в сочетании с весьма гибкими и быстрыми контейнерами, запущенными на ее основе.

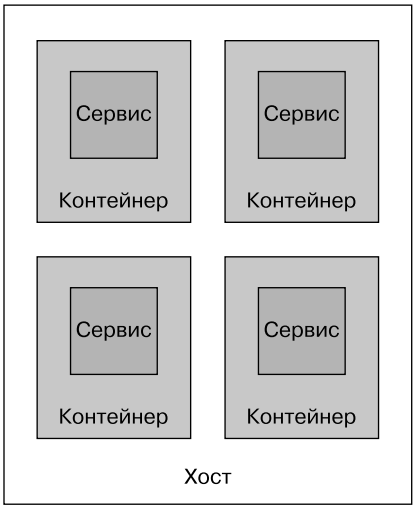


Рис. 6.10. Запуск сервисов в отдельных контейнерах

Но у Linux-контейнеров есть и свои проблемы. Представьте, что у меня имеется масса микросервисов, запущенных в их собственных контейнерах на хосте. Как они станут видны внешнему миру? Нужен некий способ, направляющий внешний мир через используемые контейнеры, то есть то, чем при обычной виртуализации занимаются многие гипервизоры. Я видел, как многие тратили уйму времени на настройку перенаправления портов с использованием для непосредственного показа контейнеров таблиц IPTables. Кроме этого, следует принять во внимание, что эти контейнеры нельзя рассматривать как абсолютно герметичные по отношению друг к другу. Существует множество задокументированных и известных способов, позволяющих процессу из одного контейнера сбежать и вступить во взаимодействие с другими контейнерами или с используемым хостом. Некоторые из этих проблем вызваны конструктивными особенностями, а часть из них связана с недочетами, находящимися в процессе устранения, но в любом случае, если не испытывается доверие к запускаемому коду, не ждите, что сможете запустить его в контейнере и при этом остаться в безопасности. Если нужна повышенная изолированность, то следует присмотреться к использованию виртуальных машин.

Docker

Docker представляет собой платформу, являющуюся надстройкой над облегченными контейнерами. Она вместо вас справляется с большим объемом работы по управлению контейнерами. В Docker ведется создание и развертывание приложений, что для мира виртуальных машин является синонимом образов, хотя и на платформе, основанной на применении контейнеров. Docker управляет предоставлением контейнеров, справляется за вас с некоторыми проблемами использования сетей и даже предоставляет собственное понятие реестра, позволяющее хранить сведения о Docker-приложениях и фиксировать их версии.

Абстракция Docker-приложений для нас полезна, поскольку точно так же, как и с образами виртуальных машин, основная технология, используемая для реализации сервиса, скрыта от нас. У нас имеются наши сборки для сервисов, создающие Docker-приложения и сохраняющие их в Docker-реестре, и больше ничто нас не интересует.

Docker может также смягчить некоторые недостатки от локального запуска множества сервисов в целях разработки и тестирования. Вместо использования Vagrant для размещения нескольких независимых виртуальных машин, в каждой из которых содержится ее собственный сервис, мы можем разместить одну виртуальную машину в Vagrant, на которой выполняется экземпляр Docker-платформы. Затем Vagrant используется для установки и удаления самой платформы Docker и использования Docker для быстрого предоставления отдельных сервисов.

Для получения преимуществ от использования платформы Docker были разработаны несколько технологий. С прицелом на использование Docker была разработана очень интересная операционная система CoreOS. Это урезанная до минимума операционная система Linux, предоставляющая только важнейшие службы, позволяющие работать платформе Docker. Это означает, что она потребляет меньше ресурсов, чем другие операционные системы, позволяя выделять используемой машине еще больше ресурсов для наших контейнеров. Вместо использования диспетчера пакетов, подоб-

ного deb- или RPM-диспетчерам, все программы устанавливаются как независимые Docker-приложения, каждое из которых запускается в собственном контейнере.

Сам Docker всех проблем за нас не решает. Его нужно представлять в качестве простой PaaS-платформы, работающей на одной машине. Если требуется инструментарий, помогающий управлять сервисами, разбросанными по нескольким Docker-экземплярам и нескольким машинам, нужно поискать другие программы, добавляющие такие возможности. К основным потребностям относится уровень диспетчеризации, позволяющий запрашивать контейнер, после чего он находит Docker-контейнер, который может запустить для вас. Оказать помощь в этой области могут принадлежащая Google недавно разработанная технология с открытым кодом Kubernetes и имеющиеся в CoreOS кластерные технологии. И похоже, что новички в этой области появляются чуть ли не каждый месяц. Еще одним интересным средством на основе Docker является Deis. Оно пытается предоставить поверх Docker PaaS-платформу, похожую на Heroku.

Ранее я уже говорил о PaaS-решениях. Я всегда вел борьбу с ними из-за того, что они довольно часто неправильно понимают уровень абстракции, а самостоятельно принимаемые решения существенно отстают от таких уже принятых решений, как Heroku. Docker получает намного больше прав, и резкий подъем интереса в этой области означает, как я предполагаю, что это средство через несколько лет станет намного более жизнеспособной платформой для всех разновидностей развертываний и всех разнообразных вариантов применения. Во многих отношениях Docker с соответствующим уровнем диспетчеризации располагается между решениями IaaS и PaaS, и для его описания уже использовался термин «контейнеры в качестве сервиса» (Containers as a Service (CaaS)).

Docker используют в производственном режиме несколько компаний. Он совместно со средствами, помогающими исключить многие недостатки, обеспечивает множество преимуществ, присущих облегченным контейнерам, в показателях эффективности и скорости предоставления контейнеров. Если вас интересуют альтернативные платформы развертывания, то я настоятельно советую присмотреться к Docker.

Интерфейс развертывания

Независимо от используемой платформы и артефактов наличие унифицированного интерфейса для развертывания исходного сервиса жизненно необходимо. Нам потребуется запускать развертывание микросервиса по мере необходимости в разнообразных ситуациях, от развертывания на локальной машине для разработки и тестирования до развертывания для работы в производственном режиме. И от разработки до производства нам нужно поддерживать как можно более однообразный механизм развертывания, поскольку крайне нежелательно столкнуться с проблемами в производстве из-за того, что при развертывании используется совершенно иной процесс!

Проработав в этой области много лет, я убежден, что наиболее разумным способом запуска любого развертывания является единый вызов инструкции командной строки с указанием нужных параметров. Инструкция может вызываться из

сценария, ее выполнение может инициироваться вашим CI-средством, или же она может набираться вручную. Для выполнения этой работы я создавал сценарии-оболочки в различных технологических стеках, от пакетных файлов Windows до bash, а также до Python Fabric-сценариев и т. д., но все инструкции командной строки использовали один и тот же основной формат.

Нам нужно знать, что именно мы развертываем, следовательно, нужно предоставить имя известного объекта или же в данном случае микросервиса. Нам также нужно знать, какая версия объекта нами востребована. Ответ на вопрос о версии, скорее всего, будет одним из трех возможных. При работе на локальной машине это будет версия для работы на этой машине. При тестировании понадобится самая последняя зеленая сборка, которая может быть просто самым последним подходящим артефактом в нашем хранилище артефактов. А при тестировании или диагностировании проблем может потребоваться развернуть конкретную сборку.

Третье и последнее, о чем нам нужно знать, — это в какой среде следует развертывать микросервис. Как уже говорилось, топология наших микросервисов от среды к среде может быть разной, но здесь она должна быть скрыта от нас.

Итак, представим, что мы создали простой сценарий развертывания, получающий три параметра. Скажем, проводится локальное развертывание сервиса каталога в нашу локальную среду. Я могу набрать следующую инструкцию:

```
$ deploy artifact=catalog environment=local version=local
```

После того как будет проведена проверка, CI-сборка сервиса получает изменение и создает новый сборочный артефакт, присваивая сборке номер b456 в соответствии со стандартом, принятым в большинстве CI-средств. Это значение проводится по всему конвейеру. Когда запускается наша стадия тестирования, при выполнении CI-стадии может быть написано следующее:

```
$ deploy artifact=catalog environment=ci version=b456
```

Тем временем контролю качества потребовалось поместить последнюю версию сервиса каталога в интегрированную тестовую среду для выполнения исследовательского тестирования и оказания помощи в оформлении проспекта. Эта команда написала следующую инструкцию:

```
$ deploy artifact=catalog environment=integrated_qa version=latest
```

Чаще всего для этого я пользуюсь Fabric, библиотекой языка Python, предназначенной для отображения вызовов командной строки на функции, наряду с такой хорошей поддержкой задач обработки на удаленных машинах, как SSH. Соедините это с такой клиентской библиотекой AWS, как Boto, и у вас будет все необходимое для полной автоматизации очень крупных сред AWS. Для Ruby в некотором роде аналогом Fabric может послужить Capistrano, а для Windows можно воспользоваться PowerShell.

Определение среды. Разумеется, чтобы все это заработало, нужен способ определения того, на что похожа наша среда и на что похож наш сервис в заданной среде. Определение среды можно представить себе как отображение микросервиса на вычислительные ресурсы, сетевые ресурсы или ресурсы хранилища. Раньше я де-

лал это с помощью YAML-файлов и для помещения в них данных использовал свои сценарии. В примере 6.1 показана упрощенная версия той работы, которую я делал года два назад для проекта, использующего AWS.

Пример 6.1. Пример определения среды

```
development:
    nodes:
    - ami_id: ami-e1e1234
      size: t1.micro (1)
      credentials_name: eu-west-ssh (2)
      services: [catalog-service]
      region: eu-west-1

production:
    nodes:
    - ami_id: ami-e1e1234
      size: m3.xlarge (1)
      credentials_name: prod-credentials (2)
      services: [catalog-service]
      number: 5 (3)
```

1. Размер используемых экземпляров варьируется с целью получения более высокого экономического эффекта. Для исследовательского тестирования не нужна 16-ядерная стойка с 64 Гбайт оперативной памяти!
2. Ключевой является возможность указать различные полномочия для разных сред. Полномочия для конфиденциальных сред хранились в различных репозиториях исходного кода, и к ним могли получать доступ только избранные.
3. Мы решили, что по умолчанию, если у сервиса имелось более одного сконфигурированного узла, для него будет автоматически создаваться система обеспечения сбалансированности нагрузки.

   Некоторые детали для краткости были удалены.

   Информация о сервисе каталога была сохранена в другом месте. Как видно из примера 6.2, от среды к среде она не изменяется.

   Пример 6.2. Пример определения среды

   ```
   catalog-service:
       puppet_manifest : catalog.pp (1)
       connectivity:
         - protocol: tcp
           ports: [ 8080, 8081 ]
           allowed: [ WORLD ]
   ```

4. Это было имя запускаемого Puppet-файла. Так уж получилось, что в данной ситуации мы использовали только Puppet, но теоретически могли бы воспользоваться и поддержкой альтернативных систем управления настройками.

Вполне очевидно, что многие моменты поведения были основаны на соглашениях. Например, мы решили нормализовать используемые сервисом порты, где бы он ни запускался, и автоматически настроили систему обеспечения сбалансированности

нагрузки в том случае, если у сервиса имеется более одного экземпляра (в AWS с этим довольно легко справляется ELB).

Создание систем, подобных этой, требует существенного объема работы. Усилия зачастую нужно прикладывать на начальном этапе, но они могут иметь большое значение для управления возникающими сложностями развертывания. Я надеюсь, что в будущем вам не придется делать это самим. Уже есть новейшее средство под названием Terraform от Hashicorp, которое работает именно в этой области. Я, как правило, уклоняюсь от упоминания в книге подобных новых средств, поскольку она посвящена скорее идеям, а не технологиям, но попытки создания средств с открытым кодом, работающих в данном направлении, предпринимаются. Хотя говорить об этом еще слишком рано, но уже существующие возможности представляются весьма интересными. Имея возможность нацеливания развертываний на различные платформы, в будущем вы можете сделать это средство своим рабочим инструментом.

Резюме

В данной главе было рассмотрено множество вопросов, о которых я хочу напомнить по порядку. Сначала мы сконцентрировали внимание на поддержании способности выпуска одного сервиса независимо от другого и на гарантиях того, чтобы она сохранялась независимо от выбираемой технологии. Я предпочитаю использовать по одному хранилищу для каждого микросервиса, но, если вы хотите развертывать микросервисы независимо друг от друга, я еще больше убежден в том, что вам нужна одна CI-сборка на каждый микросервис.

Затем, если есть такая возможность, перейдите к схеме, предполагающей размещение одного сервиса на каждом хосте или в каждом контейнере. Чтобы удешевить и облегчить управление перемещаемыми частями, обратите внимание на альтернативные технологии, такие как LXC или Docker, но при этом отнеситесь с пониманием к тому, что, какую бы технологию вы ни взяли на вооружение, ключом к управлению всеми аспектами служит культура автоматизации. Автоматизируйте все, что можно, и если используемая вами технология не позволяет этого сделать, перейдите на новую технологию! Когда дело дойдет до автоматизации, огромные преимущества вам даст возможность использования платформы, подобной AWS.

Удостоверьтесь в том, что вы усвоили степень влияния сделанного вами выбора, связанного с разработкой, на самих разработчиков, а также в том, что им тоже нравится ваш выбор. Особую важность имеет создание средств, предназначенных для самостоятельного развертывания любого конкретного сервиса в нескольких различных средах. Эти средства помогут и тем, кто занимается разработкой, тестированием и эксплуатацией сервисов.

И наконец, если вы хотите изучить эту тему глубже, я настоятельно рекомендую прочитать книгу Джеза Хамбла (Jez Humble) и Дэвида Фарли (David Farley) *Continuous Delivery* (Addison-Wesley), в которой намного больше подробностей, касающихся конструирования конвейеров и управления артефактами.

В следующей главе мы углубимся в тему, которой уже коснулись в данной главе. А именно, как проводить тестирование микросервисов, чтобы убедиться в их работоспособности.

7 Тестирование

С тех пор как я только начал создавать программный код, автоматизированное тестирование существенно усовершенствовалось. Похоже, что новое средство или технология в этой области появляются чуть ли не каждый месяц, делая такое тестирование еще лучше. Но и когда функциональность распространена по всей распределенной системе, проблемы выполнения эффективного и рационального тестирования все еще остаются нерешенными. В этой главе разбираются проблемы, связанные с тестированием систем с высокой степенью детализации, и дается ряд решений, помогающих убедиться в возможности выпуска созданных вами новых функциональных возможностей, будучи уверенными в их работоспособности.

Тестирование охватывает множество понятий. Даже если говорить только об автоматизированных тестах, возникает большое количество требующих рассмотрения вопросов. С появлением микросервисов добавился еще один уровень сложности. Понимание того, какого рода тесты можно запускать, играет важную роль в содействии установлению баланса между порой противоречащими друг другу устремлениями, чтобы как можно быстрее довести программные продукты до работы в производственном режиме, помимо того чтобы просто удостовериться в их надлежащем качестве.

Разновидности тестов

Мне, как консультанту, нравится способ распределения по категориям путем деления на секторы, и я уже начал переживать, что в этой книге его так и не придется применить. Но так уж удачно вышло, что Брайан Марик (Brian Marick) предоставил нам великолепную систему распределения тестов по категориям. На рис. 7.1 показан помогающий определить категории различных тестов вариант секторов, выделенных Мариком, который был позаимствован из книги Лизы Криспин (Lisa Crispin) и Джанет Грегори (Janet Gregory) *Agile Testing* (Addison-Wesley).

В нижней части показаны тесты технологической направленности, то есть в первую очередь помогающие разработчикам в создании системы. В эту категорию попадают тесты производительности и имеющие весьма ограниченную область действия блочные тесты, которые, как правило, автоматизированы. Это описание дается в сравнении с секторами верхней части, где показаны тесты, помогающие понять характеристики работы вашей системы тем партнерам, которые не связаны

с техническими сторонами разработки. Это могут быть сквозные тесты с весьма широкой областью действия, такие как показанное в верхнем левом секторе приемо-сдаточное тестирование или помещенное в сектор исследовательского тестирования ручное проведение тестов, представляющее собой типичное пользовательское тестирование, выполняемое с целью проверки приемлемости системы для пользователя.

Рис. 7.1. Секторы тестирования по Брайану Марику. Lisa Crispin, Janet Gregory. Agile Testing: A Practical Guide for Testers and Agile Teams, 1st Ed. © 2009. С разрешения Pearson Education, Inc., Upper Saddle River, NJ

Каждой разновидности тестов, показанной в этих секторах, отводится своя роль. В каком конкретном объеме требуется проводить каждый тест именно вам, зависит от характера вашей системы, но здесь важно понять, что для ее тестирования у вас есть богатый выбор. В последнее время наметилась тенденция отказа от ручного тестирования с широкой областью действия и преимущественного перехода по мере возможности к автоматизированному тестированию, и я с таким подходом, конечно же, согласен. Если вы все еще применяете большие объемы ручного тестирования, то прежде, чем всерьез заняться микросервисами, последуйте моему совету и обратите внимание на эту тенденцию, поскольку вы не сможете получить многие из предполагаемых преимуществ микросервисов, если не сумеете проверять свои программные продукты быстро и эффективно.

Чтобы выполнить те задачи, которые ставятся в этой главе, ручное тестирование мы просто проигнорируем. Хотя тестирование этого рода может принести немалую пользу и, безусловно, играет свою собственную и весьма важную роль, разница в тестировании архитектуры микросервисов в основном проявляется в контексте различных типов автоматизированных тестов, поэтому ими мы и займемся.

Но если дело доходит до автоматизированных тестов, то в каких объемах нам нужно проводить каждое тестирование? Чтобы ответить на данный вопрос и разобраться с возможными разнообразными компромиссами, пригодится еще одна модель.

Области применения тестов

В своей книге *Succeeding with Agile* (Addison-Wesley) Майк Кон, чтобы помочь разобраться в том, какие типы автоматизированных тестов вам нужны, начертил модель под названием «Пирамида тестов». Эта пирамида помогает нам обдумать область применения тестов, а также очерчивает пропорции тестов различного типа, которых мы должны придерживаться. Как показано на рис. 7.2, в исходной модели Кона автоматизированные тесты разбиты на блочные, сервисные и тесты пользовательского интерфейса.

Рис. 7.2. Пирамида тестов Майка Кона из его книги Succeeding with Agile: Software Development Using Scrum, 1st Ed. © 2010. С разрешения Pearson Education, Inc., Upper Saddle River, NJ

Проблема данной модели в том, что все эти понятия не имеют однозначного толкования. Особенно много значений имеет понятие «сервис», достаточно определений имеется и у понятия «блочное тестирование». Можно ли считать блочным тест, проверяющий только одну строку кода? Я бы сказал, что да. А можно ли считать блочным тест нескольких функций или классов? А тут я бы сказал, что нет, но многие со мной не согласятся! Несмотря на терминологическую неоднозначность, я все же склоняюсь к выбору названий «блочное тестирование» и «тестирование сервиса», но тесты пользовательского интерфейса предпочитаю называть сквозными тестами, именно так они далее и будут называться.

Учитывая возникшую неразбериху, нам все же стоит разобраться в том, что означают все эти различные уровни.

Рассмотрим рабочий пример. На рис. 7.3 показаны приложение сервиса технической поддержки и наш основной сайт, и оба эти компонента взаимодействуют с клиентским сервисом для извлечения, просмотра и редактирования клиентских данных. В свою очередь, клиентский сервис связывается с нашим банком бонусных баллов, где клиенты накапливают баллы, покупая компакт-диски Джастина Бибера. Наверное. Конечно, это всего лишь малая часть общей системы музыкального магазина, но и ее вполне хватает, чтобы разобрать несколько различных сценариев, которые нам может потребоваться протестировать.

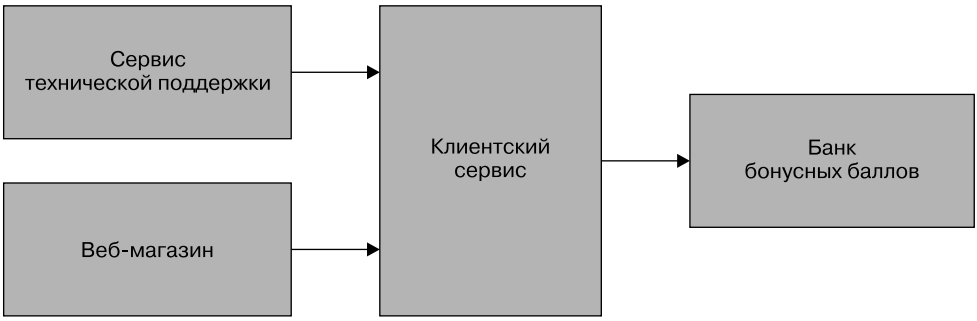


Рис. 7.3. Часть музыкального магазина, подвергаемая тестированию

Блочные тесты

Это тесты, с помощью которых тестируется, как правило, отдельный вызов функции или метода. Под эту категорию подпадают тесты, созданные в рамках концепции разработки под контролем тестирования (Test-Driven Design (TDD)), а также разновидности тестов, созданных с помощью такой технологии, как тестирование на основе свойств (Property-Based Testing). Сам сервис здесь не запускается, мы также ограничены в использовании внешних файлов или сетевых подключений. По сути, тестов такого рода требуется очень много. Если они правильно сделаны, то выполняются очень и очень быстро и можно рассчитывать на выполнение на современном оборудовании многих тысяч таких тестов меньше чем за минуту.

Это тесты, помогающие нам, разработчикам, и поэтому в соответствии с терминологией, предложенной Мариком, должны иметь технологическую, а не бизнес-направленность. В ходе их проведения мы надеемся отловить основную часть своих ошибок. Итак, в нашем примере, когда мы занимаемся клиентским сервисом, блочные тесты будут охватывать небольшую изолированную часть кода (рис. 7.4).

Основной целью этих тестов является получение очень быстрых ответных результатов, говорящих о качестве функционирования кода. Тесты могут играть весьма важную роль в поддержке разбиения кода на части, позволяя проводить реструктуризацию кода по мере выполнения работы. При этом мы будем знать, что имеющие весьма ограниченную область действия тесты тут же нас остановят, если будет допущена ошибка.

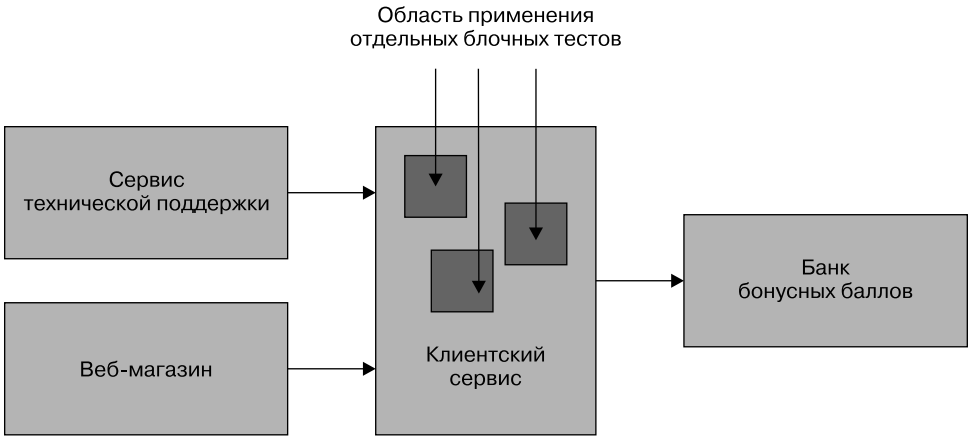


Рис. 7.4. Область применения блочных тестов в нашей взятой для примера системе

Тесты сервиса

Тесты сервиса разрабатываются для того, чтобы в обход пользовательского интерфейса выполнять непосредственное тестирование сервиса. В монолитном приложении могут тестироваться коллекции классов, предоставляющие сервис пользовательскому интерфейсу. В системе, содержащей несколько сервисов, тест сервиса используется для тестирования возможностей отдельного сервиса.

Причина, по которой требуется протестировать отдельно взятый сервис, состоит в улучшении изолированности теста с целью более быстрого обнаружения и устранения проблем. Для достижения изолированности нужно заглушить все внешние сотрудничающие компоненты, чтобы в область действия теста попадал только сам сервис (рис. 7.5).

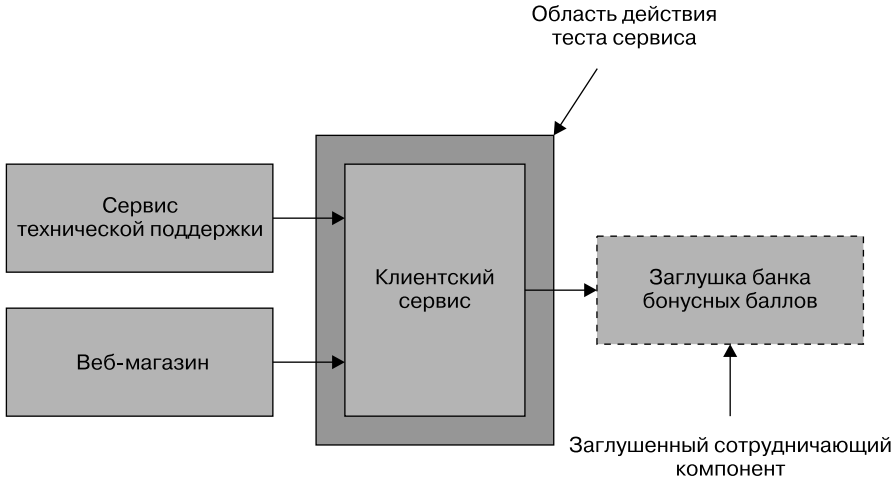


Рис. 7.5. Область применения тестов сервиса в нашей взятой для примера системе

Некоторые из этих тестов могут выполняться так же быстро, как и небольшие тесты, но, если задумать тестирование с участием реальной базы данных или с переходом по сети к заглушенным нижестоящим сотрудничающим компонентам, время проведения теста может увеличиться. Кроме того, эти тесты имеют более широкую область действия, чем простой блочный тест, поэтому, если тест не будет пройден, причину окажется найти труднее, чем при проведении блочного теста. Тем не менее в их область действия попадает гораздо меньше активных компонентов, чем при проведении широкомасштабного тестирования, поэтому проходят проще.

Сквозные тесты

Сквозные тесты проводятся в отношении всей системы. Зачастую управляют ими через графический интерфейс пользователя, имеющийся у браузера, но с возможностью без каких-либо затруднений имитировать другие виды взаимодействия с пользователем, например выкладывание файла.

Как показано на рис. 7.6, этими тестами охвачен большой объем кода, предназначенного для работы в производственном режиме. Следовательно, при прохождении этих тестов возникает удовлетворенность: с большой долей уверенности можно сказать, что протестированный код будет работать в производственном режиме. Но этой увеличенной области действия присущи некоторые недостатки, и, как мы вскоре увидим, в контексте микросервисов они могут иметь весьма запутанный характер.

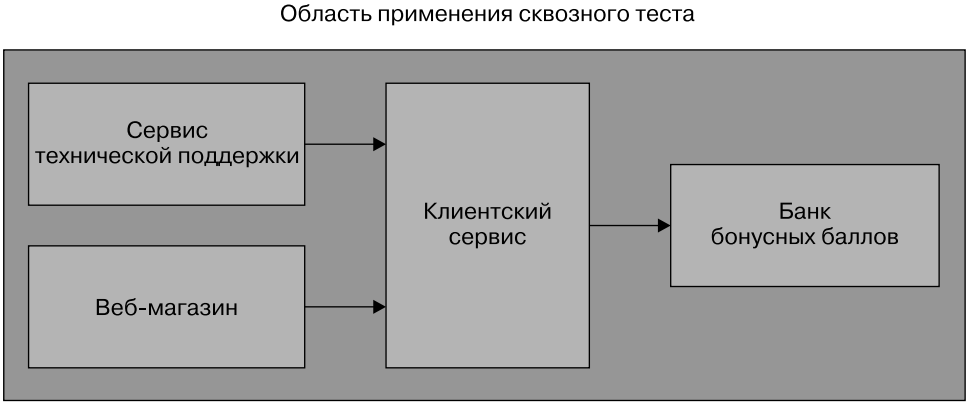


Рис. 7.6. Область применения сквозных тестов в нашей взятой для примера системе

Компромиссы

При изучении пирамиды ключевым моментом, который требуется усвоить, является то, что по мере движения к ее вершине область действия тестов увеличивается вместе с уверенностью в том, что протестированные функциональные возможности окажутся вполне работоспособными. Однако время ожидания отдачи

от тестов увеличивается, поскольку времени на их выполнение уходит больше, и когда тест дает сбой, установить отказавшую функцию может быть намного сложнее. При движении сверху вниз по направлению к основанию пирамиды тесты выполняются, как правило, намного быстрее, поэтому мы получаем намного более быстрые циклы обратной реакции. Быстрее обнаруживаются отказавшие функции, быстрее создаются сборки непрерывной интеграции, и уменьшается вероятность того, что мы перейдем к другой задаче, прежде чем обнаружится, что мы что-нибудь вывели из строя. Когда сбой происходит при проведении таких имеющих весьма ограниченную область действия тестов, мы стремимся обнаружить место сбоя, зачастую вплоть до отдельной строки кода. В то же время, если тестируется отдельная строка кода, мы не получаем никакой уверенности в работоспособности системы в целом!

Когда сбой происходит при проведении тестов с широкой областью действия вроде тестов сервиса или сквозных тестов, мы стараемся написать быстрый блочный тест, для того чтобы в будущем с его помощью определить причину возникшей проблемы. Таким образом, мы постоянно пытаемся сократить время циклов обратной реакции.

Фактически каждая команда, с которой мне приходилось работать, по-разному называла то, что Кон использует в своей пирамиде. Но как бы все это ни называть, главное — понять, что вам понадобятся различные тесты, проводимые с разными целями.

Что и в каком объеме проводить

Итак, если при проведении всех этих тестов не обойтись без компромиссов, то в каком объеме понадобится проведение каждого из них? Опыт показывает, что по мере продвижения по пирамиде сверху вниз понадобится, наверное, на порядок больше тестов, но при этом важно знать, что в вашем распоряжении имеется множество различных автоматизированных тестов, и чувствовать момент, когда текущий баланс того и другого превращается в серьезную проблему!

К примеру, мне приходилось участвовать в разработке одной монолитной системы, где у нас было 4000 блочных тестов, 1000 тестов сервисов и 60 сквозных тестов. Мы решили, что с точки зрения получения ответных результатов у нас было слишком много тестов сервисов и сквозных тестов (и последние были наиболее критикуемыми в плане продолжительности циклов получения ответов), поэтому упорно работали над переходом к тестам, имеющим менее широкую область действия.

Известным антишаблоном является то, что часто называют тестированием рожка с мороженым, или перевернутой пирамидой. Этот неверный подход характерен практическим отсутствием тестов с ограниченной областью действия, и все сводится к тестам, имеющим широкую область действия. Такие проекты зачастую характеризуются «замороженным» выполнением тестов и очень длинными циклами получения ответных результатов. Если такие тесты запускаются в рамках непрерывной интеграции, то количество получаемых сборок не будет большим и сроки, характерные для создания сборок, будут означать, что сами сборки в случае каких-либо сбоев могут находиться в нерабочем состоянии на протяжении весьма длительного времени.

Реализация тестов сервисов

В целом реализация блочных тестов дается намного проще всего остального, и существует множество различных документов, описывающих порядок их создания. Нам же намного интереснее будет разобраться в создании сервисных и сквозных тестов.

Тесты сервисов предназначены для тестирования той доли функциональности, которая распространяется на весь сервис, но, для того, чтобы изолироваться от других сервисов, нужно найти некий способ, позволяющий заглушить всех соучастников процесса. Следовательно, если нужно написать подобный тест для клиентского сервиса, показанного на рис. 7.3, следует развернуть экземпляр клиентского сервиса и, как уже говорилось, заглушить все взаимодействующие сервисы.

Первое, что нужно сделать при изготовлении сборки в рамках непрерывной интеграции, — создать для сервиса артефакт двоичного кода, поскольку его развертывание проводится довольно просто. Но как справиться с подделкой работы взаимодействующих сервисов?

Сервисный тестовый набор нуждается в запуске сервисов-заглушек для любого взаимодействующего с ним участника (или в гарантии того, что они запущены) и в настройке тестируемого сервиса на подключение к сервисам-заглушкам. Затем нужно настроить заглушки на отправку обратных ответов с целью имитации работы настоящих сервисов. Например, можно настроить заглушку банка бонусных баллов на возвращение заранее известного количества баллов, скопленных конкретными клиентами.

Использование имитации или применение заглушки

Когда речь заходит о создании заглушек взаимодействующих участников, подразумевается создание сервиса-заглушки, выдающей заранее заготовленные ответы на известные запросы того сервиса, который тестируется. Например, можно задать сервису-заглушке банка бонусных баллов при запросе баланса клиента 123 возвращать значение 15 000. Количество вызовов заглушки, будь то 0, 1 или 100, никакого влияния на прохождение теста не оказывает. Как вариант, вместо заглушки можно использовать имитатор.

При применении имитатора дело заходит чуть дальше и обеспечивается совершение вызова. Если ожидаемый вызов не сделан, тест считается непройденным. Реализация такого подхода требует более интеллектуального приема при создании имитации взаимодействующих сотрудников, а в случае слишком интенсивного использования имитатора могут возникнуть трудности при прохождении теста. А заглушку, как уже говорилось, можно вызывать 0, 1 и более раз.

Но иногда при желании получить ожидаемые побочные эффекты имитаторы могут принести существенную пользу. Например, мне может понадобиться проверить, что при создании нового клиента для него устанавливается новый остаток бонусных баллов. Соблюдение баланса между вызовами заглушек и имитато-

ров — дело тонкое и требующее серьезного отношения к себе при проведении как тестов сервисов, так и блочных тестов. Но фактически в тестах сервисов заглушки я использую значительно чаще имитаторов. Связанные с их использованием компромиссы более подробно рассматриваются в книге Стива Фримена (Steve Freeman) и Ната Прайса (Nat Pryce) *Growing Object-Oriented Software, Guided by Tests* (Addison-Wesley).

Вообще-то я редко использую имитаторы для тестов подобного рода. Но наличие инструментальных средств, способных быть как заглушками, так и имитаторами, считаю полезным.

На мой взгляд, разница между заглушками и имитаторами вполне очевидна, но я все же допускаю, что у кого-то могут возникнуть сложности с ее восприятием, особенно когда вместо этих понятий в ход идут другие, наподобие подделок, шпионов и ловушек. Мартин Фаулер называл все это, включая заглушки и имитаторы, тестовыми дублерами.

Более интеллектуальный сервис-заглушка

Обычно я создаю сервисы-заглушки самостоятельно. Для подобного тестирования мне приходилось использовать все, от Apache или Nginx до Jetty-контейнеров или даже запускаемых из командной строки веб-сервисов на Python. И, возможно, при создании этих заглушек я снова и снова занимался одним и тем же. А вот мой коллега Брендон Брайарс (Brandon Bryars) из ThoughtWorks создал сервер заглушек-имитаторов под названием Mountebank, чем потенциально освободил многих из нас от части работы.

Mountebank можно рассматривать как небольшое программное приспособление, программируемое через HTTP. И любому вызывающему сервису абсолютно безразличен тот факт, что это приспособление было написано на NodeJS. При запуске этому приспособлению отправляются команды, сообщающие, на какой из портов вешать заглушку, какой протокол обрабатывать (в настоящий момент поддерживаются TCP, HTTP и HTTPS, но планируется поддержка и многих других протоколов) и какие ответы следует отправлять после получения запроса. Если нужно воспользоваться имитатором, то это приспособление поддерживает также установку ожидаемых побочных эффектов. При желании конечные точки заглушек можно добавлять или удалять, позволяя одному экземпляру Mountebank служить заглушкой более чем для одной взаимодействующей зависимости.

Следовательно, если нужно запустить тесты сервисов только для одного клиентского сервиса, то можно запустить клиентский сервис и экземпляр Mountebank, работающий как банк бонусных баллов. И если намеченные тесты будут пройдены, я могу с легкой душой развернуть клиентский сервис! Или все же не могу? А как в таком случае быть с теми сервисами, которые вызывают клиентский сервис, — с сервисом технической поддержки и веб-магазином? Знаем ли мы, что внесенное изменение не станет причиной нарушения работы этих сервисов? Ну конечно же, мы упустили из виду весьма важные тесты, находящиеся на вершине пирамиды, — сквозные тесты.

Сложности, связанные со сквозными тестами

В микросервисных системах те возможности, которые становятся видны благодаря пользовательским интерфейсам, поставляются несколькими сервисами. Смысл сквозных тестов, обозначенный в пирамиде Майка Кона, заключается в передаче функциональных возможностей через эти пользовательские интерфейсы всему, что находится ниже, чтобы мы смогли получить представление о множестве компонентов системы.

Следовательно, для реализации сквозного теста нам нужно развернуть вместе сразу несколько сервисов, а затем запустить тест в отношении всех этих сервисов. Вполне очевидно, что область действия этого теста значительно шире, что даст нам больше уверенности в работоспособности системы! В то же время такие тесты выполняются медленнее и выявить источник ошибки в ходе их проведения значительно труднее. Присмотримся к ним пристальнее, воспользовавшись предыдущим примером, и посмотрим, как такие тесты могут быть к нему применены.

Представьте, что нам нужно внедрить новую версию клиентского сервиса. Развернуть изменения для работы в производственном режиме следует как можно скорее, но при этом есть опасения, что внесенные изменения могут нарушить работу либо сервиса технической поддержки, либо веб-магазина. Я думаю, это нам по силам. Развернем разом все наши сервисы и запустим тесты в отношении сервиса техподдержки и веб-магазина, чтобы посмотреть, не допустили ли мы какой-нибудь ошибки. Теперь вполне естественным шагом станет добавление этих тестов к концу конвейера клиентского сервиса (рис. 7.7).

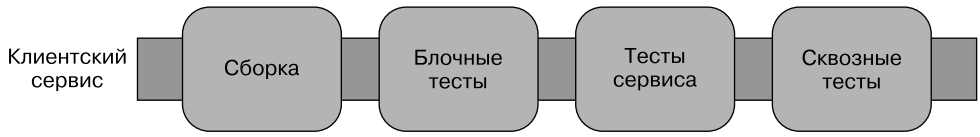


Рис. 7.7. Правильно ли мы поступаем, добавляя сюда этап сквозных тестов?

Вроде пока все идет хорошо. Но прежде всего напрашивается вопрос: какими версиями других сервисов мы должны воспользоваться? Должны ли мы проводить свои тесты в отношении версий сервиса техподдержки и веб-магазина, которые используются в процессе работы в производственном режиме? Было бы, конечно, разумно именно так и сделать, но если настала очередь запуска новой версии либо сервиса техподдержки, либо веб-магазина, как нам поступить в таком случае?

И еще один вопрос: если у нас имеется набор тестов клиентского сервиса, развертывающий множество сервисов и запускающий тесты для проверки их работоспособности, то что можно сказать о сквозных тестах, запускаемых в интересах других сервисов? Если при их проведении тестируются те же самые возможности, то можно поймать себя на мысли, что мы занимаемся одним и тем же и можем в первую очередь повторно потратить время и силы на развертывание всех этих сервисов.

На оба этих вопроса можно найти весьма изящный ответ: нужны несколько конвейеров, объединяющихся в стадию проведения сквозного теста. Как только будет выпущена новая сборка одного из наших сервисов, мы запускаем сквозные тесты, пример которых показан на рис. 7.8. Такие сходящиеся модели могут быть присущи некоторым CI-средствам с расширенной поддержкой сборочных конвейеров.

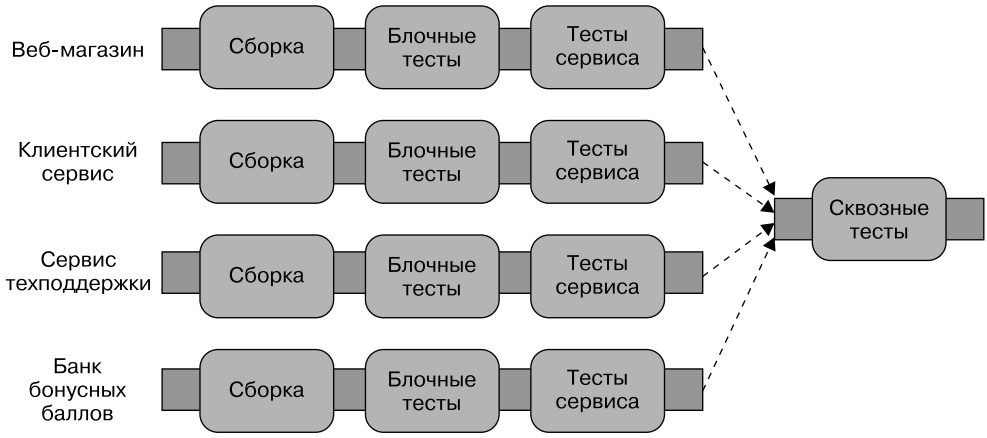


Рис. 7.8. Стандартный способ проведения сквозных тестов в отношении сервисов

Итак, при каждом изменении сервисов мы запускаем тесты, имеющие локальное применение к измененному сервису. Если они будут пройдены, мы запускаем интеграционные тесты. Вроде бы все логично? Но нерешенные проблемы все же остаются.

Недостатки сквозного тестирования

К сожалению, у сквозного тестирования имеется множество недостатков.

Тесты со странностями, не дающие четкого представления об источнике сбоя

Область действия тестов расширяется, а вместе с этим увеличивается и количество активных компонентов. При наличии множества таких компонентов сбои в процессе тестирования могут не показывать, какие именно из тестируемых функций стали причиной сбоя, что не позволит выявить наличие других проблемных мест. К примеру, если проводится тест с целью проверки возможности размещения заказа на одном компакт-диске, но этот тест запускается в отношении четырех или пяти сервисов, то при отказе одного из них мы получим сбой, не имеющий никакого отношения к характеру самого теста. Точно так же временный сетевой сбой может привести к провалу теста без получения какого-либо результата в отношении тестируемых функциональных возможностей.

Чем больше активных компонентов, тем более капризными и менее информативными в заданных для них областях могут стать наши тесты. Если есть такие тесты, которые время от времени дают сбой, но все просто их перезапускают, поскольку они могут быть пройдены при последующих запусках, значит, это тесты со странностями. Ими могут быть не только тесты, охватывающие множество различных процессов, которые могут в данной ситуации стать виновниками сбоя. Часто проблемный характер приобретают тесты, которые охватывают функции, выполняемые в нескольких потоках, где сбой может означать возникновение конфликтной ситуации, истечение времени ожидания или настоящий сбой той или иной функции. Тесты со странностями — ваши враги. Когда они дают сбой, они не вскрывают всю его подоплеку. Мы перезапускаем CI-сборки в надежде, что позже они все же будут пройдены, только для того, чтобы увидеть, как накапливаются отметки о прохождении, и неожиданно оказываемся у разбитого корыта.

Когда проявляются странности теста, важно приложить все силы, чтобы от него избавиться. В противном случае мы начнем терять веру в набор тестов, «который всегда так сбоит». Набор, содержащий тесты со странностями, может попасть под определение того, что Диана Воган (Diane Vaughan) называет нормализацией отклонения, то есть идеи, что со временем мы можем настолько привыкнуть к тому, что все идет не так, что примем это за нормальное, беспроблемное положение вещей[1]. Такое свойственное человеку восприятие действительности означает, что нам нужно выявить тесты со странностями и как можно скорее от них избавиться, не дожидаясь привыкания к тому, что тесты, проявляющие порой свой сбойный характер, нужно воспринимать как норму.

В работе *Eradicating Non-Determinism in Tests* Мартин Фаулер высказал свое отношение к тестам со странностями и сказал, что их нужно отслеживать и, если они не смогут быть немедленно исправлены, просто убирать из набора тестов. Посмотрите, нельзя ли их переписать таким образом, чтобы избежать выполнения тестируемого кода в нескольких потоках. Посмотрите, нельзя ли исходную среду сделать более стабильной. А еще лучше посмотрите, можно ли заменить тест со странностями тестом с меньшей областью действия, при использовании которого появление проблем менее вероятно. В некоторых случаях правильнее будет внести изменения в тестируемый код, упростив таким образом его тестирование.

Кто создает все эти тесты

Разумнее начать с того, что тесты, запускаемые в рамках конвейера для конкретного сервиса, должна создавать та же самая команда, в чьей собственности находится данный сервис (подробнее о владении сервисом мы поговорим в главе 10). А если речь идет о том, что к работе могут быть привлечены сразу несколько команд и теперь сквозные тесты фактически используются ими совместно, то кто тогда должен создавать эти тесты и заниматься их сопровождением?

Мне приходилось наблюдать множество неверных решений, принимаемых в подобной ситуации. Этими тестами занимались все кому не лень, любая коман-

[1] *Vaughan D.* The Challenger Launch Decision: Risky Technology, Culture, and Deviance at NASA. — Chicago: University of Chicago Press, 1996.

да могла, не разбираясь в общем состоянии всего набора, добавлять в них что угодно. Зачастую это приводило к дискредитации контрольных примеров, а иногда к тестированию по принципу упоминавшегося ранее рожка с мороженым. Наблюдались и такие ситуации, при которых отсутствие явного хозяина данных тестов приводило к игнорированию их результатов. Когда тесты проваливались, всем казалось, что проблема связана с работой других команд, поэтому за прохождение тестов не стоит волноваться.

Иногда организации реагировали на это выделением для написания тестов отдельной команды. Такой подход может иметь весьма плачевные последствия. Команда, разрабатывающая программу, все больше отдалялась от тестирования своего кода. Время цикла увеличивалось, поскольку владельцы сервиса дожидались, пока команда, разрабатывающая тесты, сподобится создать сквозные тесты для проверки функционирования только что написанного кода. Поскольку тесты пишет другая команда, та команда, которая создавала сервис, к ним практически не привлекается и поэтому, скорее всего, не знает, как запускать эти тесты и вносить в них исправления. Так как, к моему великому сожалению, такая организационная схема все еще применяется, хочу отметить большой урон, который наносит дистанцирование команды от тестов кода, создаваемого этой командой.

Выработать правильную точку зрения на решение данной проблемы действительно нелегко. Не хотелось бы дублировать усилия, равно как и полностью централизовать их до такой степени, чтобы команды, создающие сервисы, слишком сильно отстранялись от подобных дел. Наиболее удачным из встречавшихся мне случаев соблюдения баланса интересов было отношение к набору сквозных тестов как к общей базе кодов с совместным владением. Команды могли обращаться с этим набором абсолютно свободно, но команды, ведущие разработку сервисов, должны были нести общую ответственность за работоспособность всего набора. Если требуется широкое использование сквозных тестов сразу несколькими командами, то я считаю, что такой подход имеет весьма большое значение, но, по моим наблюдениям, он все еще применяется очень редко и никогда не обходится без проблем.

Насколько продолжительными бывают тесты

На проведение сквозных тестов может уйти немало времени. Мне приходилось наблюдать, как на них тратился целый день, если не больше, а в одном из проектов, над которыми мне пришлось работать, задействование полного регрессионного набора заняло шесть недель! Мне редко попадаются команды, реально курирующие свои наборы сквозных тестов для сокращения накладок в охвате тестами или затрачивающие достаточно времени на повышение их быстродействия.

Эта медлительность в сочетании с тем фактом, что тесты могут оказаться со странностями, может стать серьезной проблемой. Набор тестов, выполнение которых занимает весь день и зачастую проходит со сбоями, не имеющими ничего общего с функциональными изъянами тестируемого кода, является сущей катастрофой. Даже если имеется функциональный сбой, на его выявление может уйти множество часов. К этому моменту большинство из нас уже перейдет к другим делам, и переключение сознания на осмысление ситуации и устранение проблемы будет даваться с большим трудом.

Частично смягчить ситуацию можно запуском тестов в параллельном режиме, например воспользовавшись таким средством, как Selenium Grid. Но этот подход не заменит настоящего понимания того, что должно быть протестировано, и отказа от тестов, надобность в которых уже миновала.

Удаление тестов иногда чревато неприятностями, и я подозреваю, что становлюсь похожим на тех людей, которые хотят избавиться от конкретных мер безопасности в аэропортах. Неважно, насколько безрезультатными могут быть меры безопасности, любым разговорам об их отмене зачастую противопоставляется мгновенная отрицательная реакция с утверждениями о том, что при этом пострадает забота о безопасности людей, или ожиданием победы терроризма. Трудно вести взвешенный разговор о ценности тех или иных дополнений в сравнении с вызываемыми ими осложнениями. Также может быть довольно трудно найти компромисс между рисками и благодарностями. Поблагодарит ли вас кто-нибудь за удаление теста? Вполне возможно. Но вас, несомненно, начнут проклинать, если удаление теста повлечет за собой пропуск какого-либо изъяна. И когда речь заходит о тестовых наборах с обширной областью действия, то без такой возможности нам просто не обойтись. Если одну и ту же функцию проверяют 20 различных тестов, то, поскольку их выполнение занимает десять минут, возможно, от половины из них мы можем отказаться! Для этого требуется более четкое осознание риска, в чем люди пока не преуспели. В результате это разумное курирование и управление тестами, имеющими широкий функциональный охват и высокую нагрузку, происходит крайне редко. Желание заняться этим еще не означает реального проведения этой работы.

Сплошное нагромождение

Когда дело доходит до производительности труда разработчиков, проблема состоит не только в продолжительности циклов получения обратных результатов, связанных с проведением сквозных тестов. При длительном выполнении наборов тестов любой сбой требует времени на устранение, что снижает степень наших ожиданий относительно времени прохождения сквозных тестов. Если мы станем развертывать только те программы, которые успешно прошли все тесты (что, собственно, мы и должны делать!), это будет означать, что степени готовности к развертыванию для работы в производственном режиме достигнет только малая часть наших сервисов.

Это может привести к нагромождению невыполненных дел. Пока будут устраняться изъяны, вызванные сбоем теста интеграции, могут накопиться следующие изменения от взаимодействующих команд. Помимо того что это может усилить сборку, это означает увеличение масштаба изменений развертываемого кода. Эту проблему можно решить тем, чтобы не дать хода изменениям при сбоях сквозных тестов, но при наличии довольно долго выполняемого набора тестов это зачастую не представляется практичным. Попробуйте сказать: «Эй вы, тридцать разработчиков, не вздумайте поставить себе галочку, пока мы не потратим семь часов на отладку сборки!»

Чем шире область действия разработки и выше риск ее выпуска, тем выше вероятность что-нибудь испортить. Возможность частого выпуска нашей программы базируется в основном на идее выпуска небольших изменений сразу же по мере их готовности.

Метаверсия

По окончании сквозного тестирования вполне можно прийти к мысли: раз все эти сервисы данной версии могут работать вместе, почему бы их вместе и не развернуть? Вскоре это превращается в разговор о том, почему бы не воспользоваться номером версии для всей системы. Все по Брэндону Брайарсу (Brandon Bryars): «Теперь у вас 2.1.0 проблем».

Объединяя в единую версию изменения, внесенные сразу в несколько сервисов, мы фактически принимаем идею приемлемости одновременного изменения и развертывания сразу нескольких сервисов. Это становится нормой, не вызывающей сомнений. Поступая таким образом, мы отказываемся от одного из основных преимуществ микросервисов — возможности развертывания отдельно взятого сервиса независимо от всех других сервисов.

Зачастую привычка принимать как должное развертывание вместе сразу нескольких сервисов постепенно приводит к ситуации, в которой сервисы оказываются связанными друг с другом. Вскоре ранее удачно разделенные сервисы все больше запутываются в связях с другими сервисами, и вы этого никогда не замечаете, поскольку никогда не пытаетесь развертывать их по отдельности. В конце концов возникает сплошная путаница, при которой вам приходится дирижировать развертыванием сразу нескольких сервисов, и, как уже говорилось, связанность такого рода может оставить нас в ситуации еще более незавидной, чем та, в которой мы оказались бы при наличии единого монолитного приложения.

И в этом нет ничего хорошего.

Тестируйте маршруты, а не истории

Несмотря на указанные недостатки, при использовании одного или двух сервисов многие пользователи все же смогут справиться с управлением сквозными тестами, проведение которых в подобных ситуациях представляется вполне разумным действием. А что, если используются 3, 4, 10 или 20 сервисов? Очень быстро эти наборы тестов станут весьма сильно раздутыми, что в худшем случае приведет к взрывному росту тестируемых сценариев.

Ситуация усугубится еще больше, если мы попадемся на крючок добавления новых сквозных тестов для каждой крупицы добавляемых функциональных возможностей. Покажите мне набор исходных кодов, где каждая новая история приводит к созданию нового сквозного теста, и я покажу вам раздутый набор тестов с плохим показателем циклов получения ответных данных и огромными перекрытиями охватываемых тестами областей.

Наилучшим противопоставлением этой перспективе является сконцентрированность на небольшом количестве основных маршрутов тестирования для всей системы. Любые функциональные возможности, не охватываемые этими основными маршрутами, должны быть охвачены тестами, анализирующими сервисы в изоляции от всех других тестов. Эти маршруты должны быть взаимосогласованными и находиться в совместном владении. В случае с нашим музыкальным магазином можно сфокусироваться на таких действиях, как заказ компакт-диска, возврат

товара или, может быть, создание нового клиента, то есть на особо значимых взаимодействиях в весьма ограниченных количествах.

Концентрация на небольшом количестве тестов (небольшое количество в моем понимании выражается весьма скромной двузначной цифрой даже для сложных систем) позволит сократить количество недостатков интеграционных тестов, но избежать всех недостатков нам, конечно же, не удастся. А может быть, есть какой-нибудь более подходящий способ?

Тесты на основе запросов потребителей, спасающие ситуацию

С какой из ключевых проблем при использовании упомянутых ранее интеграционных тестов мы пытаемся справиться? Мы стараемся гарантировать, что при развертывании нового сервиса для работы в производственном режиме внесенные нами изменения не внесут разлад в работу потребителей. Одним из способов добиться обозначенной цели без необходимости проведения тестов с участием реальных потребителей является использование *контрактов, составленных на основе запросов потребителей* (CDC).

При использовании CDC определяются ожидания потребителя от сервиса (или поставщика). Ожидания потребителей оформляются в виде кода тестов, который затем запускается в отношении поставщика. При правильной постановке вопроса эти CDC должны запускаться как часть CI-сборки поставщика, гарантируя, что он никогда не будет развернут при нарушении хотя бы одного из этих контрактов. Что очень важно с точки зрения получения обратного ответа при тестировании, эти тесты должны запускаться только в отношении отдельно взятого поставщика, находящегося в изоляции, поэтому они могут выполняться быстрее и надежнее тех сквозных тестов, которые могут быть ими подменены.

В качестве примера еще раз обратимся к нашему сценарию клиентского сервиса. У него есть два отдельных потребителя: сервис техподдержки и веб-магазин. У обоих этих сервисов-потребителей есть ожидания того, как будет вести себя клиентский сервис. В данном примере создаются два набора тестов: по одному для каждого потребителя, представляющего использование клиентского сервиса сервисом техподдержки и веб-магазином. Здесь правилом хорошего тона считается совместный вклад в создание тестов, вносимый как командами потребителей, так и командой поставщика, поэтому вполне вероятно, что их созданием будут заниматься представители команд сервиса техподдержки и веб-магазина с представителями команды клиентского сервиса.

Поскольку эти CDC являются ожиданиями того, как себя будет вести клиентский сервис, их можно запускать в отношении самого клиентского сервиса с любыми заглушенными нижестоящими по отношению к нему зависимостями (рис. 7.9). С точки зрения области действия эти контракты занимают тот же уровень в тестовой пирамиде, что и тесты сервисов, хотя и с совершенно иным фокусом (рис. 7.10). Эти тесты фокусируются на том, как сервис будет использоваться потребителем, и причины их сбоев совсем другие, нежели причины сбоев при проведении тестов сервиса. Если в ходе сборки клиентского сервиса будут нарушены эти CDC, станет

совершенно понятно, на кого из потребителей это повлияет. В таком случае можно будет либо устранить проблему, либо обсудить внесение критических изменений в том порядке, который рассматривался в главе 4. Итак, используя CDC, можно идентифицировать критическое изменение до запуска программного средства в работу в производственном режиме без необходимости применения потенциально недешево обходящихся сквозных тестов.

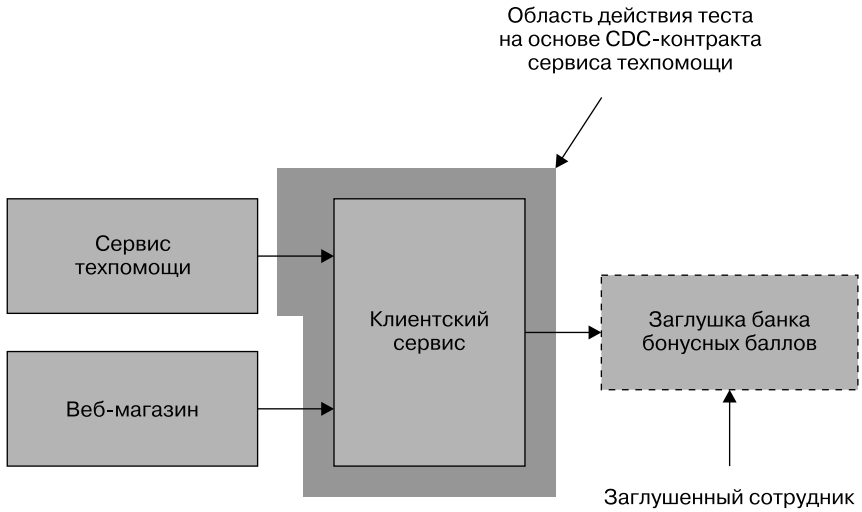


Рис. 7.9. Тесты на основе запросов потребителей в контексте примера клиентского сервиса

Рис. 7.10. Интеграция тестов на основе запросов потребителей в тестовую пирамиду

Pact

Pact представляет собой средство тестирования на основе запросов потребителей, которое первоначально было разработано для внутренних нужд компании Real-Estate.com.au, а теперь является средством с открытым кодом, основной разработкой которого руководил Бет Скурри (Beth Skurrie). Предназначенное сначала только для Ruby средство тестирования Pact теперь включает порты JVM и .NET.

Блок-схема, приведенная на рис. 7.11, показывает, что Pact работает весьма интересным образом. Потребитель начинает с того, что определяет ожидания от производителя с использованием Ruby DSL. Затем вы запускаете локальный сервер-имитатор и в отношении его запускаете ожидания для создания файла спецификации Pact. Этот Pact-файл является не чем иным, как формальной JSON-спецификацией, которую вы, конечно, можете создать и вручную, но с использованием API-интерфейса языка это делается намного проще. Кроме того, вы получаете запущенный сервер-имитатор, который может применяться для последующих изолированных тестов потребителя.

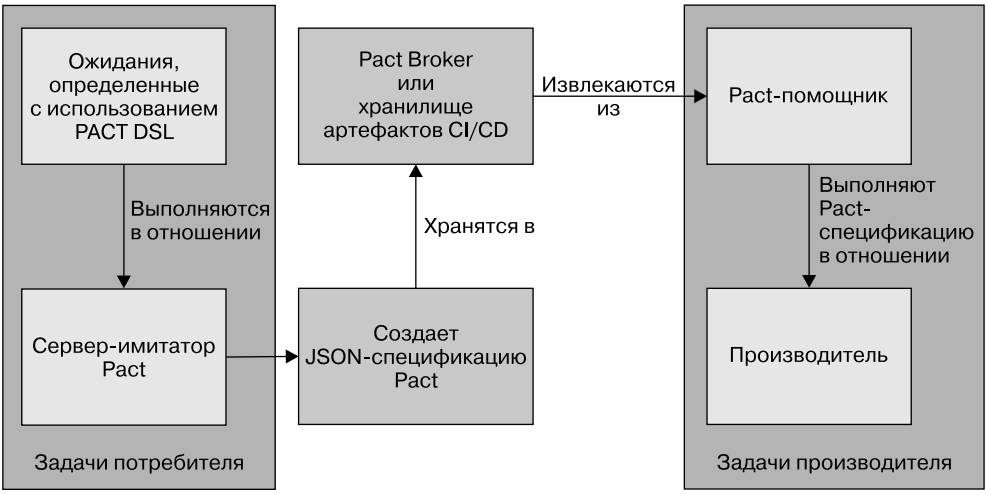


Рис. 7.11. Обзор возможностей тестирования на основе запросов потребителей с использованием Pact

Затем с использованием JSON-спецификации Pact на стороне производителя проверяется соблюдение этой спецификации потребителя для управления вызовами вашего API-интерфейса и проверки ответов. Чтобы все это работало, набор исходных кодов производителя должен иметь доступ к Pact-файлу. Как уже говорилось в главе 6, ожидается, что потребитель и производитель будут находиться в разных сборках. Особенно приятной чертой является использование независимой от выбираемых языков программирования JSON-спецификации. Это означает, что создавать спецификацию потребителя можно с применением Ruby-клиента, но

использовать ее для проверки Java-производителя с помощью принадлежащего Pact JVM-порта.

Поскольку JSON-спецификация Pact создается потребителем, это требует доступа к ней со стороны того артефакта, который создается поставщиком. Эту спецификацию можно сохранить в хранилище артефактов вашего CI/CD-средства или же можно воспользоваться средством Pact Broker, позволяющим хранить несколько версий ваших Pact-спецификаций. Это может позволить вам запускать тесты на основе потребительских запросов в отношении нескольких различных версий потребителей, если потребуется, скажем, применить тесты в отношении версии, используемой в производственном режиме, и в отношении версии, представленной самой последней сборкой.

Как ни странно, у компании ThoughtWorks имеется проект с открытым кодом под названием Pacto, который также является написанным на Ruby средством, использующимся для тестирования на основе запросов потребителей. У него есть возможность записи взаимодействий клиента и сервера для создания ожиданий. Это существенно упрощает создание контрактов для существующих сервисов, составленных на основе запросов потребителей. При использовании Pacto созданные ожидания имеют более или менее статичный характер, а с применением Pact ожидания создаются у потребителя с каждой новой сборкой. Функция определения ожиданий возможностей поставщика, которых пока еще нет, наилучшим образом вписывается в рабочий процесс, в рамках которого сервис-поставщик еще находится (или вскоре будет находиться) в разработке.

О переговорах

В обычной жизни истории называют заполнителями разговоров. CDC также можно рассматривать в данном ключе. Они становятся кодификацией ряда дискуссий о том, как должен выглядеть API-интерфейс сервиса, и когда контракты нарушаются, они становятся поводом для разговоров о том, как должен развиваться API-интерфейс.

Важно понимать, что CDC требуют активного обмена данными и доверия между потребителем и сервисом-поставщиком. Если обе стороны относятся к одной и той же команде (или это один и тот же человек!), то никаких трудностей возникать не должно. Но если вы являетесь потребителем сервиса, предоставляемого сторонним производителем, общение может быть нечастым или же могут отсутствовать доверительные отношения, способствующие работе CDC. В таких ситуациях вам, возможно, придется довольствоваться ограниченным числом интеграционных тестов с более широкой областью действия, охватывающей ненадежный компонент. Кроме того, если API-интерфейс создается для нескольких тысяч потенциальных потребителей, как в случае API общедоступного веб-сервиса, то для определения этих тестов вам, вероятно, придется играть роль потребителя самостоятельно (или же работать выборочно со своими потребителями). Игнорирование ожиданий огромного количества внешних потребителей нам не подойдет, поэтому ориентироваться нужно на возрастание важности CDC!

А нужно ли вообще пользоваться сквозными тестами?

В данной главе уже были описаны во всех подробностях многие недостатки сквозных тестов, число которых по мере вовлечения в тестирование все большего количества активных компонентов стремительно растет. Из разговоров со специалистами, имевшими вполне достаточный опыт реализации микросервисов, я понял, что многие из них со временем полностью избавились от сквозных тестов, отдав предпочтение таким инструментам, как CDC, и уделив внимание совершенствованию мониторинга. Но выбрасывать эти тесты они все равно не спешили. Многие из сквозных маршрутных тестов стали использоваться для мониторинга системы, работающей в производственном режиме, с помощью технологии под названием *«семантический мониторинг»*, которая более подробно рассматривается в главе 8.

Результаты сквозных тестов можно использовать, чтобы подстраховаться перед развертыванием для работы в производственном режиме. Пока вы будете вникать в характер работы CDC и совершенствовать производственный мониторинг и технологии развертывания, сквозные тесты могут стать весьма полезной страховочной сеткой при достижении разумного компромисса между временем цикла тестирования и уменьшением степени риска. Но при улучшениях в других областях можно приступать к сокращению зависимости от сквозных тестов вплоть до полного отказа от них.

Кроме того, ваша работа может проходить в такой обстановке, когда никто не будет стремиться к тому, чтобы быстрее запустить систему в производство и *изучить ее поведение в ходе эксплуатации*, и все силы будут брошены на избавление от дефектов до производственного развертывания системы, даже если это повлечет за собой затягивание сроков поставки программных продуктов. Пока вы не сможете обрести уверенность в том, что все источники дефектов устранены, и не поймете, что вам в процессе эксплуатации все же нужны эффективные средства мониторинга и внесения поправок, решение об использовании сквозных тестов может иметь под собой вполне разумные основания.

Разумеется, вы лучше меня сможете разобраться в управлении рисками в своей собственной организации, но я призываю как следует подумать о том, какие объемы тестирования реально нужны вашей системе.

Тестирование после перевода в производственный режим работы

Основной объем тестирования проводится перед развертыванием системы для работы в производственном режиме. Используя тесты, мы определяем ряд моделей, с помощью которых надеемся доказать, что система работает и ведет себя в соответствии с нашими пожеланиями как с функциональной, так и с нефункциональной точки зрения. Но если наши модели не отличаются совершенством, то при эксплуатации системы мы испытаем раздражение, столкнувшись с проблемами, ошибками, просочившимися в производство, обнаружением новых сбойных режимов работы и тем, что пользователи будут применять систему совершенно неожиданным для нас образом.

Одной из реакций на подобный результат зачастую является разработка все новых и новых тестов и улучшение моделей с целью отлова как можно большего количества дефектов на ранней стадии тестирования и сокращения количества проблем, с которыми приходится сталкиваться в системе, работающей в производственном режиме. Но в определенный момент нам приходится признавать, что отдача от такого подхода снижается. Тестированием, предшествующим развертыванию, мы не можем снизить вероятность сбоев до нуля.

Отделение развертывания от выпуска

Одним из способов отлавливания большего количества дефектов еще до того, как они смогут проявиться в производственном режиме, является выведение области запуска тестов за пределы традиционных этапов, проводимых до развертывания. Если можно развернуть программное средство и протестировать его по месту до направления на него производственной нагрузки, можно выявить дефекты, характерные для заданной среды. Довольно распространенным примером этого может послужить *набор для проведения дымового теста*, то есть коллекция тестов, разработанная для запуска в отношении только что развернутого программного средства с целью подтверждения работоспособности его кода. Эти тесты помогают выявить любые проблемы локальной среды. Если для развертывания любого отдельно взятого микросервиса используется инструкция командной строки (как, собственно, это и следует делать), эта инструкция должна запускать дымовые тесты автоматически.

Еще одним примером может послужить так называемое сине-зеленое развертывание, при котором имеются две одновременно развернутые копии программного средства, но при этом реальные запросы получает только одна из версий.

Рассмотрим пример, показанный на рис. 7.12. В производстве у нас используется клиентский сервис версии v123. Нужно развернуть новую версию v456. Мы развертываем ее рядом с v123, но трафик на нее не направляем. Вместо этого мы проводим тестирование только что развернутой версии на месте. После успешного прохождения тестов производственная нагрузка направляется на новую версию клиентского сервиса v456. Принято на некоторое время старую версию оставлять на прежнем месте, что в случае обнаружения ошибок даст возможность произвести быстрый откат.

Для реализации сине-зеленого развертывания требуется выполнить несколько условий. Во-первых, нужна возможность направления производственного трафика на различные хосты (или наборы хостов). Это можно сделать, изменив DNS-записи или обновив конфигурацию балансировки нагрузки. Во-вторых, нужно иметь возможность предоставления достаточного количества хостов для одновременной работы обеих версий микросервиса. При использовании поставщика облачных услуг, предоставляющего возможности их расширения, выполнить это условие будет нетрудно. Использование сине-зеленых развертываний позволяет сократить риск развертывания, а в случае возникновения какой-нибудь проблемы — вернуться в прежнее состояние. Если у вас все получится, процесс может быть полностью автоматизирован либо с полным запуском нового процесса, либо с возвратом в прежнее состояние без участия человека.

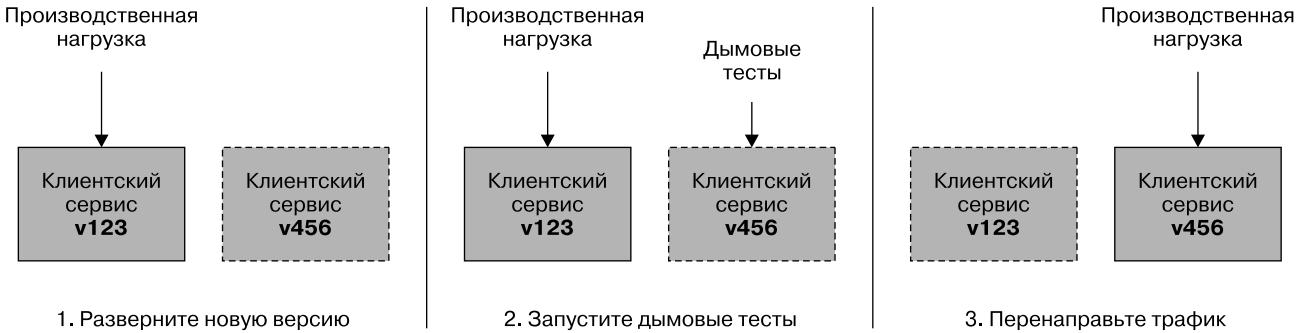


Рис. 7.12. Использование сине-зеленых развертываний для отделения развертывания от выпуска

Помимо преимуществ, заключающихся в возможностях тестирования сервиса на месте до отправки на него производственного трафика, сохраняя старую версию, работающую в процессе выпуска новой версии, мы существенно сокращаем время простоя, связанного с выпуском программного средства. В зависимости от конкретного механизма, используемого для реализации перенаправления трафика, переход от одной версии к другой может быть абсолютно незаметен для клиента, позволяя проводить развертывание с нулевым временем простоя.

В связи с этим стоит подробнее рассмотреть еще одну технологию, которую иногда путают с сине-зелеными развертываниями, поскольку в ней могут использоваться такие же технические реализации. Эта технология называется *канареечным выпуском*.

Канареечный выпуск

С помощью канареечного выпуска только что развернутое программное средство проверяется путем направления на систему некоторых объемов производственного трафика с целью выявления ожидаемого функционирования. Функционирование в соответствии с ожиданиями может охватывать несколько аспектов как функциональных, так и нефункциональных. Например, мы могли бы проверить, что только что развернутый сервис реагирует на запросы в течение 500 мс или что мы наблюдаем пропорциональный уровень ошибок, производимых новым и старым сервисами. Но можно пойти еще дальше. Представьте, что выпускается новая версия сервиса рекомендаций. Можно запустить обе версии одновременно и посмотреть, будут ли рекомендации, выданные новой версией сервиса, приводить к достаточному количеству ожидаемых продаж, что убедит нас в том, что мы не выпустили версию с неоптимальным алгоритмом.

Если новый выпуск окажется плохим, можно быстро вернуть все в прежнее состояние. Если он окажется хорошим, можно запустить увеличивающийся объем трафика через новую версию. Канареечный выпуск отличается от сине-зеленого развертывания тем, что можно рассчитывать на сосуществование версий в течение более длительного времени с частым варьированием объема трафика.

Такой подход активно используется компанией Netflix. До выпуска новые версии сервисов развертываются возле базового кластера, представляющего ту же версию, которая работает в производственном режиме. Затем Netflix запускает часть производственной нагрузки на несколько часов как в адрес новой версии, так и в адрес базовой версии, ведя подсчет в отношении обеих версий. Если канареечный выпуск проходит испытание, компания полностью запускает его в работу в производственном режиме.

При рассмотрении возможности использования канареечного выпуска нужно решить, собираетесь ли вы отводить часть производственных запросов к этому выпуску или будете копировать производственную нагрузку. Некоторые команды могут создавать теневой производственный трафик и направлять его на свой канареечный выпуск. При этом производственная и канареечная версии могут видеть совершенно одинаковые запросы, но внешне будут видны только результаты обработки запросов производственной версией. Тем самым можно будет проводить

сравнительный анализ, исключая при этом то, что сбои в канареечном выпуске станут видны клиентскому запросу. Но работа по созданию теневого производственного трафика может быть нелегкой, особенно если воспроизводимые события или запросы не являются идемпотентными.

Канареечные выпуски являются весьма эффективной технологией и могут помочь в проверке новых версий программных средств при обработке реального трафика, предоставляя при этом инструменты управления риском запуска в производство некачественного выпуска. Но эта технология требует более сложных настроек, чем сине-зеленое развертывание, и более тщательного обдумывания. При этом можно допустить более продолжительное сосуществование различных версий ваших сервисов, чем при использовании сине-зеленого развертывания, что может повлечь за собой более продолжительное, чем раньше, привлечение соответствующего оборудования. Кроме того, понадобится более сложная маршрутизация трафика, поскольку для обретения уверенности в работоспособности вашего выпуска может потребоваться повышать или снижать объем обрабатываемого им трафика. Если вы уже практиковали сине-зеленое развертывание, то уже можете иметь ряд строительных блоков и для канареечного выпуска.

Что важнее: среднее время восстановления работоспособности или среднее время безотказной работы?

Посмотрев на такие технологии, как сине-зеленое развертывание или канареечный выпуск, мы нашли способ тестирования, близкий к работе в производственном режиме (или даже внедренный в эту работу), а также создали средства, помогающие справляться со сбоями в случае их появления. Использование этих подходов является негласным признанием того, что мы не в состоянии обозначить и решить все проблемы до практического выпуска программного средства.

Иногда затрата таких же усилий на совершенствование средств корректировки выпуска может быть намного выгоднее добавления дополнительных автоматизированных тестов. В мире веб-операций это часто рассматривается как компромисс между оптимизацией под *среднее время безотказной работы* (MTBF) и *среднее время восстановления работоспособности* (MTTR).

Технологии сокращения времени восстановления работоспособности могут быть не сложнее быстрого отката в сочетании с хорошо организованным мониторингом (который рассматривается в главе 8), что похоже на сине-зеленые развертывания. Если проблема при работе в производственном режиме обнаруживается на самой ранней стадии и мы тут же проводим откат, то отрицательное воздействие на работу клиентов уменьшается. Мы также можем воспользоваться технологиями, подобными сине-зеленому развертыванию, при развертывании новой версии программного средства и тестировании его на месте до перенаправления пользователей на работу с новой версией.

Для различных организаций компромисс между MTBF и MTTR будет варьироваться, и здесь очень многое зависит от осознания реального влияния сбоев на

производственную среду. Но большинство встречавшихся мне организаций тратило время на создание функциональных тестовых наборов и уделяло совсем мало времени совершенствованию мониторинга или разработке средств восстановления работоспособности или вообще не прикладывало к этому никаких усилий. Получалось, что, устраняя ряд дефектов, проявившихся с самого начала, они не могли устранить абсолютно все дефекты и не были готовы к тому, чтобы справиться с их проявлением при работе в производственном режиме.

Существуют и другие компромиссы, не связанные с MTBF и MTTR. Например, прежде чем создавать трудоемкое программное средство, будет намного целесообразнее сначала попытаться определить, станет ли кто-нибудь им пользоваться, чтобы убедиться в актуальности замысла или бизнес-модели. При таких обстоятельствах тестирование может быть излишним, поскольку определение жизнеспособности самой идеи намного важнее, чем сведения о наличии дефектов при работе в производственном режиме. В подобных ситуациях вполне разумно вообще отказаться от тестирования до испытания в условиях производства.

Межфункциональное тестирование

Основная часть этой главы посвящена тестированию конкретных функциональных возможностей и особенностям его проведения в отношении систем на основе использования микросервисов. Но есть и другой вид тестирования, который важно рассмотреть. Для описания характеристик системы, не поддающихся простой реализации в виде обычной функции, существует обобщающий термин «нефункциональные требования». Эти требования включают в себя такие аспекты, как приемлемая задержка при выводе веб-страницы, количество пользователей, которое должно поддерживаться системой, показатель приспособленности пользовательского интерфейса к работе с ним людей с ограниченными возможностями, степень безопасности клиентских данных.

Термин «нефункциональные» никогда мне не нравился. Некоторые подпадающие под него понятия представляются весьма функциональными по своей природе! Одна из моих коллег, Сара Тарапоревалла (Sarah Taraporewalla), придумала вместо него словосочетание «межфункциональное тестирование» (Cross-Functional Requirements (CFR)), которому я всецело отдаю предпочтение. Оно говорит скорее о том, что эти элементы поведения системы действительно проявляются только в результате большого объема комплексной работы.

Многие, если не большинство CFR-требований могут быть соблюдены только в ходе работы в производственном режиме. Из этого следует, что нужно определить тестовые стратегии, помогающие понять, насколько мы приблизились к соответствию намеченным целям. Эта разновидность тестов попадает в сектор тестирования свойств. Хорошим примером таких тестов может послужить тест производительности, который мы вскоре рассмотрим более подробно.

Выполнение некоторых CFR-требований, возможно, придется отследить на отдельном сервисном уровне. Например, может быть принято решение, что вам необходима более высокая живучесть сервиса платежей, но вполне устраивает более продолжительный простой сервиса музыкальных рекомендаций, поскольку известно,

что основной бизнес может сохранить живучесть, если вы не сможете порекомендовать исполнителей, похожих на Metallica, в течение десяти минут или около того. Эти компромиссы окажут большое влияние на принципы конструирования и развития вашей системы, и здесь снова высокая степень детализации, присущая системам на основе использования микросервисов, даст вам высокие шансы успешно справиться с этими компромиссами.

Тесты выполнения CFR-требований также должны вписываться в пирамидальную схему. Некоторые из них должны носить сквозной характер, например тесты работы при полной нагрузке, другие же не должны быть сквозными. Например, при обнаружении в результате проведения сквозного теста работы при полной нагрузке узких мест, снижающих производительность, следует написать тест с более узкой областью действия, помогающий определить причину возникновения проблемы в будущем. Другие проверки выполнения CFR-требований вполне вписываются в быстрые тесты. Помнится, мне приходилось работать над проектом, где мы настаивали на обеспечении использования HTML-разметки соответствующих функций, помогающих пользоваться сайтом людям с ограниченными возможностями. Проверка созданной разметки на присутствие соответствующих элементов управления может быть проведена очень быстро и без какого-либо обмена данными по сети.

Зачастую бывает так, что размышлять об CFR-требованиях начинают слишком поздно. Я настоятельно рекомендую рассматривать CFR-требования как можно раньше, а также регулярно заниматься их пересмотром.

Тесты производительности. Есть смысл вызывать тесты производительности непосредственно как способ убедиться в соблюдении некоторых межфункциональных требований. При разбиении систем на более мелкие микросервисы повышается количество вызовов, производимых через сетевые границы. Там, где прежде операция могла повлечь за собой один вызов базы данных, теперь она может стать причиной трех или четырех вызовов других сервисов через сетевые границы с соответствующим количеством вызовов баз данных. Все это может снизить скорость работы системы. При этом отслеживание источников задержек приобретает особую важность. Когда имеется последовательность из нескольких синхронных вызовов, замедление работы любой из частей этой последовательности затрагивает всю ее целиком, что потенциально приводит к существенному отрицательному воздействию. Это придает еще большую важность наличию какого-либо способа проведения теста производительности приложений, чем это могло бы быть при использовании более монолитной системы. Зачастую смысл выполнения такого тестирования возникает не сразу, поскольку изначально у системы для него недостаточно компонентов. Суть данной проблемы мне понятна, но довольно часто это приводит к совершенно необоснованному затягиванию и проведению тестов производительности непосредственно перед первым выпуском средства в производственную среду или же вообще к игнорированию этих тестов! Не попадайтесь на эту удочку.

Как и в случае с функциональными тестами, может понадобиться проведение неких тестовых комбинаций. Можно прийти к решению о необходимости выполнения тестов производительности, изолирующих отдельные сервисы, но начать с тестов, проверяющих основные маршруты, существующие в системе. Вам может представиться возможность взять сквозные тесты маршрутов и провести их в полном объеме.

Для получения полноценных результатов данные сценарии зачастую нужно будет запускать с постепенным увеличением количества имитируемых клиентов. Это позволит увидеть изменение задержек вызовов по мере роста рабочей нагрузки. Следовательно, выполнение тестов производительности может занять довольно много времени. Кроме того, чтобы гарантировать отображение в полученных результатах той производительности, которой можно ожидать при работе в производственном режиме, потребуется максимально приблизить систему к производственным параметрам. Это может означать, что вам потребуется достичь более похожего на производственную нагрузку объема данных и понадобится большее количество машин, чтобы соответствовать производственной инфраструктуре, а выполнить подобные задачи может быть весьма нелегко. Даже если вам трудно приблизить параметры тестовой среды к производственным, тесты все равно будут иметь ценность, выражающуюся в выявлении узких мест системы. Но все же нужно знать, что при этом вы можете получать недостоверные негативные или, что еще хуже, недостоверные позитивные результаты.

Из-за большого количества времени, затрачиваемого на проведение тестов производительности, возможность их выполнения в каждом контрольном случае представляется не всегда. Согласно сложившейся практике, запускать поднабор тестов нужно ежедневно, а более крупный набор — еженедельно. Но какой бы из подходов вы ни выбрали, следует запускать эти тесты как можно регулярнее. Чем дальше вы продвинетесь в своей работе без тестов производительности, тем сложнее будет отследить виновников ее снижения. Проблемы производительности весьма трудно устранить, поэтому, если есть такая возможность, нужно уменьшить количество изменений, требующих внимания, с целью выявления новых проблем, и тогда ваша работа станет значительно легче.

И непременно вникайте в результаты! Я был крайне удивлен количеством встреченных мною команд, тративших массу усилий на разработку и выполнение тестов и никогда не анализирующих получаемые значения. Зачастую это происходит по причине незнания того, как должны выглядеть приемлемые результаты. Вам нужны четко поставленные цели. Тогда вы сможете, основываясь на результатах, сделать сборку красной или зеленой и понять, что нужно сделать с красной (не прошедшей тесты) сборкой.

Тесты производительности должны выполняться в сочетании с мониторингом реальной производительности системы (который более подробно рассматривается в главе 8), и в идеале в среде проведения тестов производительности для визуализации поведения системы должны применяться те же инструменты, которые используются и при работе в производственном режиме. Это может существенно упростить сравнение работы в режиме тестирования и в производственном режиме.

Резюме

Все затронутое в данной главе обобщенно можно назвать целостным подходом к тестированию, рассмотрение которого, я надеюсь, даст вам ряд верных направлений проведения таких тестов в ваших собственных системах. Повторим основные положения.

- Проводите оптимизацию в целях получения быстрых ответных результатов и соответствующего разделения тестов на типы.
- Избегайте использования сквозных тестов там, где можно воспользоваться контрактами на основе запросов потребителей.
- Используйте контракты на основе запросов потребителей для фокусировки вокруг них переговоров между командами разработчиков.
- Постарайтесь добиться разумного компромисса и решить, к чему все же следует прикладывать больше усилий — к тестированию или к ускоренному выявлению проблемных моментов при работе в производственном режиме (проводя оптимизацию либо под MTBF, либо под MTTR).

Если вы заинтересованы в получении дополнительных сведений о тестировании, я могу порекомендовать книгу *Agile Testing* (Addison-Wesley), написанную Лизой Криспин (Lisa Crispin) и Джанет Грегори (Janet Gregory), в которой, кроме всего прочего, более подробно рассмотрены вопросы использования секторов тестирования.

Основное внимание в данной главе уделялось приобретению уверенности в том, что код работает еще до того, как он попадет в работу в производственном режиме, но нам также нужно знать, как убедиться в том, что код будет работоспособен после развертывания. В следующей главе мы посмотрим, как проводить мониторинг систем, основанный на применении микросервисов.

8 Мониторинг

Я надеюсь, мне уже удалось показать вам, что разбиение системы на более мелкие, имеющие более высокую степень детализации микросервисы приводит к получению многочисленных преимуществ. Но при этом усложняется мониторинг систем, работающих в производственном режиме. В данной главе будут рассмотрены трудности, связанные с мониторингом и выявлением проблем в наших имеющих более высокую степень детализации системах, а также намечены некоторые подходы, которые можно применить для достижения вполне приемлемых результатов!

Представим следующую ситуацию. Пятница, вторая половина дня, команда уже собралась пораньше улизнуть в какой-нибудь бар, чтобы развеяться и славно встретить предстоящие выходные дни. И тут вдруг приходит сообщение по электронной почте: сайт стал чудить! В Twitter полно гневных посланий в адрес компании, ваш босс рвет и мечет, а перспективы спокойных выходных медленно тают.

В чем нужно разобраться в первую очередь? Что, черт возьми, пошло не так?

Когда работа ведется с монолитным приложением, по крайней мере есть на что свалить вину. Сайт замедлился? Виноват монолит. Сайт выдает странные ошибки? И в этом тоже он виноват. Центральный процессор загружен на все 100 %? Это монолит. Запах дыма? Ну вы уже поняли. Наличие единого источника всех бед отчасти упрощает расследование причин!

А теперь подумаем о нашей собственной системе, основанной на работе микросервисов. Возможности, предлагаемые пользователям, возникают из множества мелких сервисов, часть из которых в ходе выполнения своих задач обмениваются данными еще с несколькими сервисами. У такого подхода есть масса преимуществ (и это хорошо, иначе написание данной книги стало бы пустой тратой времени), но с точки зрения мониторинга мы сталкиваемся с более сложной проблемой.

Теперь у нас есть несколько отслеживаемых серверов, несколько регистрационных файлов, которые нужно будет переворошить, и множество мест, где сетевые задержки могут вызвать проблемы. И как ко всему этому подступиться? Нужно понять, что может стать причиной хаоса и путаницы, с которыми не хотелось бы столкнуться в пятничный полдень (да и в любое другое время!).

На все это есть довольно простой ответ: нужно отслеживать небольшие области, а для получения более общей картины использовать объединение. Чтобы посмотреть, как это делается, начнем с самого простого из возможного — с отдельного узла.

Один сервис на одном сервере

На рис. 8.1 представлена весьма простая установка: на одном хосте запущен один сервис. Теперь нам нужно проводить его мониторинг, чтобы быть в курсе, если что-то пойдет не так, и иметь возможность все исправить. Итак, что именно мы должны отслеживать?

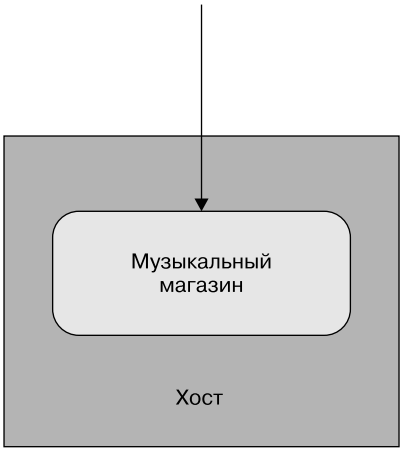


Рис. 8.1. Один сервис на одном хосте

В первую очередь нужно отслеживать сам хост. Центральный процессор, память — полезно отслеживать работу всех этих компонентов. Нужно знать, в каком состоянии они должны быть в нормальной обстановке, чтобы можно было поднять тревогу, когда они выйдут за границы этого состояния. Если нужно запустить программу мониторинга, то можно воспользоваться чем-нибудь вроде Nagios или же применить такой размещаемый прямо на хосте сервис, как New Relic.

Затем нам потребуется доступ к регистрационным журналам с самого сервера. Если пользователь сообщает об ошибке, эти журналы должны зарегистрировать ее и, я надеюсь, сообщить нам о том, когда и где произошла ошибка. В этот момент, имея в распоряжении единственный сервер, мы можем, наверное, получить эти данные, просто зарегистрировавшись на хосте и воспользовавшись для сканирования журнала средством, запускаемым из командной строки. Можно пойти еще дальше и воспользоваться `logrotate`, чтобы убрать старые журналы и не дать им заполнить все дисковое пространство.

И наконец, нужно отслеживать работу самого приложения. Как минимум неплохо будет отслеживать время отклика сервиса. Вероятнее всего, это удастся сделать путем просмотра журналов, поступающих либо из веб-сервера, являющегося фасадом вашего сервиса, либо, возможно, из самого сервиса. При желании получить более совершенную систему можно отслеживать количество ошибок, попавших в отчеты.

Время не стоит на месте, нагрузка растет, и появляется необходимость в расширении...

Один сервис на нескольких серверах

Теперь у нас есть несколько копий сервиса, запущенных на отдельных хостах (рис. 8.2), а запросы к различным экземплярам сервиса распределяются с помощью балансировщика нагрузки. Ситуация начинает усложняться. Нам по-прежнему нужно отслеживать все то же самое, что и раньше, но это следует делать таким образом, чтобы можно было найти источник проблемы.

Будет ли при высокой загруженности центрального процессора, от которой не застрахован ни один из хостов, что-либо указывать на проблему в самом сервисе, или же признак этого будет изолирован на самом хосте, указывая на то, что проблема касается именно его и, возможно, связана с нештатным поведением процесса операционной системы?

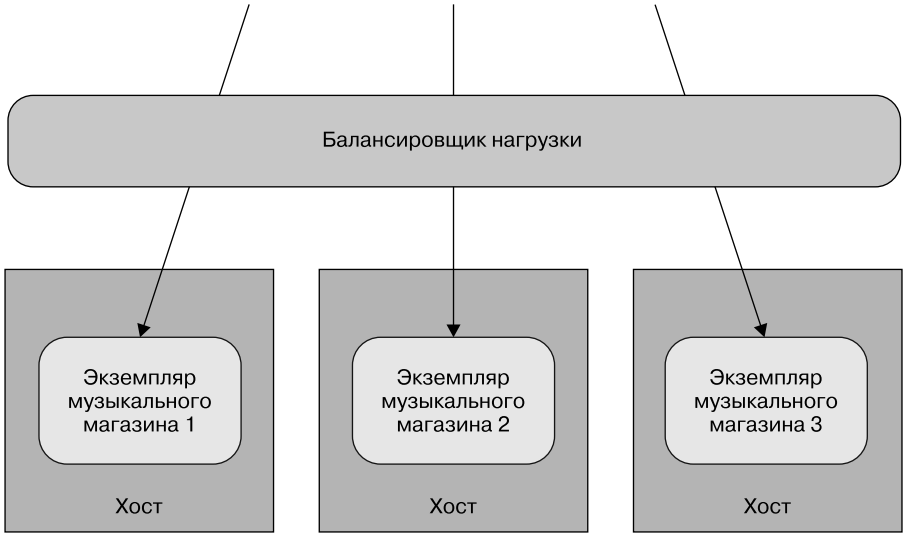


Рис. 8.2. Один сервис, распределенный по нескольким хостам

Итак, в данный момент нам также требуется отслеживать показатели на уровне отдельно взятого хоста и получать оповещения в случае отклонения от норм. Но теперь нужно видеть показатели по всем хостам, а также индивидуальные показатели каждого хоста. Иными словами, нам нужно их объединить, но при этом сохранить возможность рассматривать по отдельности. Подобную группировку хостов допускает Nagios, и пока это нас вполне устроит. Такого подхода, вероятно, будет достаточно для нашего приложения.

Теперь о журналах. Поскольку сервис запущен на более чем одном сервере, нам, чтобы заглянуть вовнутрь, придется, наверное, регистрироваться на каждом из них. Если количество хостов невелико, можно воспользоваться таким средством, как SSH-мультиплексор, которое позволяет запускать одни и те же команды сразу на нескольких хостах. Затем понадобятся большой монитор и поиск виновника с помощью команды `grep "Error" app.log`.

Для задач типа отслеживания времени отклика мы можем, не проводя специального объединения, довольствоваться отслеживанием балансировщика нагрузки. Но нужно, разумеется, отслеживать работу и самого балансировщика: если он начнет барахлить, у нас возникнут проблемы. Тут, наверное, нужно побеспокоиться и о том, как должен выглядеть нормально работающий сервис, поскольку мы будем настраивать балансировщик нагрузки на удаление из приложения неисправных узлов. Надеюсь, что, дойдя до этого места, мы уже сможем выдать несколько идей на сей счет...

Несколько сервисов на нескольких серверах

На рис. 8.3 можно наблюдать еще более интересную картину. Несколько сервисов совместно работают для предоставления определенных возможностей пользователям, и эти сервисы запущены на нескольких хостах, неважно, физических или виртуальных. Как найти искомые ошибки в тысячах строк регистрационных журналов на нескольких хостах? Как определить ненормальную работу одного из серверов или систематический характер ее проявления? И как отследить ошибку в глубине цепочки вызовов между несколькими хостами и определить причину ее появления?

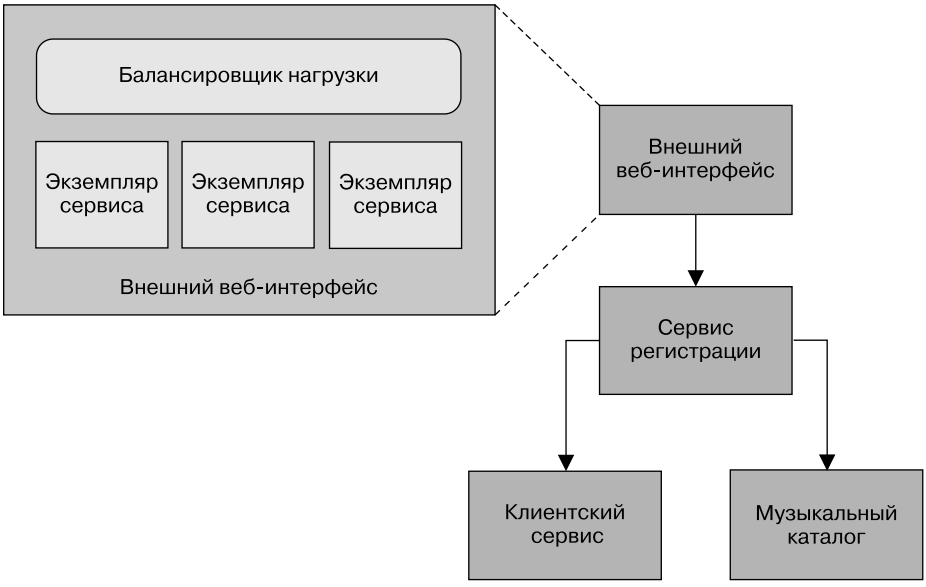


Рис. 8.3. Несколько совместно работающих сервисов, распределенных по нескольким хостам

Ответом могут стать сбор и централизованное объединение всего, до чего только могут дотянуться наши руки: от регистрационных журналов до показателей приложений.

Журналы, журналы и еще журналы...

Теперь сложности начинает создавать количество запущенных нами хостов. Похоже, что SSH-мультиплексирование в данном случае не собирается облегчать ситуацию с извлечением журналов и не существует достаточно больших экранов, чтобы на них могли уместиться терминалы, открытые для каждого хоста. Вместо этого для сбора журналов и предоставления централизованного доступа к ним мы присматриваемся к использованию специализированных подсистем. Одним из примеров может послужить такое средство, как logstash, способное провести парсинг нескольких форматов журнальных файлов и организовать их отправку нижестоящим системам для дальнейшего исследования.

На рис. 8.4 показана система Kibana, построенная на движке ElasticSearch и предназначенная для просмотра журналов. Для поиска по журналам можно воспользоваться синтаксисом запросов, позволяющим ограничивать диапазоны времени и дат или использовать регулярные выражения для поиска соответствующих им строк. Kibana может даже генерировать из журналов диаграммы и отправлять их в ваш адрес, позволяя вам, к примеру, с первого взгляда замечать количество ошибок, зарегистрированных за определенное время.

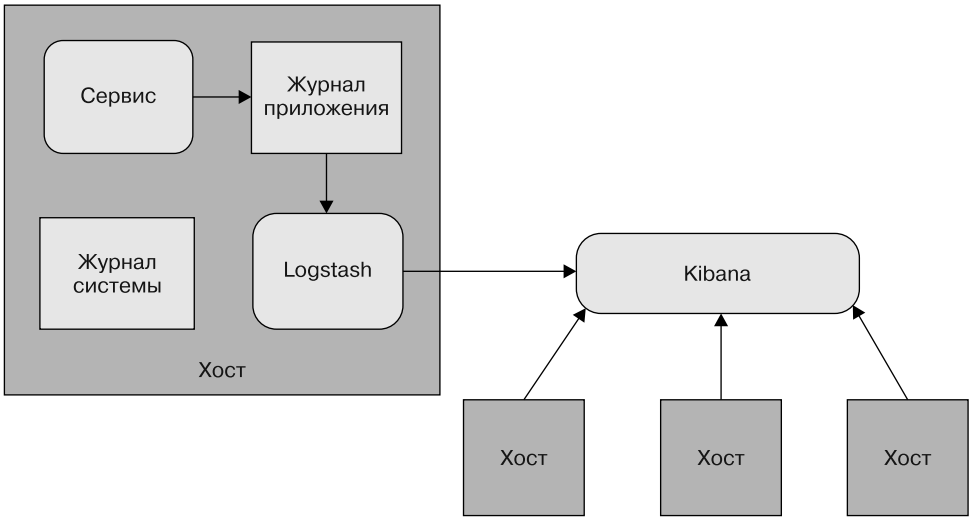


Рис. 8.4. Использование Kibana для просмотра объединенных журналов

Отслеживание показателей сразу нескольких сервисов

Наряду с решением сложностей с просмотром журналов различных хостов нам нужно подыскать наилучшие способы для сбора и просмотра показателей. При рассмотрении показателей более сложных систем может быть трудно узнать, как именно выглядят приемлемые параметры. На нашем сайте замечено около 50 HTTP-ошибок

в секунду с кодами 4XX. Плохо ли это? Загруженность центрального процессора сервисом каталогов увеличилась в обеденное время на 20 % — это нормально? Секрет осведомленности о том, когда следует поднимать тревогу, а когда можно расслабиться, кроется в сборе показателей о поведении системы за период времени, достаточно длительный для того, чтобы могла проявится четкая схема.

В более сложной среде инициализация новых экземпляров сервисов будет происходить довольно часто, поэтому нам нужно выбрать такую систему, которая бы облегчила сбор показателей с новых хостов. Нужно иметь возможность просматривать объединенные показатели для всей системы, например среднюю загруженность центральных процессоров, но также следует объединить показатели для всех экземпляров заданного сервиса или даже показатели для отдельного экземпляра этого сервиса. Это означает, что нужно иметь возможность связать метаданные с показателем, что позволит делать заключения об этой структуре.

Graphite — одна из таких систем, способная существенно упростить решение данной задачи. Она предоставляет очень простой API-интерфейс и позволяет вам отправлять показатели в реальном времени. Затем она позволяет запрашивать эти показатели для создания диаграмм и других наглядных представлений для оценки ситуации. Представляет интерес способ, используемый этим средством для обработки объема данных. Фактически оно настраивается так, чтобы снизить анализ более старых показателей, гарантируя тем самым, что объем не станет разрастаться слишком сильно. Так, к примеру, можно делать одну запись показателей центрального процессора для хостов каждые десять секунд за последние десять минут, затем для каждого последнего дня делать объединенный показатель каждую минуту, опускаясь, возможно, до одного образца каждые 30 минут за несколько последних лет. Таким образом можно хранить информацию о поведении системы за довольно продолжительный период времени, не нуждаясь в хранилище большого объема.

Graphite также позволяет вам объединять образцы или же добираться до их отдельных серий, предоставляя возможность наблюдать время отклика для всей вашей системы, группы сервисов или отдельного экземпляра. Если Graphite у вас по какой-то причине не работает, нужно обеспечить получение сходных возможностей с помощью любого другого инструментального средства. И конечно же, следует обеспечить доступ к необработанным данным для возможности получения своей собственной отчетности или показаний инструментальной панели, если есть потребность в ее использовании.

Еще одно преимущество от знания тенденций проявляется при планировании объемов. Достигли ли мы их пределов? Сколько еще можно проработать до возникновения потребностей в дополнительных хостах? В прошлом, когда мы ставили физические хосты, эта работа зачастую носила ежегодный характер. В новые времена, когда вычислительные мощности предоставляются поставщиками инфраструктур как служб (IaaS) по требованию, мы можем расширять и сужать возможности системы за считаные минуты, если не секунды. Значит, если мы разбираемся в своих схемах использования, то можем удостовериться в том, что в нашем распоряжении имеется достаточно элементов инфраструктуры для удовлетворения

текущих потребностей. Чем интеллектуальнее подход к отслеживанию тенденций и пониманию того, как на них реагировать, тем более экономически оправданной и отзывчивой может стать наша система.

Рабочие показатели сервисов

Операционные системы, под которыми мы работаем, генерируют большое количество показателей, в чем можно убедиться, найдя время для установки на Linux-машине такого средства, как collectd, и указания на него в Graphite. Такие вспомогательные подсистемы, как Nginx или Varnish, также раскрывают весьма полезную информацию, например показатели времени откликов или попаданий в кэш-память. Ну а как насчет вашего собственного сервиса?

Я настоятельно рекомендую предусматривать в ваших сервисах выставление основных показателей их работы. В самом скромном варианте для веб-сервиса нужно, наверное, выставить такие показатели, как время отклика и коэффициент ошибок, что будет жизненно важно, если ваш сервер не имеет в качестве внешнего интерфейса веб-сервера, выполняющего эту работу для вас. Но можно пойти и дальше. Например, для сервиса учетных записей может иметь смысл выставить количество просмотров клиентами их последних заказов, а веб-магазину может быть полезно отследить сумму выручки за последний день.

Зачем все это нужно? По целому ряду причин. Во-первых, существует довольно старое поверье, что 80 % возможностей программных средств никогда не используются. Сейчас я не берусь комментировать его точность, но как человек, создающий программные средства почти 20 лет, я *знаю*, что потратил много времени на разработку возможностей, которые фактически не находили применения. Разве не хочется вам узнать о том, что это за возможности?

Во-вторых, нам непременно нужно реагировать на то, как пользователи используют систему, чтобы вырабатывать свои взгляды на ее улучшение. Показатели, информирующие о поведении системы, могут всего лишь оказать нам помощь в данном вопросе. Мы выдаем новую версию сайта и определяем количество поисков по жанру, поступивших прямо в сервис каталогов. Каков этот показатель, проблематичный или ожидаемый?

И наконец, мы можем никогда не узнать, какие данные окажутся полезными! Бесчисленное количество раз мне хотелось отследить данные, которые могли бы мне помочь понять что-нибудь, только после того, как шанс сделать это был уже упущен. Чтобы справиться с этим позже, я склонялся к ошибке выставления напоказ буквально всего, что можно было, чтобы полагаться на систему показателей.

Для различных платформ существуют библиотеки, позволяющие сервисам отправлять показатели стандартным системам. Одним из примеров такой библиотеки для JVM может послужить Codahale Metrics. Она позволяет сохранять показатели в виде счетчиков, таймеров или измерений, поддерживать показатели за определенный период времени (можно указать показатели вроде «количество заказов за последние пять минут»), а также выходить во внешнюю среду за счет поддержки отправки данных средству Graphite и другим системам объединения данных и создания отчетов.

Искусственный мониторинг

Определить факт работы сервиса в *штатном* режиме можно попытаться, к примеру приняв решение о том, какой уровень загруженности центрального процессора или какое время отклика считать приемлемым. Если система отслеживания обнаружит, что текущие значения выходят за приемлемый уровень, она может подать сигнал тревоги. На это и не только способно такое средство, как Nagios.

Во многих отношениях эти значения на один шаг отстоят от того, что мы действительно хотим отследить, а именно: *работает система или нет*? Чем сложнее взаимодействия между сервисами, тем больше мы отдаляемся от фактического ответа на этот вопрос. А что, если запрограммировать системы мониторинга на действия, чем-то напоминающие действия пользователей и способные отрапортовать, если что-нибудь пойдет не так?

Впервые я сделал это в далеком 2005 году. Тогда я входил в небольшую команду ThoughtWorks, создававшую систему для инвестиционного банка. За торговый день происходила масса событий, оказывающих влияние на рынок. Наша задача заключалась в реагировании на эти изменения и отслеживании их влияния на портфолио банка. Работа велась в условиях постановки жестких сроков готовности, и целью было вместить все наши вычисления менее чем в десять секунд после наступления события. Система состояла из пяти отдельных сервисов, из которых минимум один был запущен на вычислительной grid-архитектуре, которая, помимо прочего, забирала неиспользуемые циклы центральных процессоров у примерно 250 хостов на базе настольных систем в пользу банковского центра восстановления после аварий.

Количество активных элементов в системе подразумевало генерирование большого количества шумовых помех от многих собираемых нами низкоуровневых показателей. Мы не могли получить преимущества от постепенного наращивания системы или ее работы в течение нескольких месяцев чтобы понять, как выглядят *приемлемые* показатели уровня загруженности центрального процессора или даже задержек некоторых отдельных компонентов. Наш подход состоял в выдаче фиктивных событий для оценки части портфолио, не фиксировавшихся нижестоящими системами. Примерно раз в минуту мы заставляли средство Nagios запускать задание командной строки, которое вставляло фиктивное событие в одну из наших очередей. Система его забирала и запускала разнообразные вычисления, как при любом другом задании. Разница заключалась в том, что результаты появлялись в *черновой* ведомости, использовавшейся только для тестирования. Если новая оценка не появлялась в заданное время, Nagios рапортовал об этом как о возникшей проблеме.

Эти создаваемые нами фиктивные события являли собой пример *искусственной транзакции*. Мы использовали искусственную транзакцию, чтобы убедиться, что система вела себя семантически правильно. Именно поэтому такая технология зачастую называется *семантическим мониторингом*.

На практике я убедился в том, что использование искусственных транзакций для подобного семантического мониторинга является намного более удачным индикатором наличия проблем в системе, чем получение тревожных сигналов от

низкоуровневых показателей. Но они не подменяют необходимости получения низкоуровневых показателей, поскольку нам все же нужно иметь подробные сведения, когда возникает потребность в определении причин, по которым семантический мониторинг свидетельствует о наличии проблемы.

Реализация семантического мониторинга

В прошлом реализация семантического мониторинга была весьма непростой задачей. Но мир не стоит на месте, а это означает, что теперь мы можем справиться с ней, не прилагая особых усилий! Вам ведь приходилось запускать на вашей системе тесты? Если нет, прочитайте главу 7 и возвращайтесь сюда. Готово? Ну и отлично!

Если посмотреть на имеющиеся тесты, с помощью которых проводится полное тестирование исходного сервиса или даже всей системы, то обнаружится, что у нас есть многое из того, что нужно для реализации семантического мониторинга. Система уже дает нам все зацепки, необходимые для запуска тестов и проверки результатов. Следовательно, почему бы не запустить поднабор этих тестов на постоянной основе, применив его в качестве способа мониторинга нашей системы?

Разумеется, нам нужно сделать кое-что еще. Во-первых, позаботиться о требованиях, предъявляемых к данным наших тестов. Может понадобиться найти способ, позволяющий приспособить тесты к различным актуальным данным, если они со временем изменяются, или же установить разные источники данных. Например, у нас может быть набор фиктивных пользователей, задействованных в производственном режиме с заранее известным набором данных.

Во-вторых, нужно гарантировать, что мы не вызовем случайно непредвиденные побочные эффекты. Друзья рассказали мне историю об одной компании электронной торговли, в которой случайно запустили тесты в отношении производственных систем составления заказов. Они не замечали своей ошибки до тех пор, пока в головной офис не прибыло большое количество посудомоечных машин.

Идентификаторы взаимосвязанности

Когда для того, чтобы предоставить конечному пользователю любую заданную возможность, происходит взаимодействие большого количества сервисов, один инициализирующий вызов может вылиться в генерирование нескольких вызовов нижестоящих сервисов. Рассмотрим пример. Регистрирующийся клиент заполняет в форме все поля, касающиеся его данных, и щелкает на кнопке отправки. Мы с помощью сервиса платежей скрытно проверяем действительность данных его кредитной карты, даем команду почтовому сервису отправить по почте пакет с проспектами, а сервису электронной почты — отправить приветственное сообщение на электронный адрес клиента. А теперь подумаем, что произойдет, если вызов сервиса платежей закончится выдачей случайной ошибки? Более детально устранение сбоев рассматривается в главе 11, а здесь мы поговорим о сложности диагностики причин случившегося.

Если посмотреть в журналы, то ошибка будет зарегистрирована только в сервисе платежей. Если повезет, мы сможем определить, какой запрос стал причиной возникновения проблемы, и даже просмотреть параметры вызова. А теперь примем во внимание, что это слишком простой пример и что исходный запрос может повлечь за собой целую цепочку вызовов нижестоящих сервисов. Может оказаться так, что события, повлекшие за собой сбой, обрабатывались в асинхронном режиме. Как мы сможем реконструировать весь путь вызовов, чтобы воспроизвести и устранить проблему? Зачастую нужно рассмотреть эту ошибку в более широком контексте исходного вызова — иными словами, провести трассировку цепочки вызовов наверх примерно так же, как делается трассировка стека.

Одним из подходов, который может помочь в данной ситуации, является использование идентификаторов взаимосвязанности. Когда делается первый вызов, для него можно сгенерировать глобальный уникальный идентификатор (GUID). Затем (рис. 8.5) он передается по очереди всем последующим вызовам и может быть помещен в ваши журналы в структурированном виде. Это очень похоже на то, что вы уже делали с такими компонентами, как уровень журнала или дата. При правильном применении инструментов объединения журналов после этого появится возможность выполнить трассировку этого события по всей системе:

```
15-02-2014 16:01:01 Web-Frontend INFO [abc-123] Register
15-02-2014 16:01:02 RegisterService INFO [abc-123] RegisterCustomer ...
15-02-2014 16:01:03 PostalSystem INFO [abc-123] SendWelcomePack ...
15-02-2014 16:01:03 EmailSystem INFO [abc-123] SendWelcomeEmail ...
15-02-2014 16:01:03 PaymentGateway ERROR [abc-123] ValidatePayment ...
```

Вам, конечно же, следует обеспечить осведомленность каждого сервиса о необходимости передачи идентификатора взаимосвязанности. Здесь нужны стандартизация и настойчивость при внедрении этого правила во всей вашей системе. Но когда это будет сделано, понадобится создать инструментарий для отслеживания всех видов взаимодействия. Этот инструментарий может пригодиться при отслеживании шквала событий, нестандартных случаев или даже идентификации наиболее дорогих транзакций, поскольку можно будет отобразить весь каскад вызовов.

Отследить вызовы, проходящие через несколько системных границ, поможет также такое программное средство, как Zipkin. Основываясь на идеях, заложенных в Dapper — собственной системе отслеживания компании Google, Zipkin может предоставить подробно детализированные результаты отслеживания вызовов между сервисами, а также пользовательский интерфейс, помогающий представить полученные данные. Лично я считаю требования, выдвигаемые Zipkin в плане обычных клиентов и систем сбора данных, несколько тяжеловатыми. При условии, что объединение журналов вам уже требуется для других целей, представляется, что будет намного проще вместо этого воспользоваться уже собранными данными, чем придерживаться обязательств по использованию дополнительных источников данных. Тем не менее, если окажется, что вам для отслеживания подобных межсервисных вызовов нужны более совершенные инструменты, вы можете присмотреться и к ним.

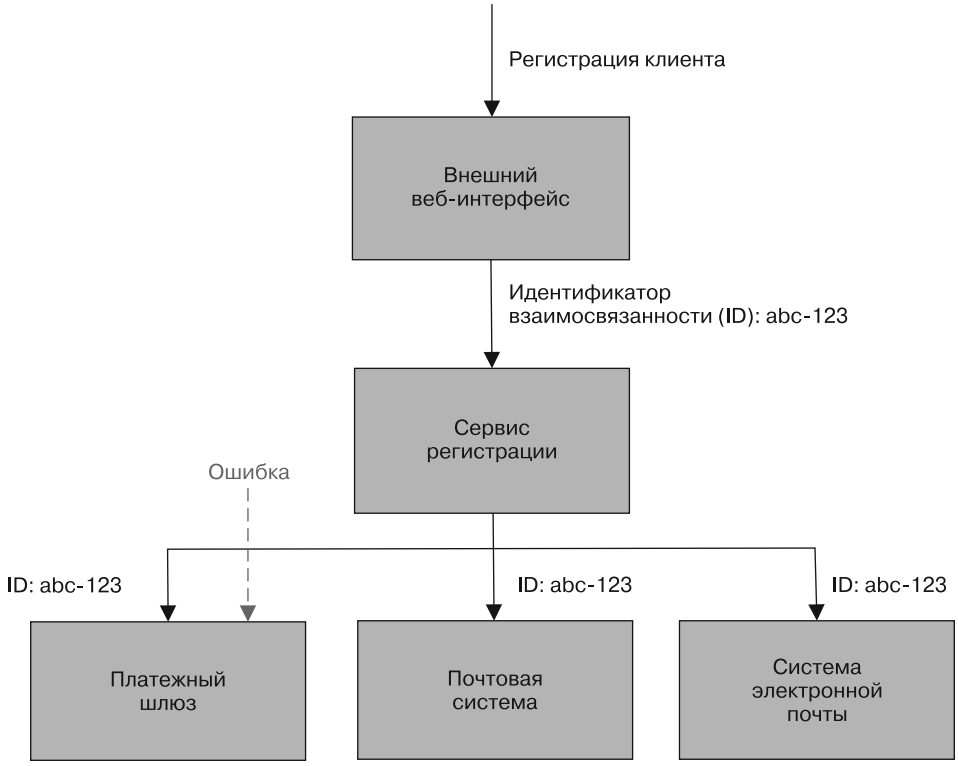


Рис. 8.5. Использование идентификаторов взаимосвязанности для отслеживания цепочек вызовов между несколькими сервисами

Настоящей проблемой, касающейся идентификаторов взаимосвязанности, является то, что зачастую вы не знаете, что нуждаетесь в них, до тех пор, пока не возникнут трудности, причины которых могут быть определены, только если эти идентификаторы использовались с самого начала! Проблема усугубляется тем, что переоснастить код идентификаторами взаимосвязанности очень трудно. С ними нужно будет работать, используя стандартные приемы, чтобы можно было легко провести реконструкцию цепочки вызовов. Изначально все это может показаться дополнительной работой, однако я бы настоятельно рекомендовал вам присмотреться к их размещению в коде при первой же возможности, особенно если предполагается применение в системе архитектурных схем на основе использования событий, что может привести к несколько странно проявляющемуся поведению.

Необходимость выполнения задач, связанных с постоянной передачей по цепочке идентификаторов взаимосвязанности, может стать весомым аргументом для применения узкоспециализированных совместно используемых клиентских библиотек-оболочек. При достижении определенных масштабов становится трудно обеспечить соблюдение всеми однообразных правильных способов вызова нижестоящих сервисов и сбор нужных данных. Достаточно не сделать этого всего лишь

одному входящему в цепочку сервису, и важная информация будет утрачена. Если вы решитесь на создание внутренней клиентской библиотеки, позволяющей добиться безупречной работы подобных механизмов, нужно обеспечить ее самую узкую специализацию и отсутствие привязанности к любому отдельно взятому производственному сервису. Например, если вы используете в качестве исходного протокола обмена данными HTTP, то просто сделайте оболочку из стандартной клиентской библиотеки HTTP, добавив в нее код, обеспечивающий распространение идентификаторов взаимосвязанности в заголовках.

Каскадные сбои

Особенно опасными могут стать сбои, имеющие каскадный характер. Представьте себе ситуацию, при которой даст сбой сетевое соединение между сайтом нашего музыкального магазина и сервисом каталогов. Сами сервисы не дают повода сомневаться в их работоспособности, но они не могут обмениваться данными. Если следить только за работоспособностью отдельно взятого сервиса, определить причину возникновения проблемы не удастся. Выявить проблему можно, используя искусственный мониторинг, например при подражании клиенту в поиске песни. Но для определения ее причины нам нужно будет также получить сведения о том, что одному сервису не виден другой сервис.

Поэтому главную роль начинает играть мониторинг точек интеграции систем. Каждый экземпляр сервиса должен отслеживать и показывать общее состояние нисходящих зависимостей от базы данных до других сотрудничающих сервисов. Следует также позволить выполнять объединение этой информации с целью получения развернутой картины происходящего. Нужно будет наблюдать за временем отклика на нисходящие вызовы, а также замечать возвращение сообщений об ошибках.

Для реализации выключателей сетевых вызовов, помогающих разобраться в каскадных сбоях более изящным образом, допускающим постепенное упрощение вашей системы, можно воспользоваться библиотеками, о которых подробнее поговорим в главе 11. Некоторые из этих библиотек, например Hystrix для JVM, неплохо справляются и с предоставлением вам подобных возможностей проведения мониторинга.

Стандартизация

Как уже говорилось, к числу обязательных, постоянно выполняемых работ по соблюдению баланса относится определение тех мест, где допускается принятие решений с узкой направленностью на отдельно взятый сервис, и тех мест, где нужна стандартизация, распространяющаяся на всю систему. Я считаю, что мониторинг является той самой областью, где крайне важна стандартизация. При наличии сервисов, сотрудничающих множеством различных способов для предоставления пользователям возможностей применения нескольких интерфейсов, нужно иметь целостный взгляд на систему.

Можно постараться вести записи в журналах в стандартном формате. Вам определенно понадобится содержать все показатели в одном месте, а также использовать перечень стандартных названий показателей, чтобы не допускать весьма раздражающих моментов, когда у одного сервиса будет показатель под названием `ResponseTime`, а у другого — точно такой же показатель, но названный `RspTimeSecs`.

Как всегда, для проведения каждой стандартизации имеются вспомогательные средства. Как уже говорилось, ключом к облегчению данной задачи являются правильные действия, поэтому почему бы не предоставить заранее настроенный образ виртуальной машины с готовыми к работе logstash и collectd наряду с прикладными библиотеками, позволяющими реально упростить общение с таким средством, как Graphite?

Расчет на аудиторию

Все данные собирают для вполне определенной цели. Точнее, мы проводим сбор всех этих данных для разных людей, чтобы помочь им справиться с поставленными перед ними задачами. Эти данные побуждают к действию. Часть данных нужна как сигнал к незамедлительному побуждению к действию нашей команды поддержки, например, в случае, когда не проходит один из тестов искусственного мониторинга. Другая часть данных, наподобие того факта, что нагрузка на центральный процессор увеличилась по сравнению с прошлой неделей на 2 %, потенциально представляет интерес, только если мы выполняем планирование использования ресурсов. Возможно также, что ваш начальник хочет немедленно узнать о том, что после последнего выпуска выручка упала на 25 %, но, скорее всего, волноваться не о чем, так как за последний час поиск по ключевым словам `Justin Bieber` возрос на 5 %.

То, что нашим людям нужно видеть и на что реагировать прямо сейчас, отличается от того, что им нужно при более глубоком анализе. Поэтому при разделении на категории людей, просматривающих данные мониторинга, нужно исходить из того:

- о чем они хотят знать в текущий момент;
- что им может понадобиться чуть позже;
- как они хотят воспользоваться данными.

Оповещайте о том, что они должны знать прямо сейчас. Создайте большое окно с данной информацией, располагающееся в углу экрана. Предоставьте легкий доступ к данным, о которых они хотят узнать чуть позже. И уделите время общению с ними, чтобы узнать, как они хотят воспользоваться данными. Обсуждение всех нюансов графического отображения количественной информации, конечно же, выходит за рамки данной книги, но для начала отличным подспорьем может послужить прекрасная книга Стивена Фью (Stephen Few) *Information Dashboard Design: Displaying Data for Ata-Glance Monitoring* (Analytics Press).

Перспективы

Мне известно множество организаций, в которых показатели разбросаны по разным системам. Показатели на уровне приложений, например количество размещенных заказов, оказываются в собственных аналитических системах вроде Omniture, которые зачастую доступны только избранным *участникам бизнес-процессов*, или же попадают в те страшные хранилища данных, где им и суждено будет сгинуть. Получить отчеты из подобных систем в реальном времени зачастую невозможно, хотя ситуация начинает выправляться. В то же время такие *системные* показатели, как время отклика, частота появления ошибок и загруженность центрального процессора, хранятся в системах, к которым могут иметь доступ рабочие команды. Эти системы обычно доступны для составления отчетов в реальном времени, и, как правило, их суть состоит в побуждении к немедленному вызову действия.

Исторически сложилось так, что узнавать о бизнес-показателях на день или два позже было вполне приемлемо, поскольку обычно все равно быстро реагировать на эти данные и предпринимать что-нибудь в ответ мы не могли. Но теперь мы работаем в такой обстановке, когда многие из нас могут выдать и несколько выпусков в день. Теперь команды оценивают себя не по тому, сколько пунктов они могут завершить, а по тому, сколько времени будет потрачено на то, чтобы код из ноутбука ушел в производство. В такой среде, чтобы совершать правильные действия, нужно иметь все показатели под рукой. Как ни странно, в отличие от наших операционных систем, те самые системы, которые сохраняют бизнес-показатели, зачастую не настроены на незамедлительный доступ к данным.

Так зачем же обрабатывать оперативные и деловые показатели одинаково? В конечном счете оба типа показателей разбиваются на события, которые свидетельствуют о чем-то случившемся в момент времени X. Следовательно, если есть возможность унифицировать системы, используемые для сбора, объединения и хранения этих событий и открывающие к ним доступ для составления отчетов, то мы получим намного более простую архитектуру.

Частью подобного решения может стать Riemann — сервер событий, позволяющий на довольно высоком уровне выполнять объединение и маршрутизацию событий. В этой же области работает и средство Suro, являющееся конвейером данных компании Netflix. В Suro явным образом обрабатываются как показатели, связанные с пользовательским поведением, так и большинство оперативных данных, фиксируемых в журналах приложений. Затем эти данные могут быть направлены различным системам вроде системы Storm, предназначенной для выполнения анализа в реальном времени, системы Hadoop, служащей для пакетной обработки в автономном режиме, или системы Kibana, предназначенной для анализа журнальных записей.

Многие организации движутся в совершенно ином направлении: отказываются от использования цепочек специализированных инструментальных средств для разных типов показателей и склоняются к применению более универсальных систем маршрутизации событий, способных на существенное расширение. Эти системы используются для предоставления намного большей гибкости при реальном упрощении нашей архитектуры.

Резюме

Итак, мы усвоили довольно большой объем информации! Я попытаюсь свести содержимое главы к нескольким легко воспринимаемым советам.

Для каждого сервиса:

- отслеживайте как минимум возвращающееся время отклика. Затем займитесь частотой появления ошибок, после чего приступайте к работе над показателями на уровне приложения;
- отслеживайте приемлемость всех ответов от нижестоящих сервисов, включая как минимум время отклика при вызовах нижестоящих сервисов, а в лучшем случае — частоту появления ошибок. Помочь в этом могут такие библиотеки, как Hystrix;
- приведите к общему стандарту способ и место сбора показателей;
- помещайте регистрационные записи в стандартном месте и по возможности используйте для них стандартный формат. Если для каждого сервиса будут применяться разные форматы, объединение будет сильно затруднено;
- отслеживайте показатели исходной операционной системы, чтобы можно было выявлять отклоняющиеся от нормы процессы и планировать использование ресурсов.

Для системы:

- объединяйте показатели на уровне хоста, например показатели загруженности центрального процессора с показателями на уровне приложения;
- выбирайте такие инструменты хранения показателей, которые позволяют выполнять их объединение на системном или сервисном уровне и извлекать их для отдельно взятых хостов;
- выбирайте такие инструменты хранения показателей, которые позволяют сохранять данные достаточно долго для того, чтобы можно было отследить тенденции в работе вашей системы;
- используйте для объединения и хранения регистрационных записей единое средство, поддерживающее обработку запросов;
- строго придерживайтесь стандартизации при использовании идентификаторов взаимосвязанности;
- разберитесь, чему для перехода к действию требуется вызов, и выстройте соответствующим образом структуру оповещения и вывода на панель отслеживания;
- исследуйте возможность унификации способов объединения всевозможных показателей, выяснив, пригодятся ли вам для этого такие средства, как Suro или Riemann.

Я также попытался обозначить направление, в котором развивается мониторинг: уход от систем, специализирующихся на выполнении какой-нибудь одной задачи, и переход к разработке типовых систем обработки событий, позволяющих составить более целостное представление о вашей системе. Это очень интересная

и постоянно развивающаяся область, и хотя ее полноценное исследование не входит в задачи данной книги, я надеюсь, что для начала полученной вами информации вполне достаточно. При желании расширить познания можете обратиться к моей более ранней публикации *Lightweight Systems for Realtime Monitoring* (O'Reilly), где глубже рассматриваются как некоторые из этих, так и многие другие идеи.

В следующей главе мы коснемся другого целостного представления наших систем и рассмотрим ряд уникальных преимуществ, предоставляемых архитектурами с высокой степенью детализации в области решения задач обеспечения безопасности, а также ряд сложностей, с которыми при этом придется столкнуться.

9 Безопасность

Нам уже знакомы истории о пренебрежении мерами безопасности крупномасштабных систем, приведшие к выставлению наших данных на обозрение всякого рода сомнительным личностям. Но в последнее время откровения вроде тех, которыми поделился Эдвард Сноуден, повысили нашу осведомленность о ценности данных о нас, которые хранятся в различных компаниях, а также ценности данных о клиентах, которые мы сами храним в создаваемых нами системах. В этой главе будет дан краткий обзор ряда аспектов безопасности, которые следует принимать во внимание при разработке систем. Несмотря на то что предоставляемая информация не является исчерпывающей, она покажет ряд доступных вам основных вариантов и станет отправной точкой для ваших собственных дальнейших исследований.

Нам нужно подумать о том, в какой защите нуждаются наши данные при их перемещении из одной точки в другую и какая им нужна защита, когда они просто хранятся в системе. Нужно продумать вопросы обеспечения безопасности используемых операционных систем, а также сетевого оборудования. Перед нами обширное поле для размышлений и реальных действий! В какой мере нам следует обезопасить свою систему? Как определить критерии достаточности мер безопасности?

Но, кроме этого, нужно подумать и о людях. Как мы узнаем о том, кто есть кто и как следует действовать дальше? И какое отношение все это имеет к обмену данными между нашими серверами? С этого и начнем.

Аутентификация и авторизация

Когда речь заходит о людях и о том, что вступает во взаимодействие с нашей системой, основными понятиями являются аутентификация и авторизация. В контексте безопасности под аутентификацией понимается процесс подтверждения того, что сторона, заявившая о себе, таковой на самом деле и является. Для человека аутентификация обычно сводится к набору имени и пароля. Предполагается, что только этот человек имеет доступ к такой информации и потому лицо, набравшее ее, и должно быть данным человеком. Разумеется, существуют и другие, более сложные системы. Мой телефон теперь позволяет мне использовать отпечаток пальца, чтобы я мог подтвердить свою личность. В общем смысле, когда речь заходит о том, кто или что проходит аутентификацию, мы называем данную сторону принципалом.

Авторизация представляет собой механизм, с помощью которого мы переходим от принципала к действию, которое ему разрешено совершить. Зачастую, когда принципал проходит аутентификацию, нам предоставляется информация о нем, которая поможет принять решение о том, что ему будет позволено делать. Нам, к примеру, сообщат, в каком отделе или офисе он работает, то есть мы получим те части информации, которые могут использоваться нашей системой для принятия решения о том, что он может, а чего не может делать.

Если речь идет о единых, монолитных приложениях, то аутентификацией и авторизацией для вас занимается само приложение. К примеру, Django, веб-среда языка Python, поставляется с уже готовой системой управления пользователями. Что же касается распределенных систем, нам нужно подумать над созданием более совершенных схем. Мы не хотим, чтобы все регистрировались в различных системах отдельно, применяя для каждой системы разные имена пользователей и пароли. Наша цель состоит в создании единого идентификатора, позволяющего проходить аутентификацию только один раз.

Общепринятые реализации технологии единого входа

Общепринятым подходом к аутентификации и авторизации является использование какого-либо из решений единого входа (SSO). Соответствующие возможности в данной области предоставляются SAML — реализации, доминирующей в области промышленных предприятий, и OpenID Connect. В них применяются более или менее одинаковые основные понятия, хотя терминология немного различается. В данной главе будут использоваться термины, применяемые в SAML.

Когда принципал пытается получить доступ к ресурсу (например, интерфейсу на веб-основе), он перенаправляется на аутентификацию с участием провайдера идентификации. При этом у него могут быть запрошены имя пользователя и пароль или же может быть использовано что-то более совершенное — вроде двухфакторной аутентификации. После того как поставщик убедится в том, что принципал был аутентифицирован, он выдает информацию сервис-провайдеру, позволяя ему решать, нужно ли предоставлять принципалу доступ к ресурсу.

Провайдером идентификации может быть система на внешнем оборудовании или что-нибудь, что находится внутри вашей организации. Например, компанией Google предоставляется провайдер идентификации OpenID Connect. Но промышленным предприятиям свойственно иметь собственного провайдера идентификации, который может быть связан с сервисом каталогов вашей компании. Сервисом каталогов может быть какое-либо средство вроде Lightweight Directory Access Protocol (LDAP) или Active Directory. Эти системы позволяют хранить информацию о принципалах, свидетельствующую о тех ролях, которые они играют в организации. Зачастую сервис каталогов и провайдер идентификации представляют собой единое целое, а иногда они могут быть разделены, но тесно связаны друг с другом. Например, Okta содержит SAML-провайдер идентификации, выполняющий задачи двухфакторной идентификации, но как источник достоверных данных он может быть связан с сервисами каталогов вашей компании.

SAML представляет собой стандарт на основе использования SOAP-протокола, он известен тем, что с ним весьма непросто работать, несмотря на доступность поддерживающих его библиотек и инструментальных средств. OpenID Connect является стандартом, возникшим в качестве конкретной реализации OAuth 2.0 и основанным на способах управления технологией единого входа, принятых Google и рядом других компаний. В нем используются простые REST-вызовы, и, на мой взгляд, это сделано, скорее всего, для проникновения на рынок промышленных предприятий за счет простоты использования. Сейчас главным камнем преткновения является отсутствие поддерживающих его провайдеров идентификации. Для открытых для публичного просмотра сайтов может вполне подойти использование в качестве вашего провайдера средств компании Google, но для внутренних систем или систем, в которых требуются повышенный контроль и отображение того, как и где устанавливаются ваши данные, понадобится собственный домашний провайдер идентификации. На момент написания книги доступными в данном качестве были два из весьма немногих вариантов, OpenAM и Gluu, что не может сравниться с богатством выбора вариантов для SAML (включая средство Active Directory, которое, похоже, получило всеобщее распространение). До тех пор пока имеющиеся провайдеры идентификации не начали поддерживать OpenID Connect, распространение этого средства будет ограничено ситуациями, при которых люди будут вполне удовлетворены использованием публичных провайдеров идентификации.

Итак, несмотря на то, что, на мой взгляд, у OpenID есть будущее, вполне возможно, что на его широкое распространение понадобится время.

Шлюз технологии единого входа

При установке микросервиса вопрос о перенаправлении на провайдера идентификации и о квитировании связи с ним может решаться каждым сервисом. Очевидно, это приведет к многократному дублированию работы. Конечно, в решении вопроса может помочь общая библиотека, но мы ведь должны избегать связанности микросервисов, исходящей от общего кода. К тому же такое решение будет бесполезным в случае использования нескольких различных технологических стеков.

Вместо обременения каждого сервиса решением вопросов управления квитированием с вашим провайдером идентификации можно воспользоваться шлюзом, работающим в качестве прокси-сервера и располагающимся между вашими сервисами и внешним миром (рис. 9.1). Идея состоит в том, что мы можем централизовать поведение для перенаправления пользователя и выполнения квитирования в одном месте.

Но нам еще нужно решить проблему получения расположенными ниже сервисами такой информации о принципалах, как их имена пользователей или роли, которые они играют в организации. Если используется HTTP, эту информацию можно поместить в заголовки. Одним из средств, выполняющих эту задачу за вас, является Shibboleth, использование которого вместе с Apache для получения отличных результатов в управлении интеграцией с провайдерами идентификации на основе SAML мне уже приходилось наблюдать.

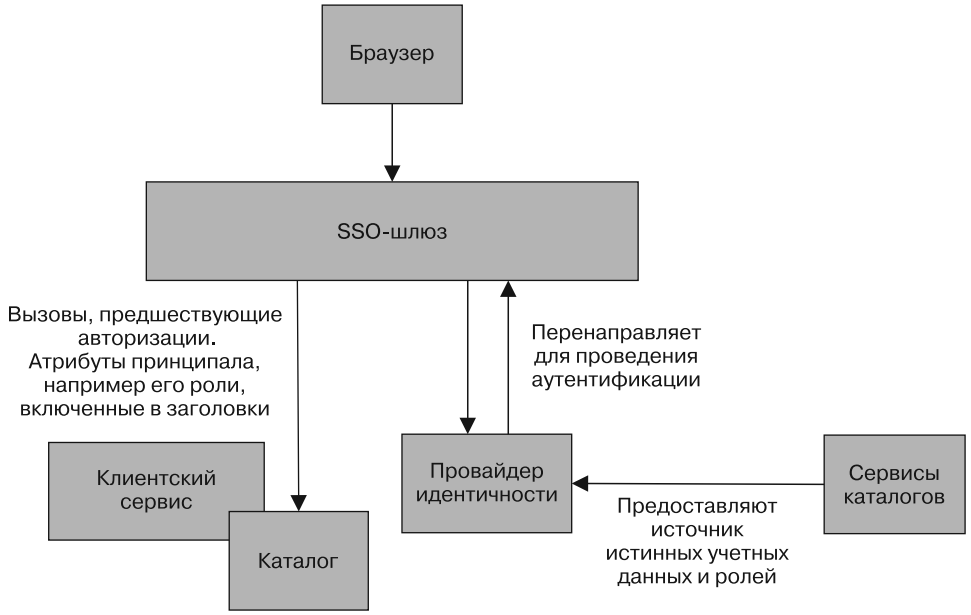


Рис. 9.1. Использование шлюза для управления единым входом (SSO)

Если будет принято решение о том, чтобы переложить ответственность за аутентификацию на шлюз, возникнет еще одна проблема, связанная с затруднениями при осмыслении поведения микросервисов, когда они рассматриваются в изоляции от всего остального. Помните, как в главе 7 исследовались затруднения, связанные с воспроизведением среды, подобной той, в которой ведется работа в производственном режиме? Если вы пойдете по пути использования шлюза, нужно гарантировать разработчикам возможность запуска их сервисов за этим шлюзом без приложения чрезмерных усилий.

И еще одной, последней проблемой, связанной с этим подходом, является возможность возникновения у вас ложного чувства безопасности. Мне нравится идея глубоко эшелонированной обороны — от периметра сети к подсети, брандмауэру, машине, операционной системе, используемому оборудованию. У вас есть возможность реализовать меры безопасности на всех этих рубежах, часть из которых мы вскоре рассмотрим. Мне встречались люди, которые клали все яйца в одну корзину, всецело полагаясь на шлюз. И нам всем известно, что происходит, когда у нас есть единая точка отказа...

Разумеется, шлюз можно применять и для других целей. Например, при использовании уровня Apache-экземпляров, выполняющих Shibboleth, на этом уровне можно также принимать решения о завершении HTTPS, запуске обнаружения вторжений и т. д. Но при этом нужно проявлять осторожность. На уровни шлюзов зачастую возлагается все больше и больше функциональных обязанностей, что может вылиться в возникновение гигантской точки связывания. И чем больше функциональных нагрузок, тем шире становится поле для проведения атак.

Авторизация с высокой степенью детализации

Шлюз может быть в состоянии обеспечить вполне эффективную широкомасштабную аутентификацию. Например, он может воспрепятствовать доступу к приложению технической поддержки любого незарегистрировавшегося пользователя. При условии, что шлюз в результате аутентификации может извлечь атрибуты, касающиеся принципала, можно рассчитывать на возможность принятия более гибких решений. Например, весьма распространено сведение людей в группы или присвоение им тех или иных ролей. Этой информацией можно воспользоваться, чтобы понять, что они могут делать. Следовательно, доступ к приложению технической поддержки можно предоставить только принципалам с конкретной ролью (например, только сотрудникам). Но, кроме разрешения (или запрещения) доступа к конкретным ресурсам или конечным точкам, нам нужно допустить всех остальных к самим микросервисам, а для этого следует принимать дальнейшие решения о том, какие действия разрешать.

Возвращаясь к приложению технической поддержки, нужно решить, следует ли позволять видеть абсолютно все подробности любому сотруднику организации? Скорее всего, их работа будет распределена по ролям. Например, принципалу из группы `CALL_CENTER` может быть разрешено просматривать любые информационные блоки, относящиеся к клиенту, за исключением подробностей, касающихся его платежей. Принципалу также может быть разрешено выдавать возвраты, но сумма может быть ограничена. А вот сотрудникам, имеющим роль `CALL_CENTER_TEAM_LEADER`, может быть позволено выдавать более крупные суммы возврата.

По отношению к отдельно взятому микросервису эти решения должны быть локальными. Мне встречались люди, применяющие различные атрибуты, предоставляемые провайдерами идентификации, самым ужасным образом, используя совершенно мелочные роли вроде `CALL_CENTER_50_DOLLAR_REFUND` (то есть специалист колл-центра с правами возврата сумм до 50 долларов), где они в конечном итоге помещали в свои службы каталогов информацию, относящуюся к какой-то небольшой части одной из характеристик нашего системного поведения. Поддержка таких установок превращается в настоящий кошмар и оставляет для наших сервисов весьма скромные возможности иметь собственный независимый жизненный цикл, поскольку вдруг оказывается, что часть информации, касающейся поведения сервиса, находится за его пределами, возможно, в системе, управляемой другой частью организации.

Вместо этого предпочтение нужно отдавать ролям более общего плана, смоделированным вокруг характера работы вашей организации. Если вспомнить все, о чем говорилось в предыдущих главах, можно сделать вывод: мы стремимся создавать такие программные средства, которые соответствуют порядку работы нашей организации. Поэтому и роли нужно выбирать, руководствуясь тем же принципом.

Взаимная аутентификация и авторизация сервисов

До сих пор термин «принципал» использовался для описания любых субъектов, способных пройти аутентификацию и авторизоваться для выполнения тех или иных

действий, но фактически приводимые примеры касались людей, использующих компьютеры. А как насчет программ или других сервисов, проходящих взаимную аутентификацию?

Разрешение всего в пределах периметра

Нашим первым вариантом может быть простое предположение, что любые вызовы сервиса, которые делаются внутри нашего периметра, вызывают безоговорочное доверие.

В зависимости от степени конфиденциальности данных этот вариант может стать вполне приемлемым. Некоторые организации пытаются обеспечить безопасность по периметру своих сетей и, следовательно, предполагают, что, когда два сервиса общаются друг с другом, делать что-либо дополнительно не нужно. Но стоит только взломщику проникнуть в вашу сеть, у вас практически не будет защиты против посреднической атаки. Если взломщик решит перехватить и прочитать отправленные данные, внести в них изменения без вашего ведома или даже при некоторых обстоятельствах подделать диалог, вы об этом можете даже не догадаться.

Доверие внутри периметра, безусловно, наиболее распространенная из встречавшихся мне в организациях форм. Они могут решить запустить этот трафик через HTTPS-протокол, но не более того. Не могу сказать, что это подходящий вариант! Боюсь, что для большинства организаций, в которых я видел использование данной модели, безоговорочная доверительность не являлась осознанным решением, скорее большинство их представителей просто не понимало существующих рисков.

Стандарт HTTP(S) Basic Authentication

Стандарт HTTP Basic Authentication позволяет клиенту отправлять имя пользователя и пароль в стандартном HTTP-заголовке. После этого сервер может проверить эти элементы и подтвердить, что клиенту разрешен доступ к сервису. Преимущество заключается в том, что это предельно понятный и повсеместно поддерживаемый протокол. Проблема в том, что отправка по протоколу HTTP весьма проблематична, потому что имя пользователя и пароль не отправляются в безопасном режиме. Любая промежуточная инстанция может просмотреть информацию, находящуюся в заголовке, и увидеть данные. Поэтому HTTP Basic Authentication следует использовать с привлечением протокола HTTPS.

При использовании HTTPS клиент получает твердые гарантии, что сервер, с которым он ведет обмен данными, является тем самым сервером, с которым клиент задумал связаться. Кроме этого, дается гарантия защиты от подглядываний за трафиком между клиентом и сервером или от манипуляций с полезной нагрузкой.

Серверу нужно управлять собственными SSL-сертификатами, что может стать проблематичным при управлении несколькими машинами. Некоторые организации берут на себя процесс выдачи сертификатов, что становится дополнительной адми-

нистративной и рабочей нагрузкой. Инструментальные средства, занимающиеся автоматизированным управлением этим процессом, еще не достигли достаточного совершенства, и заниматься процессом выдачи вам не стоит. Самостоятельно подписанные сертификаты аннулировать нелегко, и поэтому требуется глубже продумывать сценарии возникновения аварийных ситуаций. Посмотрите, нельзя ли обойтись без всей этой работы, постаравшись полностью избежать применения самостоятельно сделанных подписей.

Еще один недостаток выражается в том, что трафик, проходящий через SSL, не может быть кэширован обратными прокси-серверами, подобными Varnish или Squid. Это означает, что если вам нужно кэшировать трафик, то это следует сделать либо внутри сервера, либо внутри клиента. Положение дел можно исправить, если завершать SSL-трафик в балансировщике нагрузки и размещать кэш-память за этим балансировщиком.

Нужно также подумать о том, что случится, если используется готовое SSO-решение, подобное SAML, у которого уже есть доступ к именам пользователей и паролям. Хотим ли мы, чтобы наш основной сервис аутентификации использовал тот же набор учетных данных, позволяя иметь один процесс для их выдачи и отзыва? Мы могли бы сделать это посредством сервиса, общающегося с тем же самым сервисом каталогов, который поддерживает SSO-решение. В противном случае придется хранить имена пользователей и пароли самостоятельно внутри сервиса, но тогда появится риск продублированности поведения.

Единственное замечание: при таком подходе серверу известно лишь, что у клиента есть имя пользователя и пароль. Мы не имеем ни малейшего понятия, исходит ли эта информация от той машины, от которой мы ее ожидаем, — она может приходить от кого угодно, находящегося в нашей сети.

Использование SAML или OpenID Connect

Если в качестве схемы аутентификации и авторизации вами уже используется SAML или OpenID Connect, этим можно воспользоваться и для взаимодействия между сервисами. Если используется шлюз, весь внутрисетевой трафик нужно будет также направлять через шлюз, но если каждый сервис справляется с интеграцией своими силами, этот подход должен работать просто замечательно. Преимущество заключается в использовании существующей инфраструктуры и получении возможности централизации всех ваших элементов управления доступом к сервисам в центральном сервере каталогов. Если же нужно предотвратить возможность атаки в пути следования данных, передачу все равно следует направлять через HTTPS.

У клиентов для прохождения аутентификации с помощью провайдера идентификации имеется набор учетных данных, который они и используют, а сервис для принятия решения по детализированной аутентификации получает эту информацию. Это означает, что вам нужны учетные записи для ваших клиентов, которые иногда называют учетными записями сервиса. Этот подход довольно часто используют многие организации. Но следует предупредить: если вы собираетесь создавать учетные записи сервисов, постарайтесь сузить рамки их использования. Продумайте

возможность наличия у каждого микросервиса собственного набора учетных записей. Это упростит отзыв или изменение доступа в случае компрометации учетных данных, поскольку достаточно будет лишь аннулировать набор затронутых учетных данных.

Но есть еще два недостатка. В первую очередь, так же, как и при использовании Basic Auth, требуется обеспечить надежное хранение учетных данных: где должны находиться имя пользователя и пароль? Клиент должен найти безопасный способ хранения этих данных. Вторая проблема заключается в том, что некоторые технологии, имеющиеся в данной области для проведения аутентификации, требуют довольно утомительного программирования. В частности, SAML превращает реализацию клиента в довольно непростое дело. При использовании OpenID Connect выполняемые действия несколько проще, но, как уже говорилось, эта технология пока еще не имеет хорошей поддержки.

Клиентские сертификаты

Еще один подход к идентификации клиента заключается в использовании возможностей протокола безопасности транспортного уровня (Transport Layer Security (TLS)), последователя SSL, для формирования клиентских сертификатов. Здесь у каждого клиента имеется сертификат X.509, используемый при установке связи между клиентом и сервером. Сервер может проверить аутентичность клиентского сертификата, предоставляя твердые гарантии надежности клиента.

Рабочие задачи по управлению сертификатами здесь еще более обременительны, чем при использовании сертификатов на стороне сервера. И дело не только в ряде основных проблем, касающихся создания большего количества сертификатов и управления ими, а скорее в тех сложностях, которые касаются самих сертификатов. И вы должны быть готовы к тому, чтобы потратить много времени на определение причин, по которым сервис не признает то, что, по вашему мнению, будет совершенно допустимым клиентским сертификатом. И тогда, если произойдет самое худшее, нужно еще взять в расчет сложность аннулирования и повторного выпуска сертификатов. Помочь сможет применение групповых сертификатов, но это не решит всех проблем. Эта дополнительная нагрузка означает, что вы будете присматриваться к использованию данной технологии при особой обеспокоенности о конфиденциальности пересылаемых данных или при отправке данных по сети, не находящейся под вашем полным контролем. Поэтому вы, к примеру, можете принять решение о безопасном обмене только теми данными, которые имеют для сторон особую важность при их отправке по Интернету.

HMAC через HTTP

Как уже говорилось, использовать Basic Authentication через обычный HTTP, если вы обеспокоены компрометацией имени пользователя и пароля, не очень разумно. Традиционной альтернативой служит направление трафика через HTTPS, но у этого способа имеется ряд недостатков. Кроме управления сертификатами, издержки

HTTPS-трафика могут стать причиной дополнительной нагрузки на серверы (хотя, если честно, сейчас в таком случае влияние на нагрузку значительно меньше, чем было несколько лет назад), да к тому же имеются сложности с кэшированием такого трафика.

Альтернативный способ подписи запроса, который активно используется API-интерфейсами S3 компании Amazon и является частью OAuth-спецификации, — применение кода проверки подлинности сообщений на основе хеш-функции (HMAC).

При использовании HMAC тело запроса хешируется закрытым ключом и получившийся хеш отправляется вместе с запросом. Затем сервер использует собственную копию закрытого ключа и тело запроса для воссоздания хеша. При совпадении запрос принимается. Приятным обстоятельством является то, что, если где-нибудь на дистанции кто-то проведет какие-либо манипуляции с запросом, хеш не совпадет и сервер будет знать, что запрос был подделан. А закрытый ключ в запросе никогда не пересылается, поэтому он не может быть скомпрометирован в пути! Дополнительным преимуществом является то, что затем трафик может намного легче кэшироваться и издержки на генерирование хешей могут быть намного ниже, чем на обработку HTTPS-трафика (хотя у вас может сложиться и другое мнение).

У этого подхода есть три недостатка. Во-первых, как клиенту, так и серверу нужен общий секрет, который каким-то образом должен быть передан. Как осуществляется его совместное использование? Он должен быть жестко задан на обоих концах, но затем в случае компрометации этого секрета возникнет проблема аннулирования доступа. Если пересылать этот ключ по какому-то альтернативному протоколу, то нужно быть уверенным в том, что этот протокол также обладает весьма высокой степенью безопасности!

Во-вторых, это схема, а не стандарт, поэтому существуют разные способы ее реализации. В результате наблюдается явная нехватка хороших, открытых и практичных реализаций этого подхода. В общем, если этот подход вам интересен, то можете обратиться к дополнительным источникам информации, чтобы разобраться в различных способах его реализации. Я бы предложил просто посмотреть, как он реализован компанией Amazon для ее S3, и скопировать их подход, особенно в использовании рациональной функции хеширования с соответствующим длинным ключом, таким как SHA-256. Стоит также присмотреться к веб-маркерам в формате JSON — JSON web tokens (JWT), поскольку в них реализуется очень похожий подход, который, кажется, набирает обороты. Но при этом имейте в виду, что выполнить все правильно нелегко. Мой коллега работал с командой, которая, занимаясь собственной JWT-реализацией, пропустила одну булеву проверку и сделала недействительным весь свой код аутентификации! Надеюсь, со временем мы увидим реализации библиотек, более пригодные к повторному использованию.

И в-третьих, следует понимать, что этот подход гарантирует только то, что никакая третья сторона не сможет провести манипуляции с запросом и сам закрытый ключ останется закрытым. Все остальные данные в запросе будут по-прежнему видны тем, кто шпионит в сети.

API-ключи

API-ключи используются всеми открытыми API-интерфейсами таких сервисов, как Twitter, Google, Flickr и AWS. API-ключи позволяют сервису идентифицировать того, кто осуществляет вызов, и наложить ограничения на то, что он может сделать. Зачастую ограничения выходят за рамки простого предоставления доступа к ресурсам и могут распространяться на такие действия, как ограничение скорости конкретных абонентов для обеспечения качества предоставления сервиса другим людям.

Когда речь заходит об использовании API-ключей в ваших собственных подходах к обмену данными между микросервисами, конкретный рабочий механизм будет зависеть от используемой технологии. В некоторых системах применяется один общий API-ключ и используется подход, похожий на недавно рассмотренный HMAC. Более распространенный подход заключается в применении пары из открытого и закрытого ключей. Обычно управление ключами осуществляется централизованно, так же, как мы бы централизованно управляли определением идентичности людей. В данной области очень популярна модель шлюза.

Частично популярность API-ключей обусловлена тем фактом, что их применение для программ совсем не сложно. В сравнении с обработкой SAML-квитирования аутентификация на основе API-ключа намного проще и понятнее.

Конкретные возможности систем сильно отличаются друг от друга, и у вас есть несколько вариантов как в коммерческой области, так и в области программ с открытым кодом. Некоторые средства просто обрабатывают обмен API-ключами и выполняют некоторые основные функции управления ключами. Другие средства предлагают все, вплоть до включения ограничения скорости трафика, монетизации, API-каталогов и систем обнаружения.

Некоторые API-системы позволяют создавать мост от API-ключей к существующим сервисам каталогов. Это дает возможность выпускать API-ключи для принципалов (представленных людьми или системами) в вашей организации и следить за жизненным циклом этих ключей примерно так же, как вы бы управляли их обычными учетными данными. Это дает возможность разрешать доступ к вашим сервисам различными способами, но при сохранении единого источника достоверности, например с использованием SAML при аутентификации людей для SSO и API-ключей — при обмене данными между сервисами (рис. 9.2).

Проблема помощника

Аутентификация принципала с помощью отдельно взятого микросервиса осуществляется довольно просто. А что получится, если этому сервису для завершения операции понадобится совершить дополнительные вызовы? Посмотрите на рис. 9.3, где показан сайт интернет-магазина MusicCorp. Наш интернет-магазин представляет собой написанный на JavaScript пользовательский интерфейс, работающий в среде браузера. Он совершает вызовы приложения магазина, находящегося на серверной стороне, используя схему внутренних интерфейсов для внешних интерфейсов, которая рассматривалась в главе 4. Вызовы, осуществляемые между браузером и сервером, могут быть аутентифицированы с помощью SAML, OpenID Connect или подобных им технологий. Пока нас все устраивает.

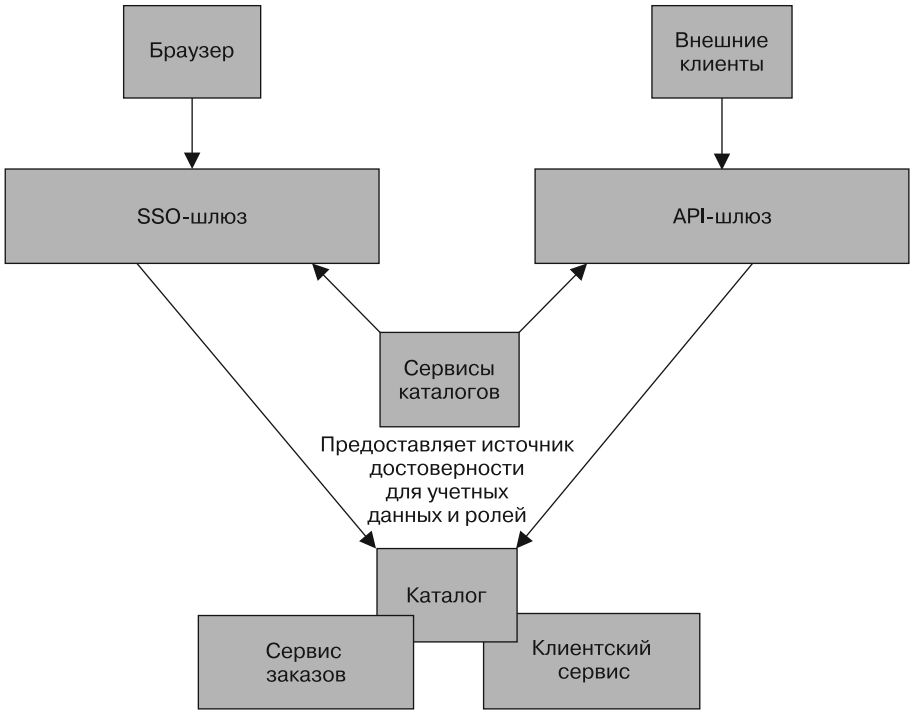


Рис. 9.2. Использование сервисов каталогов для синхронизации информации о принципалах между SSO и API-шлюзом

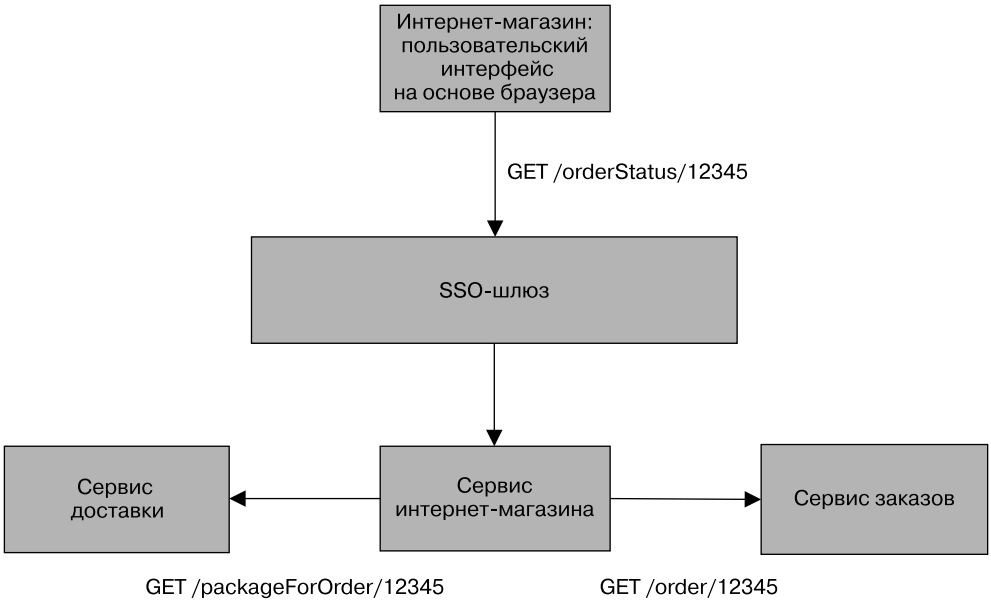


Рис. 9.3. Пример создания ситуации, при которой можно обмануть помощника

После регистрации я могу щелкнуть на ссылке и просмотреть подробности заказа. Для отображения информации следует извлечь исходный заказ из сервиса заказов, но, кроме этого, нужно просмотреть информацию о доставке заказа. Следовательно, щелчок на ссылке /orderStatus/12345 заставит интернет-магазин при запросе подробностей инициировать вызов из сервиса интернет-магазина как к сервису заказов, так и к сервису доставок. Но примут ли эти нижестоящие сервисы вызовы от интернет-магазина? Можно было бы принять установку подразумеваемого доверия, поскольку вызов поступил в границах периметра, что вполне допустимо. Можно было бы даже воспользоваться сертификатами или API-ключами, чтобы подтвердить, что информацию действительно запрашивает интернет-магазин. Но будет ли этого достаточно?

Здесь мы сталкиваемся с уязвимостью, которая называется *проблемой обманутого помощника*, что в контексте межсервисной связи относится к ситуации, при которой злоумышленник может обмануть сервис-помощник, заставив делать за него вызовы к нижестоящему сервису, на что он не должен быть способен. Например, в качестве клиента после входа в систему интернет-магазина я могу просмотреть подробности своей учетной записи. А что, если я смогу обмануть пользовательский интерфейс интернет-магазина и заставлю его сделать запрос на получение чьих-либо данных, возможно, осуществлением вызова с теми данными учетной записи, с которыми я входил в систему?

Что в данном примере остановит меня от вопросов относительно заказов, не имеющих ко мне никакого отношения? Раз уж я вошел в систему, то могу отправлять запросы, касающиеся чужих, не моих заказов, чтобы посмотреть, не могу ли я извлечь какую-либо полезную для себя информацию. Если такое произойдет, можно попытаться найти защиту внутри самого интернет-магазина, проверяя, какой заказ кому принадлежит, и выдавая отказ, если кто-нибудь станет запрашивать то, что не должен получать. Но если у нас имеется множество различных приложений, выявляющих эту информацию, то потенциально эта логика может быть продублирована в множестве мест.

Можно направлять запросы непосредственно из пользовательского интерфейса к сервису заказов и позволять ему проверять допустимость запроса, но тогда мы столкнемся с проявлением тех недостатков, которые рассматривались в главе 4. В качестве альтернативы при отправке интернет-магазином запроса сервису заказов может сообщаться не только какой именно заказ нужен, но и то, от чьего имени он запрашивается. Некоторые схемы аутентификации позволяют передавать нижестоящим сервисам исходные учетные данные принципала, хотя при использовании SAML это сродни какому-то кошмару с участием вложенных SAML-утверждений, который технически достижимы, но связаны с такими трудностями, что этим никто не хочет заниматься. Конечно же, это может усложниться еще больше. Представьте себе, что будет, если интернет-магазин, в свою очередь, востребует совершение дополнительных нисходящих вызовов. Насколько глубоко мы должны пойти в проверке допустимости для всех этих помощников?

К сожалению, данная проблема в силу своей сложности не имеет легких решений. Но вам следует знать о ее существовании. В зависимости от степени

уязвимости рассматриваемой операции вам, возможно, придется выбирать между подразумеваемым доверием, проверкой идентичности вызывающей стороны или запросом у вызывающей стороны учетных данных исходного принципала.

Безопасность данных, находящихся в покое

Мы несем ответственность и за данные, расположенные в разных местах, особенно если речь идет о конфиденциальных данных. Будем надеяться на то, что мы сделали все возможное, чтобы предотвратить доступ злоумышленников в нашу сеть, а также на то, что они не могут взломать наши приложения или операционные системы, чтобы получить доступ к основной закрытой информации. Но мы должны быть готовы к их атакам, и помочь в этом сможет глубоко эшелонированная оборона.

Многие заметные взломы систем безопасности направлены на то, чтобы в руки злоумышленников попали данные, находящиеся в состоянии покоя, и чтобы злоумышленник мог прочитать их. Это случается либо из-за того, что данные хранились в незашифрованной форме, либо из-за того, что механизм, используемый для их защиты, имел существенный недостаток.

Существует множество различных механизмов, позволяющих защитить конфиденциальную информацию, но независимо от того, какой из них будет выбран, существует ряд общих положений, которые нужно учитывать.

Пользуйтесь хорошо известными средствами

Проще всего загубить шифрование данных попыткой реализации собственных алгоритмов шифрования или каких-то совершенно иных приемов шифрования. Независимо от используемого языка программирования вам нужно обращаться к пересматриваемым, постоянно совершенствующимся и заслужившим хорошую репутацию реализациям алгоритмов шифрования. Пользуйтесь именно такими алгоритмами! И подпишитесь на рассылки и консультации по выбранной вами технологии, чтобы быть уверенными в том, что обо всех уязвимостях вы узнаете сразу же, как только их обнаружат, и в том, что вы пользуетесь исправленными и обновленными версиями средств, являющихся реализацией данной технологии.

Что касается шифрования данных, находящихся в покое, то, если у вас нет весьма веских аргументов в пользу чего-либо другого, остановите свой выбор на широко известных реализациях AES-128 или AES-256, предназначенных для вашей платформы[1]. Реализации AES, которые с высокой долей вероятности были

[1] В принципе, длина ключа увеличивает объем работы, требующейся для его взлома. Поэтому можно предположить, что чем длиннее ключ, тем лучше защищены ваши данные. Но весьма уважаемым специалистом в области обеспечения безопасности Брюсом Шнайером (Bruce Schneier) в отношении AES-256 для конкретных типов ключей были выявлены некоторые незначительные проблемы. Это одна из тех областей, в которых в ходе чтения данной книги нужно дополнительно просмотреть все последние советы специалистов!

всесторонне протестированы (и хорошенько подправлены), имеются в библиотеках времени выполнения как у Java, так и у .NET-технологии. Существуют и отдельные библиотеки для большинства других платформ, например библиотеки Bouncy Castle для Java и C#.

Что касается паролей, вам следует присмотреться к использованию технологии под названием *«хеширование паролей со случайным набором символов»*.

Плохо реализованное шифрование еще хуже, чем его полное отсутствие, поскольку ложное чувство защищенности может привести к потере бдительности.

Все зависит от ключей

До сих пор речь шла о том, что шифрование зависит от реализации алгоритма, получающей шифруемые данные и ключ, после чего выполняющей шифрование данных. А где же хранятся сами ключи? Если я шифрую данные, переживая за то, что кто-нибудь может похитить всю мою базу данных, и храню ключи в той же самой базе данных, то поставленной цели я явно не добьюсь! Поэтому ключи нужно хранить в каком-нибудь другом месте. Но где именно?

Одним из решений может стать использование для шифровки и расшифровки данных отдельного безопасного устройства. Еще одно решение — применение отдельного хранилища ключей, которым ваш сервис может воспользоваться, когда ему понадобится ключ. Важнейшей операцией может стать управление жизненным циклом ключей (и доступ к их изменению), и справиться с этой операцией за вас могут соответствующие системы.

В некоторые системы управления базами данных даже включена встроенная поддержка шифрования, например Transparent Data Encryption в SQL Server, которая нацелена на выполнение данных функций незаметно для пользователей. Даже если выбранная вами база данных способна на подобные действия, разберитесь, как происходит обработка ключей, и выясните, ослабевает ли в результате этого угроза, от которой вы стараетесь защититься.

Еще раз хочу напомнить, что это дело непростое. Не пытайтесь реализовать собственный вариант и хорошенько изучите доступные средства!

Выберите защищаемые объекты

Установка на шифрование абсолютно всех данных может в каком-то смысле упростить ситуацию. Не нужно будет гадать, что стоит, а что не стоит защищать. Но вам все же придется думать о том, какие данные следует помещать в файлы журналов, чтобы проще было разбираться с проблемами. К тому же вычислительные издержки от шифрования всех данных могут стать весьма обременительными, что в результате выльется в необходимость использования более мощного оборудования. Все еще больше усложнится, когда частью ваших схем разбиения функциональных возможностей на более мелкие части станет применение миграций баз данных. В зависимости от характера вносимых изменений могут потребоваться расшифровка данных, их миграция и повторная шифровка.

Разбив вашу систему на большее количество узкоспециализированных сервисов, можно было бы определить единое хранилище данных, которое могло бы быть зашифровано целиком, но сомнительно, что из этого что-либо вышло бы. Более разумно будет зашифровать известный набор таблиц.

Расшифровка по требованию

Шифруйте данные при первой же возможности. Выполняйте их расшифровку только по требованию и сделайте так, чтобы они не могли храниться где-либо еще.

Шифровка резервных копий

Резервное копирование данных считается правилом хорошего тона. Нам необходимо выполнять резервное копирование особо важных данных, и те данные, которые определены как наиболее ценные и подлежащие шифрованию, в силу своей важности также достойны резервного копирования! Казалось бы, это очевидно, но мы все же должны убедиться в том, что резервные копии тоже зашифрованы. Это означает, что нам следует знать, какие ключи понадобятся для обработки той или иной версии данных, особенно если эти ключи меняются. Здесь проявляется особая важность наличия четкой системы управления ключами.

Глубоко эшелонированная оборона

Как уже упоминалось, мне не нравится класть все яйца в одну корзину. Я имею в виду глубоко эшелонированную оборону. Мы уже говорили об обеспечении безопасности передаваемых данных и о защите данных, находящихся в покое. Но можем ли мы повысить безопасность, применив какие-то другие средства защиты?

Брандмауэры

Весьма разумной мерой предосторожности является применение одного или нескольких брандмауэров. Некоторые из них устроены весьма просто и позволяют лишь ограничить доступ определенным типам трафика с помощью конкретных портов. Другие способны на более изощренные действия. ModSecurity, к примеру, является разновидностью приложения-брандмауэра, которое может содействовать дросселированию связи с конкретными диапазонами IP-адресов и определять разновидности попыток взлома.

Имеет смысл воспользоваться более чем одним брандмауэром. Например, вы можете принять решение обеспечивать безопасность хоста локальным использованием таблиц IP-адресов — IPTables, настроив допустимые входы и выходы. Эти правила могут быть привязаны к локально запущенным сервисам, а по периметру может быть выставлен брандмауэр, контролирующий общий доступ.

Регистрация

Качественная регистрация и особенно возможность объединения данных регистрационных журналов нескольких систем не относится к предохранительным мерам, но может способствовать определению причин и ликвидации последствий всевозможных негативных происшествий. Например, после применения исправлений в системе безопасности в журналах зачастую можно наблюдать случаи использования людьми конкретных уязвимостей. Внесение исправлений гарантирует невозможность повторения подобных случаев, но, если это уже произошло, вам может понадобиться перейти в режим восстановления данных. Доступность журналов позволяет отследить наступление неблагоприятных событий.

Но при этом следует иметь в виду, что к информации, которую мы сохраняем в журналах, нужно относиться со всей ответственностью! Конфиденциальную информацию следует отсеивать, чтобы гарантировать отсутствие утечки через журналы важных данных, которые могут стать отличной целью для взломщиков.

Система обнаружения (и предотвращения) вторжений

Система обнаружения вторжений (IDS) может вести мониторинг сетей или хостов с целью обнаружения подозрительного поведения и оповещения о проблемах по мере их возникновения. Система предотвращения вторжений (IPS) вдобавок к мониторингу подозрительных действий способна вмешиваться в них, не давая им совершиться. В отличие от брандмауэра, который в первую очередь анализирует окружающую обстановку, препятствуя проникновению вредоносного кода, IDS и IPS активно выискивают подозрительное поведение внутри периметра. Когда все начинается с нуля, благоразумнее устанавливать IDS. Эти системы основаны на эвристическом анализе, как и многие приложения-брандмауэры, и вполне возможно, что универсальный стартовый набор правил будет либо слишком мягким, либо недостаточно мягким по отношению к поведению вашего сервиса.

Использование более пассивной IDS-системы для предупреждения о возникающих проблемах может стать неплохим способом настройки правил, прежде чем вы перейдете к использованию более активных возможностей.

Обособление сетей

При использовании монолитной системы возможности структурирования сетей для предоставления дополнительной защиты ограничены. Но, применяя микросервисы, вы можете поместить их в разных сегментах сети, чтобы получить дополнительный контроль над тем, как сервисы общаются друг с другом. К примеру, AWS дает возможность автоматического предоставления виртуального закрытого облака (VPC), позволяющего хостам находиться в отдельных подсетях. Затем можно указать, какие VPC-облака могут видеть друг друга, установив правила пиринга, и даже направлять трафик через шлюзы для доступа к прокси-серверу, очерчивая

фактически несколько периметров, на которых могут быть предприняты дополнительные меры безопасности.

Это дает вам возможность сегментировать сети на основе командной принадлежности или, возможно, степени риска.

Операционная система

Работа систем зависит от большого количества программных средств, создаваемых не нами, и может иметь уязвимости с точки зрения безопасности, которые способны отразиться на нашем приложении (имеются в виду операционные системы и другие вспомогательные средства, под управлением которых оно работает). Здесь вам может пригодиться ряд рекомендаций общего плана. Начните с запуска сервисов исключительно в качестве пользователей операционной системы, имеющих как можно меньше прав доступа, чтобы при компрометации их учетной записи мог быть нанесен минимальный вред.

Затем вносите исправления в свои программные средства. И делайте это регулярно. Этот процесс должен быть автоматизирован, и вы должны быть осведомлены о том, что ваши машины не синхронизированы с самыми последними пакетами исправлений. Здесь могут пригодиться такие средства, как SCCM от компании Microsoft или Spacewalk от RedHat, которые могут оповестить вас о том, что на машине не установлены самые последние исправления, и, если требуется, инициировать обновление программных средств. Если используются такие средства, как Ansible, Puppet или Chef, то проблем с автоматизированным получением изменений у вас, скорее всего, нет, поскольку эти средства могут успешно решать эту задачу, но абсолютно все за вас они делать не будут.

Именно такие средства и нужно использовать, но, как ни удивительно, мне довольно часто приходилось видеть, как весьма важные программные средства запускались на старых операционных системах без внесенных в них последних исправлений. Вы можете использовать наиболее известную и самую защищенную в мире систему безопасности на уровне приложения, но если при этом на вашей машине в качестве основы запущена старая версия веб-сервера с неисправленной ошибкой переполнения буфера, система по-прежнему будет слишком уязвимой.

Если вы используете Linux, то нужно обратить внимание также на наличие модулей безопасности для самой операционной системы. К примеру, AppArmour позволяет определить ожидаемое поведение вашего приложения и в нем имеется ядро, которое будет постоянно следить за приложением. Как только оно начнет делать что-нибудь, чем не должно заниматься, ядро вмешается в его работу. Кроме AppArmour, имеется также SeLinux. Хотя с технической точки зрения оба модуля могут работать на любой современной Linux-системе, на практике некоторые дистрибутивы поддерживают один из них лучше, чем другой. К примеру, AppArmour используется по умолчанию в Ubuntu и SuSE, а SELinux традиционно хорошо поддерживается дистрибутивом RedHat. Самым последним вариантом таких модулей является GrSSecurity, разработанный с прицелом на то, чтобы стать проще в использовании, чем AppArmour или GrSecurity, и с попыткой расширения до их

возможностей, но ему для работы требуется собственное ядро. Я бы советовал присмотреться ко всем трем модулям, чтобы понять, который из них вам больше подойдет, но мне нравится идея использования в работе еще одного уровня защиты и профилактики.

Рабочий пример

Наличие системной архитектуры, состоящей из узкоспециализированных сервисов, дает нам больше свободы в реализации системы безопасности. Для тех частей, которые работают с наиболее конфиденциальной информацией или предоставляют наиболее полезные возможности, можно избрать оснащение с наиболее жесткими мерами безопасности. Насчет же других частей системы степень своего беспокойства можно значительно снизить.

Вернемся к проекту MusicCorp, объединив ряд концепций, чтобы посмотреть, где и в какой степени следует применить ряд технологий обеспечения безопасности. Основное внимание мы уделим обеспечению безопасности передаваемых и просто хранящихся данных. На рис. 9.4 показан поднабор всей системы, подлежащий анализу и страдающий от нашего вопиющего невнимания к вопросам его безопасной работы. Все данные здесь отправляются с использованием простого старого протокола HTTP.

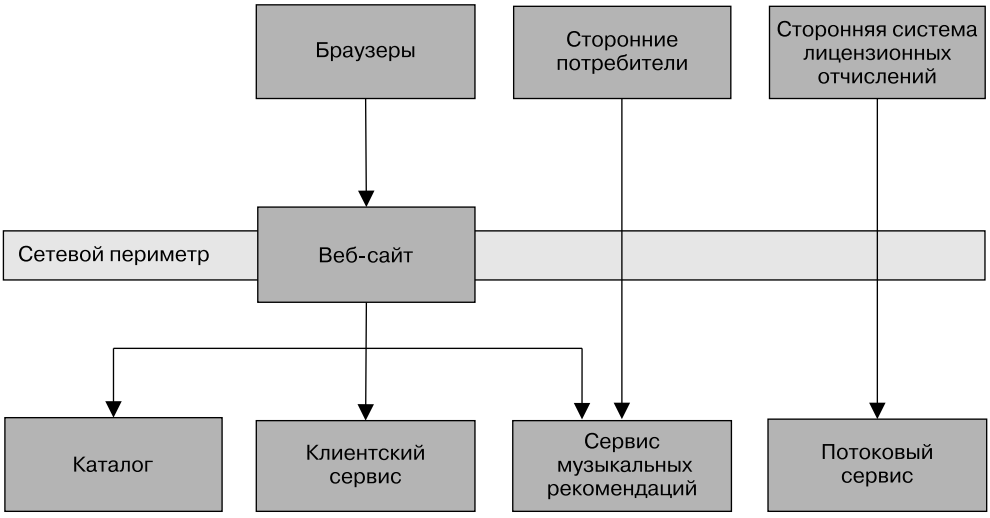


Рис. 9.4. Поднабор проекта MusicCorp, который, к сожалению, не обладает безопасной архитектурой

Здесь имеются стандартные браузеры, которые наши клиенты используют для осуществления покупок в магазине. Здесь также вводится концепция стороннего шлюза лицензионных отчислений: мы начали работать со сторонней компанией, которая будет заниматься лицензионными отчислениями для нашего нового пото-

кового сервиса. Она выходит с нами на связь, чтобы забрать записи о том, какая музыка и когда была востребована потоковым сервисом, то есть получить информацию, которую мы старательно скрываем от конкурирующих компаний. И наконец, мы выставляем напоказ данные нашего каталога для других сторонних организаций, позволяя им, к примеру, вставлять в сайты музыкальных ревю метаданные об артистах и песнях. Внутри нашего сетевого периметра имеется несколько сотрудничающих сервисов исключительно для внутреннего потребления.

В браузере мы будем использовать сочетания стандартного HTTP-трафика для незащищенного содержания, чтобы ему было позволено находиться в кэше. Все надежно защищенное содержимое безопасных, прошедших регистрацию страниц будет отправляться по протоколу HTTPS, который обеспечит клиентам дополнительную защиту, если они будут пользоваться, к примеру, открытыми WiFi-сетями.

Что же касается сторонней системы лицензионных отчислений, то нас волнует не только характер выставляемых данных, но и легитимность получаемых запросов. Здесь мы настаиваем на том, чтобы сторонние партнеры пользовались клиентскими сертификатами. Все данные отправляются по надежному зашифрованному каналу, увеличивая возможность убедиться в том, что их запросил известный нам человек. Нам, конечно же, нужно подумать и о том, что произойдет, если данные выйдут из-под нашего контроля. Будет ли наш партнер заботиться о них так же тщательно, как и мы?

Что касается снабжения данными каталога, нам бы хотелось, чтобы эта информация распространялась как можно шире, облегчая людям покупку нашей музыки! Но нам не хочется, чтобы этой возможностью злоупотребляли, а также хочется знать, кто использует наши данные. В этом случае наиболее подходящими для нас будут API-ключи.

Внутри сетевого периметра все обстоит немного тоньше. Как противодействовать тем, кто пытается пробраться в нашу сеть? В идеале нам хотелось бы как минимум воспользоваться протоколом HTTPS, но управлять им довольно хлопотно. Вместо этого мы решили (хотя бы на первых порах) поместить всю работу в прочный сетевой периметр, имеющий настроенный соответствующим образом брандмауэр и выбирающий для отслеживания вредоносного трафика (например, сканирования портов или атак, вызывающих отказ от обслуживания) подходящее оборудование или программные приспособления для обеспечения безопасности.

При этом мы побеспокоились о некоторых наших данных и о тех местах, где они находятся. Нас не волнуют данные сервиса каталогов, ведь мы же сами захотели, чтобы они попали в общее пользование, и даже предоставили для этого соответствующий API-интерфейс! Но нас очень беспокоит безопасность клиентских данных. Поэтому мы решили зашифровать данные, хранящиеся в клиентском сервисе, и расшифровывать их при чтении. Если взломщики проберутся в нашу сеть, они смогут выдавать запросы к нашему API-интерфейсу клиентской службы, но текущая реализация не позволит извлекать клиентские данные в больших объемах. Если же реализация позволяет подобные действия, то лучше рассмотреть возможность использования для защиты информации клиентских сертификатов. Даже если взломщики проникнут в машину и в запущенную на ней базу данных

и распорядятся скачать все ее содержимое, им понадобится доступ к ключу, чтобы расшифровать данные, которыми они собираются воспользоваться.

На рис. 9.5 показана окончательная картина происходящего. Как видите, наш технологический выбор основан на понимании характера защищаемой информации. Возможно, у вас будут совершенно иные взгляды на обеспечение безопасности используемой архитектуры, тогда вы сможете остановиться на решениях, не похожих на эти.

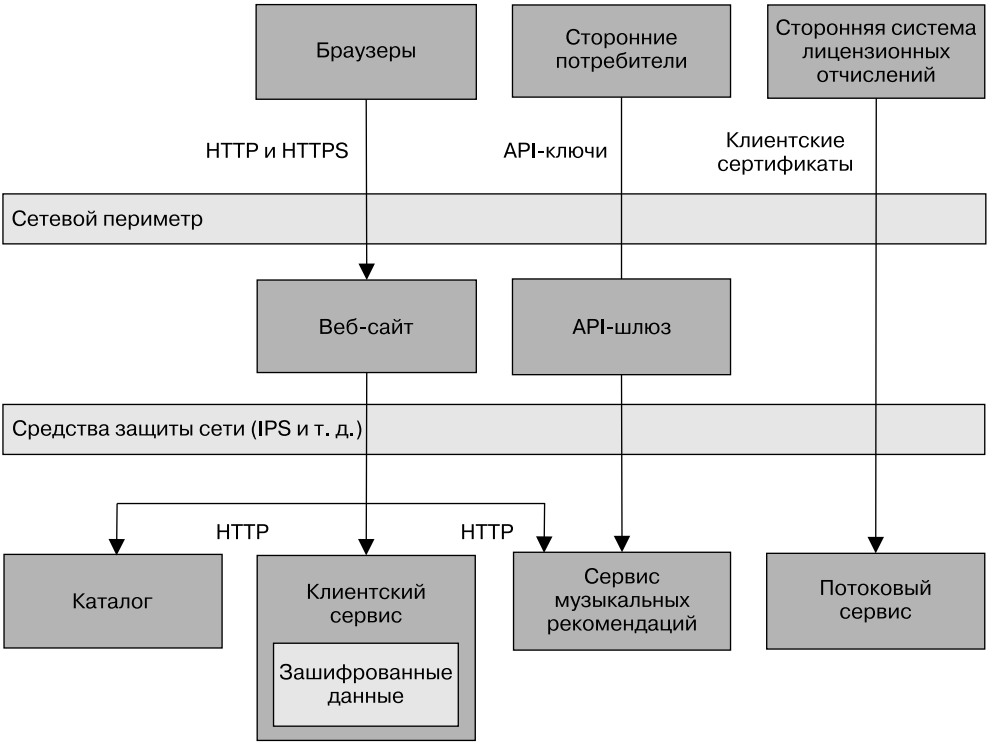


Рис. 9.5. Более безопасная система, используемая в проекте MusicCorp

Проявляйте сдержанность

По мере удешевления дискового пространства и роста возможностей баз данных получать и сохранять большие объемы информации становится все проще. Эти данные имеют ценность не только для бизнеса как такового, который все чаще рассматривает их в качестве ценного актива, но в равной степени и для пользователей, заботящихся о своей неприкосновенности. К данным, относящимся к индивидууму или позволяющим получить о нем информацию, нужно относиться с особым вниманием.

А что, если нам упростить ситуацию? Почему бы не стереть как можно больше личной информации и не сделать это как можно раньше? Нужно ли нам при реги-

страции запроса пользователя навечно сохранять весь IP-адрес или можно заменить последние несколько цифр символами «x»? Нужно ли сохранять чьи-то имя, возраст, пол и дату рождения, чтобы выложить предложения о покупке товара, или достаточно будет информации о примерном возрасте и почтового индекса?

Положительно ответив на эти вопросы, можно получить множество преимуществ. Во-первых, если не хранить эти данные, то никто их не сможет украсть. Во-вторых, если они у вас не хранятся, то никто (например, какое-нибудь правительственное агентство) не сможет их запросить!

Эту концепцию хорошо поясняет немецкий термин Datensparsamkeit (означающий минимизацию данных). Появившись в немецком законодательстве о конфиденциальности, он является воплощением концепции хранения только той информации, которая необходима для выполнения бизнес-операций или соответствия местным законам. Вполне очевидно, что это противоречит тенденции к хранению все более солидных объемов информации, но это толчок к пониманию того, что противоречие все же существует!

Человеческий фактор

Основная часть того, о чем здесь говорилось, касается основ реализации технологических мер безопасности для защиты ваших систем и данных от внешних взломщиков-злоумышленников. Но вам также нужны процессы и политики, имеющие дело с человеческим фактором в вашей организации. Как аннулировать доступ к учетным данным, когда кто-нибудь уйдет из организации? Как защититься от социальной инженерии? В качестве неплохого умственного упражнения рассмотрим, какой ущерб может нанести раздосадованный бывший сотрудник вашей системе, если ему захочется это сделать. Для того чтобы придумать защитные меры, неплохо будет поставить себя на место потенциального вредителя, ведь никакие злоумышленники не знают вашу организацию изнутри так хорошо, как бывший сотрудник!

Золотое правило

Если вам больше нечего почерпнуть из данной главы, возьмите хотя бы это: не создавайте собственную систему шифрования. Не изобретайте собственные протоколы системы защиты. Вы все равно не сможете изобрести ничего путного в шифровании или разработке криптографических средств защиты, если только вы не специалист по криптографии с многолетним стажем. Но, даже являясь специалистом в этой области, вы все равно можете сделать что-нибудь не так.

Многие из ранее упомянутых средств, подобных AES, представляют собой весьма зрелые промышленные технологии, в основу которых положены алгоритмы, прошедшие коллегиальное рецензирование, программная реализация которых тщательно тестировалась и исправлялась в течение многих лет. Эти средства вполне успешно справляются со своими задачами! Зачастую изобретение колеса превращается в пустую трату времени, но, когда дело касается безопасности, такая изобретательская деятельность может стать весьма рискованным занятием.

Создание системы безопасности

Как и в случае с автоматизированным функциональным тестированием, мы не хотим, чтобы вопросами безопасности занимались разные группы людей и делали это в последнюю очередь. Главное — помочь разработчикам в разъяснении проблем безопасности, поскольку подъем общего уровня информированности каждого разработчика о проблемах безопасности может в первую очередь сократить количество таких проблем. Для начала неплохо бы ознакомить людей с десятью самыми значимыми уязвимостями из открытого проекта обеспечения безопасности веб-приложений — списком OWASP Top Ten и средой тестирования безопасности — OWASP Security Testing Framework. Специалисты, конечно, народ занятой, но если у вас есть с ними деловой контакт, то попросите их помочь в данном вопросе.

Существует ряд автоматизированных инструментальных средств, способных испытать вашу систему на наличие уязвимостей, например на возможность атак с применением межсайтовых сценариев. Хорошим примером может послужить средство Zed Attack Proxy, известное также как ZAP. Будучи проинформированным о результатах работы, ZAP предпринимает попытки воссоздания вредоносных атак на ваш сайт. Существуют и другие средства, использующие статический анализ с целью поиска распространенных ошибок программирования, которые могут стать прорехами в системе безопасности, например Brakeman для Ruby. Если эти средства легко интегрировать в обычные CI-сборки, их следует включить в стандартные проверочные процедуры. Другие виды автоматизированного тестирования — более сложная процедура. Например, использование средств, подобных Nessus, для сканирования на предмет обнаружения уязвимостей происходит немного сложнее и может потребовать для интерпретации результатов человеческого вмешательства. Тем не менее эти тесты все же поддаются автоматизации, и, может быть, имеет смысл запускать их на том же этапе, что и нагрузочные тесты.

Хорошие разновидности моделей создания систем безопасности для команд поставки имеются также у разработанного компанией Microsoft процесса создания безопасных программных средств — Security Development Lifecycle. Некоторые аспекты этого процесса кажутся излишними, но вам нужно в нем разобраться, чтобы понять, какие из аспектов вписываются в ваш рабочий процесс.

Внешняя проверка

Я полагаю, что в вопросах обеспечения безопасности важную роль может сыграть возможность дать внешнюю оценку проделанной работе. Такие действия, как тест на проникновение, проводимые внешней стороной, фактически имитируют реальные попытки взлома. Тем самым можно будет взглянуть на работу со стороны и заметить допущенные командой ошибки, которые порой из-за близости проблемы к команде ей самой не видны. Если ваша компания довольно велика, в ней может существовать специально выделенная команда, занимающаяся безопасностью информационных систем, которая может помочь в этом деле. Если такой

команды у вас нет, подыщите внешних исполнителей, способных выполнить эту работу. Обратитесь к ним заранее, выясните, как они хотели бы работать и сколько дней им нужно потратить на тестирование.

Следует также определить объем проверочной работы, который требуется выполнить перед каждым выпуском. Как правило, для небольших дополнительных выпусков проводить, к примеру, полный тест на проникновение не нужно, а при больших изменениях в выпусках он может оказаться весьма кстати. Ваши потребности зависят от выбранной степени допустимого риска.

Резюме

Итак, мы опять возвращаемся к основной теме книги, в которой утверждается, что наличие системы, разбитой на узкоспециализированные сервисы, дает нам множество вариантов решения проблемы. Возможность использования микросервисов не только позволяет уменьшить влияние любой отдельной бреши в системе безопасности, но и дает нам больше возможностей найти компромиссы в отношении издержек, связанных с более сложными и безопасными подходами в работе с конфиденциальными данными, и выбрать менее сложные подходы в тех случаях, когда риски оцениваются значительно ниже.

Как только вы осознаете уровни угроз для различных частей системы, вы должны приступить к осмыслению того, когда следует рассматривать вопросы обеспечения безопасности: при передаче данных, их хранении или ни в одном из этих процессов.

Наконец, вам следует осознать важность глубоко эшелонированной обороны и обеспечить постоянное внесение исправлений в используемую операционную систему. Но даже если вы считаете себя непревзойденным специалистом, не следует пытаться реализовать собственную криптографическую систему!

Если вы желаете ознакомиться с общим обзором решения проблем безопасности для приложений, создаваемых на основе использования браузера, то для начала хорошим подспорьем станет некоммерческий проект обеспечения безопасности веб-приложений Open Web Application Security Project (OWASP), в рамках которого постоянно обновляется документ, содержащий описание десяти наиболее существенных угроз безопасности, — Top 10 Security Risk, который должен стать настольным пособием для каждого разработчика. И наконец, если у вас есть желание получить более развернутое представление о криптографии, присмотритесь к вышедшей в издательстве Wiley книге *Cryptography Engineering* Нильса Фергюсона (Niels Ferguson), Брюса Шнайера (Bruce Schneier) и Тадаёси Коно (Tadayoshi Kohno).

Освоение мер безопасности зачастую зависит от сознательности людей и приемов их работы с нашими системами. Один еще не рассмотренный с точки зрения использования микросервисов аспект касается взаимодействия организационных структур и самих архитектур. Но, как и в вопросах обеспечения безопасности, мы увидим, что игнорирование человеческого фактора может стать весьма серьезной ошибкой.

10 Закон Конвея и проектирование систем

Пока что основной материал книги был сфокусирован на технических проблемах перехода к архитектуре с использованием узкоспециализированных сервисов. Но есть еще и организационные вопросы, которые также стоит рассмотреть. В этой главе речь пойдет о том, что вы на свой страх и риск игнорируете организационную структуру своей компании!

Наша отрасль производства еще сравнительно молода, и в ней, похоже, постоянно что-то обновляется. И лишь несколько основных законов выдержали испытание временем. Например, закон Мура, утверждающий, что плотность транзисторов в интегральных микросхемах удваивается каждые два года, оказался невероятно точен (хотя есть люди, предрекающие скорое замедление этой тенденции). На мой взгляд, гораздо полезнее в моей повседневной деятельности оказался другой закон, справедливый практически во всех областях, — это закон Конвея.

В статье Мелвина Конвея (Melvin Conway) *How Do Committees Invent*, опубликованной в журнале Datamation в апреле 1968 года, было замечено: *«Организации, проектирующие системы (здесь имеется в виду более широкое толкование, включающее не только информационные системы), неизбежно производят конструкцию, чья структура является копией структуры взаимодействия внутри самой организации»*.

Зачастую это высказывание цитируется в различных формах как закон Конвея. Эрик С. Раймонд (Eric S. Raymond) дал краткое изложение этого феномена в книге *The New Hacker's Dictionary* (MIT Press), заявив следующее: *«Если над компилятором работают четыре группы, то вы получите компилятор, работающий в четыре прохода»*.

Доказательства

История гласит, что, когда Мелвин Конвей отправил свою статью на эту тему в журнал Harvard Business Review, редакция ее не приняла, заявив, что его утверждение ничем не подкреплено. Я же наблюдал подтверждение этой теории на практике во многих различных ситуациях настолько часто, что принял ее за истину. Но вам не нужно принимать мои слова на веру: после того как Конвей вы-

вел эту теорию, в данной области был проделан большой объем работы. Для изучения взаимосвязанности организационной структуры и создаваемой ею системы проведен целый ряд исследований.

Организации со слабыми и сильными связями

В исследовании *Exploring the Duality Between Product and Organizational Architectures* (Harvard Business School) Алан Маккормак (Alan MacCormack), Джон Руснак (John Rusnak) и Карлисс Болдуин (Carliss Baldwin) рассмотрели несколько различных программных систем, которые были приблизительно классифицированы как созданные организациями со связями, имеющими либо слабый, либо сильный характер. Думаю, что в организациях с сильными связями коммерческий продукт обычно подкрепляется четкой выверенностью целей и представлений, а организации со слабыми связями хорошо представлены распределенными сообществами, разрабатывающими продукты с открытым кодом.

Проводя исследования, в ходе которых сравнивались пары схожих продуктов от организаций каждого из этих типов, авторы пришли к выводу, что организации с менее сильными связями обычно создают системы с большей модульностью и меньшей степенью связанности, а организации с более сильными связями — программные средства, обладающие меньшей модульностью.

Windows Vista

Компания Microsoft, изучая влияние своей организационной структуры на качество конкретного продукта, а именно Windows Vista, провела собственное эмпирическое исследование. В частности, исследователи проанализировали несколько факторов с целью определения надежности отдельно взятого компонента системы[1]. После изучения ряда показателей, включая такой часто используемый показатель качества, как сложность кода, они пришли к выводу, что показатели, связанные с организационной структурой, оказались статистически наиболее значимыми оценками.

Следовательно, у нас есть еще один пример влияния организационной структуры на характеристики системы, создаваемой данной организацией.

Netflix и Amazon

Наверное, идея обязательной согласованности организации и архитектуры может быть неплохо проиллюстрирована на примере Amazon и Netflix. В Amazon довольно рано начали понимать преимущества владения командами полным жизненным циклом управляемых ими систем. Там решили, что команды должны всецело распоряжаться теми системами, за которые они отвечают, управляя всем жизненным циклом этих систем. Но в Amazon также знали, что небольшие команды могут работать быстрее больших. Это привело к созданию команд, которые можно было бы

[1] И как всем нам известно, похвастаться особой надежностью Windows Vista не могла!

накормить двумя пиццами. Это стремление к созданию небольших команд, владеющих полным жизненным циклом своих сервисов, и стало основной причиной того, что в Amazon была разработана платформа Amazon Web Services. Для обеспечения самодостаточности своих команд компании понадобилось создать соответствующий инструментарий.

Этот пример был взят на вооружение компанией Netflix и с самого начала определил формирование ее структуры вокруг небольших независимых команд, образуемых с прицелом на то, что создаваемые ими сервисы также будут независимы друг от друга. Тем самым обеспечивалась оптимизация скорости изменения архитектуры систем. Фактически в Netflix разработали организационную структуру для желаемой архитектуры создаваемых систем.

Так что же со всем этим делать?

Итак, все доказательства, как анекдотические, так и эмпирические, указывают на то, что организационная структура оказывает сильное влияние на характер (и качество) создаваемых нами систем. Но как это понимание сможет нам помочь? Рассмотрим несколько различных организационных ситуаций и разберемся, какое влияние каждая из них может оказать на конструкцию систем.

Адаптация к направлениям обмена данными

Для начала рассмотрим простую единственную команду. Она отвечает за все аспекты разработки и реализации системы. И может часто организовывать узкоспециализированную связь. Представьте себе, что эта команда отвечает за один-единственный сервис, скажем сервис каталогов нашего музыкального магазина. Теперь рассмотрим внутреннее устройство сервиса: множество вызовов узкоспециализированных методов или функций. Как уже говорилось, мы стремимся к тому, чтобы сервисы создавались в результате разбиения системы на такие части, при которых темпы развития внутри сервисов были намного выше темпов развития между сервисами. Наша единственная команда, имея возможность организовывать узкоспециализированное общение, отлично сочетается с направлениями обмена данными в коде внутри сервиса.

Одной команде проще обсуждать предложения по внесению изменений и перестроению программного кода, и, как правило, у нее имеется высокоразвитое чувство собственности.

Теперь представим себе другой сценарий. Предположим, что вместо одной находящейся в одном месте команды, владеющей нашим сервисом каталогов, есть две команды в Великобритании и Индии. Обе они принимают активное участие во внесении изменений в сервис и при этом фактически коллективно владеют этим сервисом. Географическая разобщенность и разные часовые пояса затрудняют оперативное общение между этими командами. Они полагаются на контакты более общего плана через видеоконференции и электронную почту. Просто ли будет сотруднику команды из Великобритании уверенно выполнить небольшое перестроение кода? Издержки от общения географически удаленных друг от друга

команд весьма велики, и поэтому высока и стоимость координации усилий при внесении изменений.

При увеличении стоимости координации внесения изменений случается одно из двух. Либо люди находят способы для сокращения стоимости координации и общения, либо прекращают вносить изменения. Именно последний вариант ожидает нас в случае использования объемных, трудоемких в обслуживании баз исходного кода.

Мне вспоминается один клиентский проект, над которым я работал, когда одним сервисом совместно владели две географически разделенные команды. Со временем каждая из сторон начала уточнять, какой работой она будет заниматься. Это позволило каждой из них завладеть той частью базы исходного кода, внутри которой предполагаются наименьшие расходы на внесение изменений. Затем команды вели дискуссии более общего плана о взаимосвязанности этих двух частей; фактически направление общения, ставшее возможным внутри организационной структуры, соответствовало API-интерфейсу, имеющему более общий характер, который сформировал границы между двумя половинами базы исходного кода.

Какое это имеет к нам отношение при рассмотрении того, как развивается конструкция нашего собственного сервиса? Я бы предположил, что географические границы между людьми, участвующими в разработке системы, могут стать отличным стимулом для разбиения сервиса на части и что в общем вы должны продумать возможность назначения собственником сервиса одной находящейся в одном месте команды, которая сможет поддерживать стоимость внесения изменений на низком уровне.

Возможно, в вашей организации решат, что требуется увеличить штат сотрудников, работающих над вашим проектом, открыв офис в другой стране. Тогда вам придется хорошенько подумать о том, разработку какой части вашей системы можно будет туда переместить. Возможно, это подстегнет вас к принятию решения о том, по каким стыкам выполнять следующие разбиения.

В связи с этим стоит также заметить, что, основываясь как минимум на наблюдении авторов ранее упомянутого отчета *Exploring the Duality Between Product and Organizational Architectures*, можно сказать: если организация, создающая систему, имеет менее крепкие связи (к примеру, она состоит из географически разобщенных команд), то создаваемые этой организацией системы будут, как правило, иметь более модульную структуру и поэтому, будем надеяться, обладать меньшей связанностью. Поддерживать в географически разбросанной организации тенденцию получения более тесной интеграции за счет того, что одна команда владеет сразу несколькими сервисами, очень трудно.

Владение сервисом

Что я подразумеваю под владением сервисом? В общем смысле это означает, что команда, владеющая сервисом, несет ответственность за внесение изменений в этот сервис. Команда должна иметь возможность свободно проводить реструктуризацию кода по собственному усмотрению, если только эти изменения не нарушат условий использования сервисов. Для многих команд права владения

распространяются на все аспекты сервиса, от исходных требований до создания, развертывания и сопровождения приложения. Эта модель особенно широко распространена при работе с микросервисами: небольшой команде намного легче владеть небольшим сервисом. Возросший уровень владения приводит к росту автономности и скорости поставки. Когда одна команда отвечает за развертывание и сопровождение, это означает, что у нее есть стимул создавать сервисы, которые потом можно будет легко развернуть, то есть опасения насчет «перебрасывания чего-нибудь через забор» развеиваются, когда это что-то просто некому перебрасывать!

Конечно, это одна из тех моделей, которым я отдаю предпочтение. Она передает право принятия решений тем людям, которые могут справиться с этим лучше других, повышая тем самым эффективность работы и автономность команды, но и заставляя ее нести ответственность за свою работу. Мне приходилось видеть слишком многих разработчиков, передающих свои системы на тестирование и развертывание и считающих, что на этом их работа завершена.

Побудительные мотивы для создания общих сервисов

Мне встречалось множество команд, принимавших на вооружение модель общего владения сервисами. Я не считаю эту модель оптимальной по причинам, которые уже были рассмотрены. Но думаю, что разобраться в мотивах, побуждающих людей к выбору общих сервисов, для нас очень важно, особенно в том смысле, что это, возможно, позволит нам присмотреть для себя ряд весьма убедительных альтернативных моделей, с помощью которых можно будет решить основные проблемы людей.

Слишком большие трудности разбиения на части

Вполне очевидно, что одной из причин, по которым вам может встретиться отдельно взятый сервис, находящийся во владении более чем одной команды, будет слишком высокая стоимость разбиения сервиса, или же в вашей организации могут не видеть в этом никакого смысла. Такое часто случается при работе с большими монолитными системами. Если это самые большие затруднения, с которыми вам пришлось столкнуться, то я надеюсь, что вы сможете воспользоваться некоторыми из советов, которые были даны в главе 5. Вы также можете рассмотреть вопрос объединения команд, чтобы привести их состав к более полному соответствию архитектуре системы.

Команды разработки функций

Идея команд разработки функций, известных также как функционально ориентированные команды, заключается в том, что небольшие команды разрабатывают набор функций, реализующих всю функциональную нагрузку, которая потребует-

ся даже при пересечении границ компонента (или даже сервиса). Цели, стоящие перед командами разработки функций, вполне обоснованны. Эта структура позволяет команде сконцентрироваться на конечном результате, гарантируя тем самым, что проделанная ими работа будет содействовать устранению ряда сложностей, возникающих при попытке скоординировать изменения, вносимые несколькими различными командами.

Во многих ситуациях команды разработки функций являются реакцией на работу традиционных IT-организаций, где структура команд выстраивается вокруг технических границ. Например, у вас может быть команда, отвечающая за пользовательский интерфейс, еще одна команда, отвечающая за логику приложения, и третья команда, работающая с базой данных. В этой среде команда разработки функций сможет оказать существенную помощь, поскольку она работает над предоставлением функциональных возможностей для всех этих уровней.

При повсеместном использовании команд разработки функций все сервисы могут считаться общими. Внести изменения в любой сервис, в любой фрагмент кода может кто угодно. В таком случае роль кураторов сервисов существенно усложняется, если только она вообще существует. К сожалению, мне редко приходится видеть работу кураторов там, где используется эта схема, и отсутствие этой работы приводит к возникновению тех проблем, о которых говорилось ранее.

Еще раз рассмотрим, что собой представляют микросервисы, — это сервисы, смоделированные по образцу не технической, а бизнес-области. И если команда, владеющая каким-либо конкретным сервисом, аналогичным образом соответствует какой-либо бизнес-области, то, скорее всего, она сможет сохранить ориентированность на клиента и разглядеть больше перспектив разработки, поскольку у нее есть целостное понимание и владение всеми технологиями, связанными с сервисом.

Могут, конечно, происходить и комплексные изменения, но их вероятность существенно снижена тем обстоятельством, что мы избегаем формирования команд по технологическим признакам.

Узкие места, касающиеся вопросов поставки

Одной из основных причин, по которым люди стремятся к переходу на применение совместно используемых сервисов, является стремление избежать узких мест в вопросах поставки программных средств. Что делать, если в отдельно взятый сервис требуется внести сразу множество изменений? Представим, что мы предоставляем клиенту возможность видеть музыкальный жанр записи во всех наших товарах, а также возможность добавления совершенно нового типа материала — виртуальных музыкальных рингтонов для мобильного телефона. Команде сайта нужно внести изменения, чтобы отобразить информацию о жанре, а команде мобильной версии приложения — поработать над тем, чтобы позволить пользователям перемещаться по перечню рингтонов, запускать их предварительное прослушивание и покупать их. Оба изменения должны быть внесены в сервис каталогов, но, к сожалению, половина команды загрипповала, а другая половина застряла на выяснении причины сбоя сервиса при работе в производственном режиме.

Избежать подобной ситуации помогут несколько вариантов, не связанных с привлечением общих сервисов. Можно просто подождать и занять команды сайта и мобильного приложения чем-нибудь другим. В зависимости от степени важности функции или того, насколько долгой предполагается задержка, этот вариант вполне может нам подойти, но может и перерасти в весьма острую проблему.

Вместо этого можно будет усилить команду каталогов, чтобы они смогли быстрее справиться с работой. Чем стандартнее будут используемые в вашей системе технологический стек и идиомы программирования, тем проще будет другим специалистам вносить изменения в ваши сервисы. Разумеется, оборотной стороной медали, как уже говорилось, будет то, что стандартизация неизменно приводит к сокращению возможностей команд по выбору наиболее подходящего решения для выполнения своей работы и может привести к разнообразным малоэффективным решениям. Если же команда находится на другой стороне планеты, реализация такого варианта может стать невозможной.

Еще один вариант заключается в разбиении каталога на отдельный основной музыкальный каталог и каталог рингтонов. Если изменение, вносимое для поддержки рингтонов, очень небольшое и вероятность того, что эта область в будущем станет бурно развиваться, совсем невелика, это разбиение может быть преждевременным. Но если функции, связанные с рингтонами, приносят прибыль в течение десяти недель, разбиение сервиса на две части с передачей владения команде мобильного варианта может приобрести вполне определенный смысл.

Но есть и еще одна модель, которая может неплохо работать в наших интересах.

Семейственный открытый код

А что, если мы приложили максимум усилий, но так и не смогли найти способа, исключающего применение нескольких общих сервисов? В таком случае вполне определенный смысл может иметь правильное внедрение модели семейственного открытого кода.

При использовании обычного открытого кода его исполнителями являются немного людей. Они же являются кураторами кода. При желании внести изменения в проект с открытым кодом вы либо обращаетесь к одному из исполнителей, чтобы он внес для вас эти изменения, либо вносите изменения самостоятельно и отправляете исполнителям запрос на их прием. Основные исполнители по-прежнему несут ответственность за исходный код и являются его владельцами.

Эта же схема может неплохо работать и внутри организации. Возможно, те люди, которые изначально работали над сервисом, больше не входят в одну команду и теперь разбросаны по всей организации. Но, если у них еще есть права исполнителей, их можно найти и попросить помочь, возможно вступив с ними в сотрудничество, а если у вас уже есть подходящий инструментарий, исполнителям можно отправить запрос на его прием.

Роль кураторов

Нам хочется, чтобы наши сервисы были практичными. Хочется, чтобы код был высокого качества и сам сервис проявлял некую последовательность при сборе всего в общее целое. Мы также хотим убедиться в том, что изменения, вносимые сейчас, не затруднят внесение намечаемых на будущее изменений сверх допустимого. Это означает, что внутри нашей организации необходимо применять те же схемы, которые используются в обычном открытом коде, то есть выделить группу надежных исполнителей (основную команду) и ненадежных исполнителей (людей, не входящих в команду предлагающих изменения).

Основная команда должна иметь способ проверки и утверждения изменений. Нужно убедиться в идиоматической согласованности изменений, то есть в том, что они следуют общим принципам программирования остального кода. Следовательно, людям, выполняющим проверку, приходится тратить время на работу с теми, кто вносит предложения, чтобы убедиться в достаточной качественности вносимого изменения.

Хорошие контролеры проделывают в связи с этим большой объем работы, общаясь непосредственно с теми, кто вносит предложения, и поощряя их хорошие поступки. Плохие контролеры могут воспользоваться своим положением для демонстрации власти над другими или ведения мировоззренческих сражений вокруг произвольных технических решений. Будучи свидетелем обеих форм поведения, хочу сказать только одно: в любом случае на это тратится определенное время. Когда кураторы позволяют ненадежным исполнителям отправлять свои изменения в ваш исходный код, вам нужно решить, стоит ли беспокоиться об издержках на использование контролера — о том, может ли основная команда заняться чем-нибудь более полезным, чем проверка исправлений или изменений.

Зрелость

Чем менее стабильным или зрелым является сервис, тем труднее будет позволять людям, которые не входят в основную команду, присылать исправления или изменения. Пока не сформируется основа сервиса, команда может не знать, как отличить хорошее от плохого, и поэтому стремиться к тому, чтобы узнать, на что должно быть похоже дельное предложение. На этом этапе сервис подвергается значительным изменениям.

При ведении большинства проектов с открытым кодом предложения от широкой аудитории ненадежных исполнителей стараются не принимать до тех пор, пока не будет сформировано ядро первой версии. В вашей организации также есть смысл следовать этой модели. Если сервис уже достаточно зрелый и редко подвергается изменениям (таков, к примеру, сервис покупательской корзины), возможно, настало время открыть его для внесения предложений от других сотрудников.

Инструментарий

Для поддержки модели семейственного открытого кода на самом высоком уровне требуется наличие соответствующего инструментария. Важную роль играет

использование средства распределенного управления версиями с предоставлением людям возможности отправки запросов на внедрение кода или чего-либо подобного этому. В зависимости от размеров организации может также понадобиться инструментарий, позволяющий обсуждать прохождение запросов на исправления и изменения, в котором может подразумеваться, а может, и нет, наличие системы полномасштабного обзора кода, но наибольшую пользу может принести возможность выставления встроенных комментариев по поводу исправлений. И наконец, нужно максимально облегчить исполнителю создание и развертывание вашего программного средства и сделать его доступным для других исполнителей. Обычно под этим подразумевается наличие четко определенных конвейеров сборки и развертывания и централизованных хранилищ артефактов.

Ограниченные контексты и структуры команд

Как уже упоминалось, мы намечаем границы сервисов вокруг ограниченных контекстов. Из этого следует, что нам нужно, чтобы команды также были увязаны с ограниченными контекстами. Такая увязка дает массу преимуществ. Во-первых, команде будет проще усвоить понятия в выделяемой ей области внутри ограниченного контекста, поскольку они будут связаны между собой. Во-вторых, сервисы внутри ограниченного контекста, вероятнее всего, будут общаться друг с другом, упрощая тем самым устройство системы и координацию выпусков. И в-третьих, с точки зрения взаимодействия команд поставки с заинтересованными лицами упрощается налаживание командой хороших взаимоотношений с одним или двумя специалистами в данной области.

Осиротевшая служба?

Что можно сказать о сервисах, которые больше не имеют активного сопровождения? Поскольку мы движемся в направлении использования узкоспециализированных архитектур, сами сервисы становятся меньше. Как уже говорилось, одной из целей уменьшения сервисов является их упрощение. Менее сложные сервисы с меньшим количеством функций долго могут не нуждаться в изменениях. Рассмотрим довольно скромный сервис покупательской корзины, предоставляющий ряд весьма незатейливых возможностей: положить в корзину, убрать из корзины и т. д. Вполне вероятно, что этот сервис не потребует внесения изменений в течение многих месяцев после написания даже притом, что приложение продолжает активно развиваться. И что с ним случится? Кто владеет этим сервисом?

Если структуры команд увязаны с ограниченными контекстами организации, значит, даже сервисы, не требующие частого внесения изменений, имеют де-факто своих владельцев. Представим себе команду, сформированную в соответствии с контекстом потребителей веб-продаж. Она должна заниматься сервисами сайта,

покупательской корзины и рекомендаций. Даже если сервис покупательской корзины не менялся месяцами, вполне естественно, что изменениями в нем займется эта команда. Одно из преимуществ микросервисов выражается, конечно же, в том, что при необходимости изменить сервис, чтобы добавить в него новое свойство, перезапись сервиса не должна занять слишком много времени даже для команды, не испытывающей особого рвения.

Тем не менее если за основу был принят разноязычный подход, предполагающий использование нескольких технологических стеков, то сложности внесения изменений в осиротевший сервис могут усугубиться, если в вашу команду больше не входят специалисты нужного профиля.

Конкретный пример: RealEstate.com.au

Основной бизнес компании REA связан с недвижимостью. Но он включает несколько различных аспектов, каждый из которых работает как отдельное направление бизнеса (LOB). Например, в одном направлении бизнеса занимаются сделками с жилой недвижимостью в Австралии, в другом — коммерческой недвижимостью, а третье может иметь отношение к одной из внешних бизнес-задач компании REA. У этих направлений бизнеса имеются IT-команды (или подразделения) поставки, причем у некоторых направлений может существовать только одно подразделение, а у самого крупного их четыре. Итак, на направлении жилой недвижимости несколько команд занимаются созданием сайта и перечня услуг, чтобы позволить людям просматривать собственность. Бывает, что специалисты переходят из одной команды в другую, но обычно долго определенное направление бизнеса не покидают, гарантируя тем самым компетентность в данной части бизнес-области. Это, в свою очередь, способствует общению между различными заинтересованными лицами и командой, поставляющей им те или иные функции программных средств.

Ожидается, что каждое подразделение внутри направления бизнеса несет ответственность за весь жизненный цикл создаваемых им сервисов, включая создание, тестирование и выпуск, поддержку и даже вывод из эксплуатации. Основная команда службы поставок выдает рекомендации и указания этим командам, а также разрабатывает инструментарий, помогающий выполнять их задачи. Ключевым положением является культура автоматизации, и в REA широко используется AWS как основная составляющая, повышающая автономность команд. Порядок работы всего этого механизма показан на рис. 10.1.

С работой бизнеса увязывается не только организация поставок. Увязывание распространяется и на архитектуру. Примером могут послужить методы интеграции. Внутри направления бизнеса все сервисы вольны обмениваться данными друг с другом любым подходящим для них способом в соответствии с решениями подразделений, выполняющих роль их кураторов. Но весь обмен данными между направлениями бизнеса предписано вести путем обмена пакетами в асинхронном режиме, что является одним из немногих железных правил весьма небольшой архитектурной команды. Этот обмен данными, имеющий более общий характер, соответствует такому же разностороннему обмену данными, который практикуется

и между различными частями бизнеса. Благодаря настоятельному требованию ведения пакетного обмена между направлениями каждому направлению бизнеса предоставляется широкая степень свободы действий и самоуправления. Им разрешено удалять свои сервисы по собственному усмотрению, и пока они могут поддерживать пакетную интеграцию с другими частями бизнеса и с заинтересованными лицами, никого это волновать не будет.

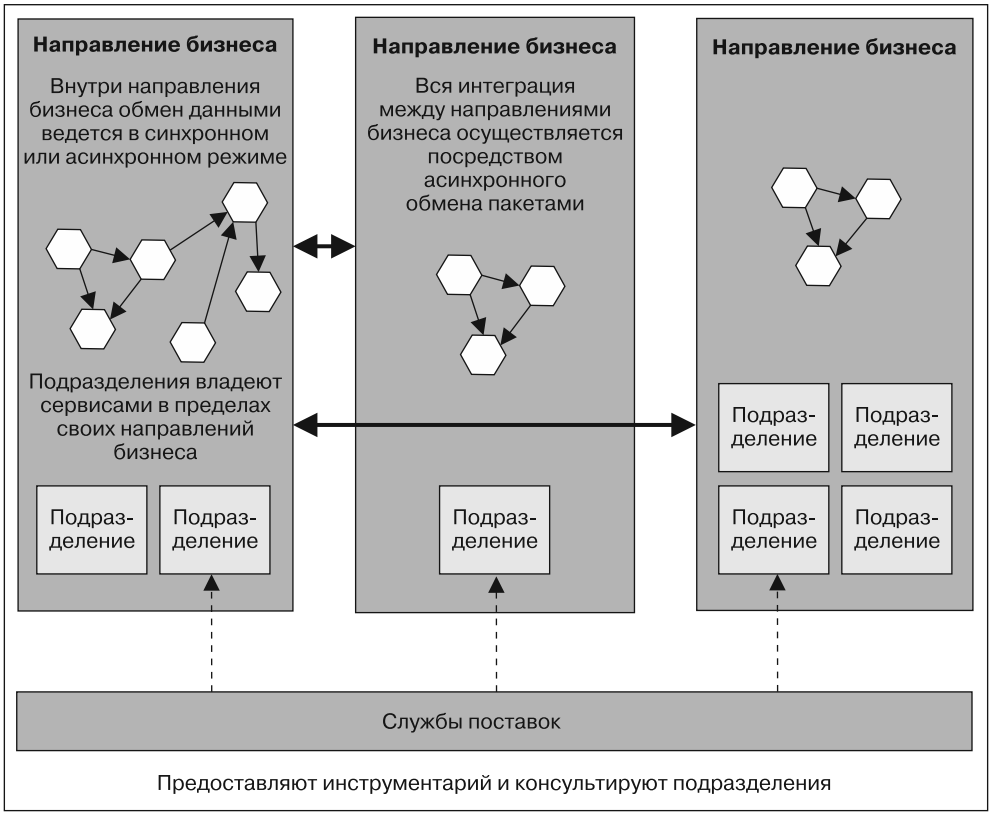


Рис. 10.1. Общее представление о структуре организации Realestate.com.au и ее команд, а также об увязке с архитектурой

Эта структура позволяет добиться высокой автономности не только команд, но и различных частей бизнеса. Начав с совсем небольшого количества сервисов несколько лет назад, REA теперь обладает сотнями сервисов, количество которых превышает количество сотрудников и растет высокими темпами. Возможность поставки изменений помогла компании добиться значительных успехов как на местном рынке, так и за рубежом. И что отраднее всего, из разговоров с людьми я вынес впечатление, что существующее состояние как архитектуры, так и организационной структуры сформировалось благодаря самым последним шагам на этом пути, но никак не достижению конечного пункта назначения. Я полагаю, что в ближайшие пять лет облик REA снова претерпит существенные изменения.

Организации с адаптивностью, достаточной для изменения не только системной архитектуры, но и организационной структуры, могут получить огромные преимущества в сфере повышения автономности команд и сокращения времени вывода на рынок новых свойств и функциональных возможностей. REA не пребывает в вакууме, а является одной из множества организаций, реализующих подобную системную архитектуру.

Закон Конвея наоборот

До сих пор речь шла о том, как организационная структура влияет на устройство системы. А как насчет обратного влияния? То есть может ли устройство системы изменить организацию? Несмотря на то что я не смог найти столь же убедительных доказательств идеи о том, что закон Конвея работает и в обратном порядке, периодически мне приходилось сталкиваться с этим явлением.

Наверное, наилучшим примером был клиент, с которым я работал много лет назад. В те дни, когда Всемирная сеть только зарождалась, а Интернет воспринимался как нечто поступающее на гибком диске компании AOL через входную дверь, эта компания представляла собой довольно солидную издательскую фирму, у которой был небольшой, весьма скромный сайт. Это был сайт ради сайта, и, по большому счету, порядок, в котором работал бизнес, был совершенно неважен. Когда была создана исходная схема, порядок работы системы строился на весьма произвольных технических решениях.

Содержимое для этой системы обеспечивалось несколькими способами, но основная часть поступала от третьих лиц, размещавших объявления для всеобщего обозрения. Была система ввода информации, позволявшая третьим лицам за плату создавать содержимое, центральная система, получавшая данные и улучшавшая их различными способами, и система вывода, формировавшая окончательный вид сайта.

Вопрос о том, были ли исходные конструкторские решения правильными на то время, мы оставим для историков, но за многие годы в компании не происходило существенных изменений, и я и многие мои коллеги удивились бы, если бы устройство системы подошло для нынешнего состояния компании. Физический печатный бизнес организации существенно сократился, а преобладают доходы и, следовательно, бизнес-операции из интернет-сферы.

Все это время мы наблюдали, как организация четко подстроилась под систему, разработанную сторонними специалистами. Три канала или подразделения, присутствовавшие в IT-сфере бизнеса, были увязаны с каждой из сторон бизнеса: вводом, ядром и выводом. Внутри каналов были выделены команды поставки. В то время я не понял, что эти организационные структуры не предшествовали устройству системы, а фактически сформировались вокруг этого устройства. По мере сокращения физической стороны бизнеса и роста его цифровой стороны устройство системы ненароком указало путь роста самой организации.

В итоге мы поняли, что, несмотря на имеющиеся недостатки в устройстве системы, для перехода в новое качество понадобилось внесение изменений в структуру организации. Сколько лет прошло, а этот процесс все еще не завершен!

Люди

Неважно, как это выглядит на первый взгляд,
но проблема всегда в людях.

Джеральд Вайнберг (Gerry Weinberg).
Второй закон консалтинга

Следует признать, что в среде микросервисов разработчику труднее думается о написании кода в собственной ограниченной области. Он должен больше вникать в последствия вызовов, пересекающих границы сетей, или последствия сбоев. Мы также говорили о том, что с микросервисами проще испытывать применение новых технологий — от хранения данных до языков программирования. Но для тех, кто попал в мир микросервисов из мира монолитных систем, где большинству разработчиков приходится использовать один язык, совершенно не обращая внимания на оперативные интересы, такое положение вещей может стать толчком к внезапному пробуждению.

Точно так же может стать обременительным принуждение команд разработчиков к повышению автономности. Люди, которые в прошлом перебрасывали свою работу кому-то еще через забор, привыкли выдвигать обвинения в чужой адрес и не могут почувствовать себя комфортно, когда несут за свою работу полную ответственность. Могут быть даже выработаны условия контракта, предписывающие вашим разработчикам ношение приборов оперативного оповещения о сбоях в работе систем, которые они поддерживают!

Хотя эта книга посвящена в основном решению технологических вопросов, людей нельзя сбрасывать со счетов, ведь речь идет о людях, которые создали то, чем вы пользуетесь сейчас, и создают все, что будет применяться в дальнейшем. Если придумывать, как все должно делаться, без учета мнения персонала или людских возможностей, то это, скорее всего, не приведет ни к чему хорошему.

Что касается данной темы, то у каждой организации имеется свой ход развития. Нужно понимать, насколько ваш персонал готов к изменениям. Не нужно их подталкивать сверх всякой меры! Вероятно, на короткий период времени у вас попрежнему будет отдельная команда, занимающаяся поддержкой клиентов или разработкой, что даст разработчикам время на то, чтобы подстроиться под новые инструкции. Возможно, придется смириться с тем, что для выполнения всей этой работы потребуются люди различных наклонностей. При любом подходе нужно понимать, что в мире микросервисов следует четко определять зоны ответственности людей, а также понимать, почему именно такие зоны ответственности играют для вас важную роль. Это может помочь вам заметить возможные квалификационные пробелы и подумать о том, как их заполнить. Для многих людей это путешествие станет весьма непростым. Всегда нужно помнить о том, что без людей, с которыми вы работаете, любые изменения, которые захочется внести, с самого начала могут быть обречены на провал.

Резюме

Закон Конвея подчеркивает опасности попыток навязывания моделей систем, не соответствующих структуре организации. Он подводит нас к стремлению закрепить владение микросервисами за командами, расположенными в одном месте, которые, в свою очередь, выстраиваются вокруг соответствующих ограниченных контекстов, существующих в организации. Если такая двойная увязка отсутствует, мы получаем точки напряженности, рассмотренные в данной главе. Признавая связь между этими двумя понятиями, мы сможем гарантировать наличие смысла создаваемой нами системы для заказавшей ее организации.

Часть рассмотренных здесь вопросов имела отношение к сложностям работы с расширяющимися организациями. Но есть и другие требующие рассмотрения технические вопросы, которыми следует заняться, когда системы начинают расширяться, выходя за рамки нескольких обособленных сервисов. К их рассмотрению мы приступим в следующей главе.

11 Масштабирование микросервисов

Когда работаешь с изящными небольшими примерами форматом с книжную страницу, все кажется простым. Но реальный мир намного сложнее. Что будет, когда наши архитектуры микросервисов станут расширяться и превращаться из простых и скромных в нечто более сложное? Что будет, когда нам придется справляться со сбоями нескольких отдельных сервисов или управлять сотнями сервисов? Какие схемы нужно будет скопировать, когда микросервисов у вас станет больше, чем людей? Давайте все это выясним.

Сбои могут происходить везде

Мы понимаем, что нештатных ситуаций не избежать. Может отказать жесткий диск. Может дать сбой наша программа. И все, кто читал об ошибках, происходящих при распределенном вычислении, могут сказать о том, что знают о ненадежности сети. Мы можем приложить максимум усилий, пытаясь ограничить число сбоев, но при определенном масштабе сбои становятся неизбежными. Жесткие диски, к примеру, сейчас надежнее, чем когда-либо прежде, но временами и они выходят из строя. Чем больше у вас жестких дисков, тем выше вероятность отказа для отдельной стойки; при расширении масштаба отказ становится статистически неизбежен.

Даже тем из нас, кто не собирается слишком сильно расширять свою систему, все же стоит учитывать возможность сбоев. Если мы, к примеру, можем изящно обрабатывать сбои сервиса, то у нас получится и обновлять службу на месте, поскольку иметь дело с запланированными отключениями намного проще, чем с незапланированными.

Мы также сможем тратить немного меньше времени на попытки остановить неизбежное и уделять немного больше времени на то, чтобы справиться с ними как можно изящнее. Меня удивляет, что немалое количество организаций тратят силы и средства в первую очередь на попытки предотвращения сбоев и значительно меньше усилий направляют на упрощение восстановления после них.

Предположение о том, что все может дать сбой и неизбежно его даст, заставит изменить представление о том, как решать проблемы.

Я видел пример такого мышления, когда много лет назад был в кампусе Google. В приемной одного из зданий в Маунтин-вью в качестве своего рода экспозиции

стояла старая стойка с машинами. И там я кое-что заприметил. Эти серверы были без кожухов — голые материнские платы, вставленные в стойку. Но мне бросилось в глаза то, что жесткие диски были смонтированы на липучках. Я спросил одного из сотрудников Google о том, почему так сделано. Он ответил, что жесткие диски настолько часто выходят из строя, что им не захотелось их прикручивать. Их просто вытаскивали, бросали в мусорное ведро и крепили на липучке новый диск.

Итак, позвольте повторить: при расширении, даже если будут приобретены самый лучший комплект, самое дорогое оборудование, избежать того, что что-то может дать сбой и сделает это, просто невозможно. Поэтому нужно предполагать, что сбой может произойти. Если это обстоятельство учитывать во всем, что вы создаете, и планировать сбои, можно будет пойти на различные компромиссы. Если известно, что система сможет справиться с тем фактом, что сервер может дать сбой и даст его, то зачем беспокоиться и сильно тратиться на все это? Почему бы не воспользоваться простой материнской платой с самыми дешевыми компонентами (и какими-нибудь липучками), как это было сделано в Google, не слишком переживая за стойкость отдельного узла?

Слишком много — это сколько?

Тема межфункциональных требований уже рассматривалась в главе 7. Представление о межфункциональных требованиях формируется из рассмотрения таких аспектов, как живучесть данных, доступность сервисов, пропускная способность и приемлемое время отклика сервисов. Многие технологии, упоминаемые в этой и других главах, имеют отношение к подходам, позволяющим реализовать эти требования, но о том, какими конкретно могут быть эти требования, знаете только вы.

Обладание автоматически масштабируемой системой, способной реагировать на возросшую нагрузку или отказ отдельных узлов, может быть фантастическим результатом, но окажется избыточным для системы создания отчетов, которую нужно запускать только дважды в месяц и для которой вполне приемлемо пару дней находиться в простое. Аналогично этому отработка способов сине-зеленых развертываний для ликвидации простоев сервиса может иметь смысл для системы электронной торговли, но для корпоративной базы знаний, доступной только во внутренней сети организации (в интранете), будет, наверное, излишней.

Задать количество приемлемых отказов или допустимую скорость системы можно на основе требований, предъявляемых пользователями данной системы. Это поможет вам понять, какие технологии разумнее всего будет применить. Тем не менее пользователи не всегда смогут сформулировать конкретные требования. Следовательно, чтобы получить правильную информацию и помочь им понять, каковы будут относительные затраты на предоставление различных уровней услуг, нужно задавать вопросы.

Как уже упоминалось, межфункциональные требования от сервиса к сервису могут отличаться друг от друга, но мне хотелось бы предложить определить некоторые универсальные межфункциональные требования, а затем переопределить их для конкретных вариантов использования. Когда рассматриваются вопросы о необходимости и способах масштабирования системы, позволяющего лучше

справиться с нагрузкой или сбоями, постарайтесь сначала разобраться со следующими требованиями.

- **Время отклика/задержка.** Сколько времени должно тратиться на ту или иную операцию? Здесь может пригодиться измерение данного показателя в ходе работы различного количества пользователей с целью определения степени влияния возрастающей нагрузки на время отклика. Из-за характерных особенностей сетей неизбежны выпадения, поэтому может пригодиться задание целей для заданной процентили отслеживаемых ответов. Цель должна также включать количество параллельных подключений (пользователей), с которым, как ожидается, сможет справиться ваша программа. Следовательно, вы можете сказать: «Мы ожидаем, что у сайта значение 90-й процентили времени отклика будет порядка 2 секунд при обслуживании 200 параллельных подключений в секунду».
- **Доступность.** Ожидается ли падение сервиса? Считается ли, что сервис работает в режиме 24/7? Некоторым при оценке доступности нравится смотреть на периоды допустимого вынужденного простоя, но насколько это практично для тех, кто вызывает ваш сервис? Я должен либо полагаться, либо не полагаться на доступность сервиса. Фактически оценка периодов вынужденного простоя более полезна с точки зрения исторической отчетности.
- **Живучесть данных.** Каков приемлемый объем потери данных? Каков обязательный срок хранения данных? Скорее всего, для разных случаев можно будет дать разные ответы на эти вопросы. Например, можно выбрать вариант годичного хранения журналов регистрации пользовательских сеансов или с целью экономии дискового пространства — менее продолжительный срок, но записи о финансовых транзакциях может потребоваться хранить в течение многих лет.

Когда требования будут определены, вам понадобится способ постоянной систематической оценки их соблюдения. Можно, к примеру, прийти к решению о проведении тестов производительности, чтобы убедиться, что в этом смысле система отвечает приемлемым целям, нужно также обеспечить отслеживание соответствующей статистики и в производственном режиме работы!

Снижение уровня функциональных возможностей

Важной частью создания отказоустойчивой системы, особенно когда функциональные возможности распределяются среди нескольких микросервисов, которые могут находиться как в рабочем, так и в нерабочем состоянии, является обеспечение ее способности безопасно снижать уровень функциональности. Представим себе стандартную веб-страницу на нашем сайте электронной торговли. Чтобы собрать вместе различные части этого сайта, требуется участие в работе нескольких микросервисов. Один микросервис может выводить на экран подробности о предлагаемом к продаже альбоме, другой — показывать цену и уровень запасов товара. И мы, наверное, будем показывать также содержимое покупательской корзины,

чем может заниматься еще один микросервис. Если при отказе одного из этих сервисов станет недоступной вся страница, то мы, вероятно, создали систему менее устойчивую, чем та, которая требует доступности только одного сервиса.

Нам нужно понять, каково влияние каждого отказа, и выработать способы надлежащего снижения уровня функциональности. Если недоступен сервис покупательской корзины, это, наверное, будет весьма неприятно, но мы по-прежнему сможем показывать веб-страницу с перечнем товаров. Возможно, мы просто скроем покупательскую корзину или выведем вместо нее значок с надписью «Скоро вернусь!».

При работе с единым монолитным приложением нам не приходится принимать множество решений. Здоровье системы зависит от работы двоичного кода. Но при использовании архитектуры микросервисов нужно рассматривать намного более тонкие ситуации. Зачастую правильные действия в любой ситуации не связаны с принятием технического решения. Нам может быть известно, что технически можно сделать при отказе корзины, но пока мы не сможем осмыслить бизнес-контекст, мы не поймем, какое действие нужно предпринять. Возможно, мы закроем весь сайт, позволим людям просматривать каталог товаров или заменим ту часть пользовательского интерфейса, в которой содержатся элементы управления корзиной, номером телефона, по которому можно сделать заказ. Но для каждого показываемого клиенту интерфейса, в котором используются несколько микросервисов, или для каждого микросервиса, зависящего от нескольких нижестоящих, сотрудничающих с ним микросервисов, следует задаться вопросом: «Что произойдет при отказе?» — и знать, что нужно будет делать.

Критический взгляд на каждую из возможностей в понятиях межфункциональных требований позволит намного лучше сориентироваться в определении необходимых действий. Теперь рассмотрим, что можно сделать с технической точки зрения, чтобы при возникновении сбоя мы могли с ним успешно справиться.

Архитектурные меры безопасности

Существует несколько схем, которые я в совокупности называю *архитектурными мерами безопасности* и которые при возникновении нештатных ситуаций могут использоваться, чтобы предотвращать появление раздражающих, распространяющихся за пределы сервиса эффектов. Вам важно усвоить их положения и строго придерживаться стандартизации в системе, чтобы гарантировать, что ни один ее нерадивый компонент не сможет прямо у вас на глазах вызвать всеобщее обрушение. Вскоре мы посмотрим, какие основные меры безопасности заслуживают внимания, но перед этим я хочу поделиться небольшой историей, чтобы очертить круг возможных нештатных ситуаций.

Я был техническим руководителем проекта по созданию сайта классифицированной онлайн-рекламы. Сайт обслуживал довольно большие объемы информации и приносил довольно высокий доход. Наше основное приложение занималось выводом на экран классифицированных рекламных объявлений, а также служило прокси-сервером вызовов, осуществляемых по адресам других сервисов, предоставляющих различные виды продукции (рис. 11.1).

Фактически это пример *приложения-душителя*, где новая система перехватывает вызовы, совершаемые в адрес ранее существовавших приложений, и постепенно полностью заменяет собой эти приложения. В рамках этого проекта мы уже были на полпути к списанию прежних приложений. Мы только что перешли к использованию самых крупных объемов и самых прибыльных продуктов, но многие рекламные объявления по-прежнему обслуживались целым рядом прежних приложений. С точки зрения как количества поисковых запросов, так и дохода от этих приложений быстро избавиться от них не представлялось возможным.

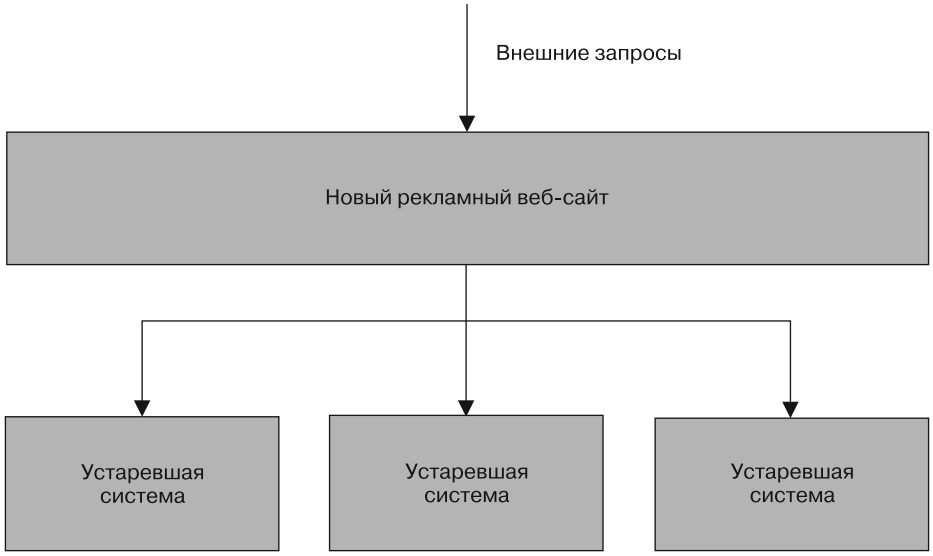


Рис. 11.1. Сайт классифицированных рекламных объявлений, на котором происходит постепенное избавление от устаревших приложений

Какое-то время наша система демонстрировала высокую живучесть и примерное поведение, справляясь с незначительными нагрузками. Со временем в пиковые моменты нам пришлось обрабатывать 6000–7000 запросов в секунду, и несмотря на то, что большинство запросов интенсивно кэшировалось реверсными прокси-серверами, находящимися перед серверами нашего приложения, поисковые запросы товаров, являющиеся наиболее важным аспектом сайта, в большинстве своем в кэше отсутствовали и требовали обслуживания по полному серверному циклу.

Однажды утром, как раз перед достижением ежедневной пиковой нагрузки, что происходило во время обеденного перерыва, система стала замедляться, а затем постепенно выходить из строя. В новом основном приложении у нас было несколько уровней мониторинга, достаточных для того, чтобы сообщить о достижении каждым узлом приложения 100%-й пиковой нагрузки на центральный процессор, превышающей нормальные уровни даже для пиковых ситуаций. Вскоре обрушилась вся система.

Нам удалось выяснить причину произошедшего и восстановить работу сайта. Оказалось, что одна из нижестоящих рекламных систем, самая старая и хуже всех поддерживаемая, начала выдавать ответы очень медленно. Подобное поведение является одним из наихудших режимов сбоя, с которым можно столкнуться. Отсутствие системы определяется довольно быстро. А когда она просто *замедляется*, то прежде, чем среагировать на сбой, приходится некоторое время занимать выжидательную позицию. Но какой бы ни была причина неполадок, мы создали систему, уязвимую для каскадного сбоя. Практически не контролируемый нами нижестоящий сервис способен был обрушить всю систему.

Пока одна команда изучала проблемы, возникшие с нижестоящей системой, все остальные приступили к выявлению причин возникновения нештатной ситуации в нашем приложении. Были обнаружены сразу несколько проблем. Для обслуживания нижестоящих подключений мы использовали пул HTTP-соединений. Потоки этого пула имели показатели времени ожидания, настроенные на время ожидания HTTP-вызова, направляемого в адрес нижестоящей системы, что было вполне приемлемо. Проблема заключалась в том, что всем исполнителям при замедлении нижестоящей системы приходилось ожидать истечения лимита времени. Пока они ждали, в пул приходили новые запросы, требовавшие исполнительных потоков. Из-за отсутствия доступных исполнителей эти запросы зависали. Оказалось, что у библиотеки пула соединений, которую мы использовали, была настройка лимита времени ожидания исполнителей, но *по умолчанию она была отключена*! Это вызвало огромное скопление заблокированных потоков. У нашего приложения в любой момент времени обычно было 40 параллельных соединений. В течение пяти минут возникшая ситуация привела к резкому возрастанию количества соединений до 800, что и обрушило систему.

Хуже того, нижестоящий сервис, с которым шел диалог, обеспечивал менее 5 % функциональных возможностей, используемых нашей клиентской базой, а доля доходов от него была еще меньше. Разобравшись в ситуации, мы пришли к стойкому убеждению, что с системами, просто замедляющими свою работу, справиться *намного* сложнее, чем с системами, которые быстро выходят из строя. Замедление в распределенных системах имеет убийственный эффект.

Даже при наличии правильно выставленных в пуле лимитов времени у нас для всех исходящих запросов был общий единственный пул HTTP-соединений. Это означало, что один медленный сервис мог в одиночку исчерпать количество доступных исполнителей, даже если все остальные продолжали работать в штатном режиме. В конце концов стало понятно, что рассматриваемый нижестоящий сервис дал сбой, но мы продолжали отправлять трафик в его направлении. В данной ситуации это означало, что мы усугубили и без того плохое состояние дел, поскольку у нижестоящего сервиса все равно не было шансов на восстановление нормального режима работы. Во избежание повторения подобных случаев мы сделали три доработки: выставили правильные значения *времени ожидания*, реализовали *переборки*, чтобы отделить друг от друга различные пулы соединений, и создали *предохранитель*, исключающий отправку вызовов к нездоровой системе.

Антихрупкая организация

В своей книге «Антихрупкость» (Random House) Нассим Талеб (Nassim Taleb) рассказывал о таких вещах, как получение, как бы странно это ни звучало, пользы от сбоев и нештатной работы. Ариэль Цейтлин (Ariel Tseitlin) применительно к тому, как работает Netflix, воспользовался этой концепцией для выработки понятия «антихрупкая организация».

Масштабы работы Netflix хорошо известны, как и тот факт, что Netflix целиком полагается на AWS-инфраструктуру. Эти два фактора означают, что данное понятие должно включать в себя также возможность возникновения сбоя. Компания Netflix выходит за рамки этого подхода, фактически *провоцируя* сбой, чтобы убедиться в том, что система к нему устойчива.

Некоторые организации были бы рады устроить *испытательные дни*, в которые сбой имитируется выключаемыми системами и наблюдением за реакцией различных команд. Когда я работал в Google, это весьма часто практиковалось для различных систем, и я, конечно же, думал, что многие организации могут извлечь пользу из регулярного выполнения подобных упражнений. Google выходит за рамки простых тестов для имитации сбоя сервера и как часть своих ежегодных упражнений DiRT (Disaster Recovery Test — тестирование на восстановление работоспособности после аварии) имитирует широкомасштабные бедствия наподобие землетрясений. Компания Netflix также практикует более агрессивный подход, создавая программы, вызывающие сбой, и ежедневно запуская их в производственном режиме.

Наиболее известная программа называется Chaos Monkey, она занимается тем, что в течение определенного времени выключает случайно выбранные машины. Сведения о том, что такое может произойти и реально происходит в производственном режиме, означают, что разработчики, создавшие системы, должны быть к этому по-настоящему готовы. Chaos Monkey является лишь одной из составляющих используемого в Netflix комплекса роботов имитации сбоев под названием Simian Army. Программа Chaos Gorilla используется для вывода из строя центра доступности (эквивалента датацентра в AWS), а программа Latency Monkey имитирует медленную работу сетевого соединения между машинами. Компания Netflix сделала эти инструментальные средства доступными под лицензией открытого кода. Для многих завершающим тестом надежности системы может стать выпуск на волю собственной обезьяньей армии (то есть Simian Army) в своей производственной инфраструктуре.

Включение и провоцирование сбоев посредством программных средств и создание систем, способных справиться с ними, — это всего лишь часть того, что делает Netflix. В этой компании понимают важность извлечения уроков из происходящих сбоев и привития культуры терпимости к допускаемым ошибкам. Затем к извлечению уроков привлекают разработчиков, поскольку каждый разработчик также отвечает за сопровождение своих сервисов, работающих в производственном режиме.

Искусственно вызывая сбои и создавая условия для их появления, компания Netflix обеспечивает более успешное масштабирование своих систем и лучше реагирует на нужды своих клиентов.

Принимать экстремальные меры по примеру Google или Netflix нужно не всем, при этом важно понять, что для работы с распределенными системами нужен иной

взгляд на вещи. Сбои неизбежны. То, что ваша система в данный момент разбросана по нескольким машинам (которые могут и будут сбоить) и по сети (которая обязательно проявит свою ненадежность), может как минимум повысить степень уязвимости системы. Следовательно, независимо от того, собираетесь ли вы предоставлять сервис в таких же масштабах, как Google или Netflix, готовность к сбоям, характерным для более распределенных архитектур, играет весьма важную роль. Итак, что же нам нужно сделать, чтобы справиться со сбоями в системах?

Настройки времени ожидания

Настройки времени ожидания очень легко упустить из виду, но для правильной работы с нижестоящими системами они играют весьма важную роль. Долго ли мне нужно ждать, пока я не смогу считать нижестоящую систему фактически отказавшей?

Если слишком долго ждать решения о том, что вызов не удался, можно замедлить работу всей системы. Если сделать время ожидания слишком маленьким, можно будет посчитать потенциально работоспособный вызов неудавшимся. Если полностью отказаться от времени ожидания, то обрушившаяся нижестоящая система может «подвесить» всю систему.

Настройки времени ожидания нужно иметь для всех вызовов, адресуемых за пределы процесса, и для всех таких вызовов нужно выбирать время ожидания по умолчанию. Зарегистрируйте истечение времени ожидания, найдите причину и соответствующим образом скорректируйте значение времени ожидания.

Предохранители

У вас дома предохранители существуют для защиты электрических устройств от скачков напряжения. Если произойдет такой скачок, предохранитель сработает, защищая дорогостоящие домашние устройства. Предохранитель можно выключить вручную, чтобы отключить электричество в какой-нибудь части дома, что позволит безопасно работать с электропроводкой. В книге Майкла Нигарда (Michael Nygard) *Release It!* (Pragmatic Programmers) показано, как та же идея может творить чудеса, когда используется в качестве защитного механизма для наших программных средств.

Рассмотрим историю, которой я только что поделился. Нижестоящее устаревшее рекламное приложение реагировало очень медленно, пока в конце концов не вернуло ошибку. Даже при правильной настройке времени ожидания до получения ошибки мы томились бы в долгом ожидании. А затем повторили бы попытку при следующем поступлении запроса и снова ждали. Плохо, конечно, что нижестоящий сервис сбоит, но ведь он при этом заставляет и нас замедлить работу.

Предохранитель же срабатывает после конкретного количества безответных запросов к нижестоящему сервису. И пока он находится в этом состоянии, все последующие запросы быстро получают отказ. По истечении определенного времени клиент отправляет несколько запросов, чтобы определить, не восстановился ли нижестоящий сервис, и при получении достаточного количества нормально обслуженных запросов восстанавливает сработавший предохранитель. Обзор подобного процесса показан на рис. 11.2.

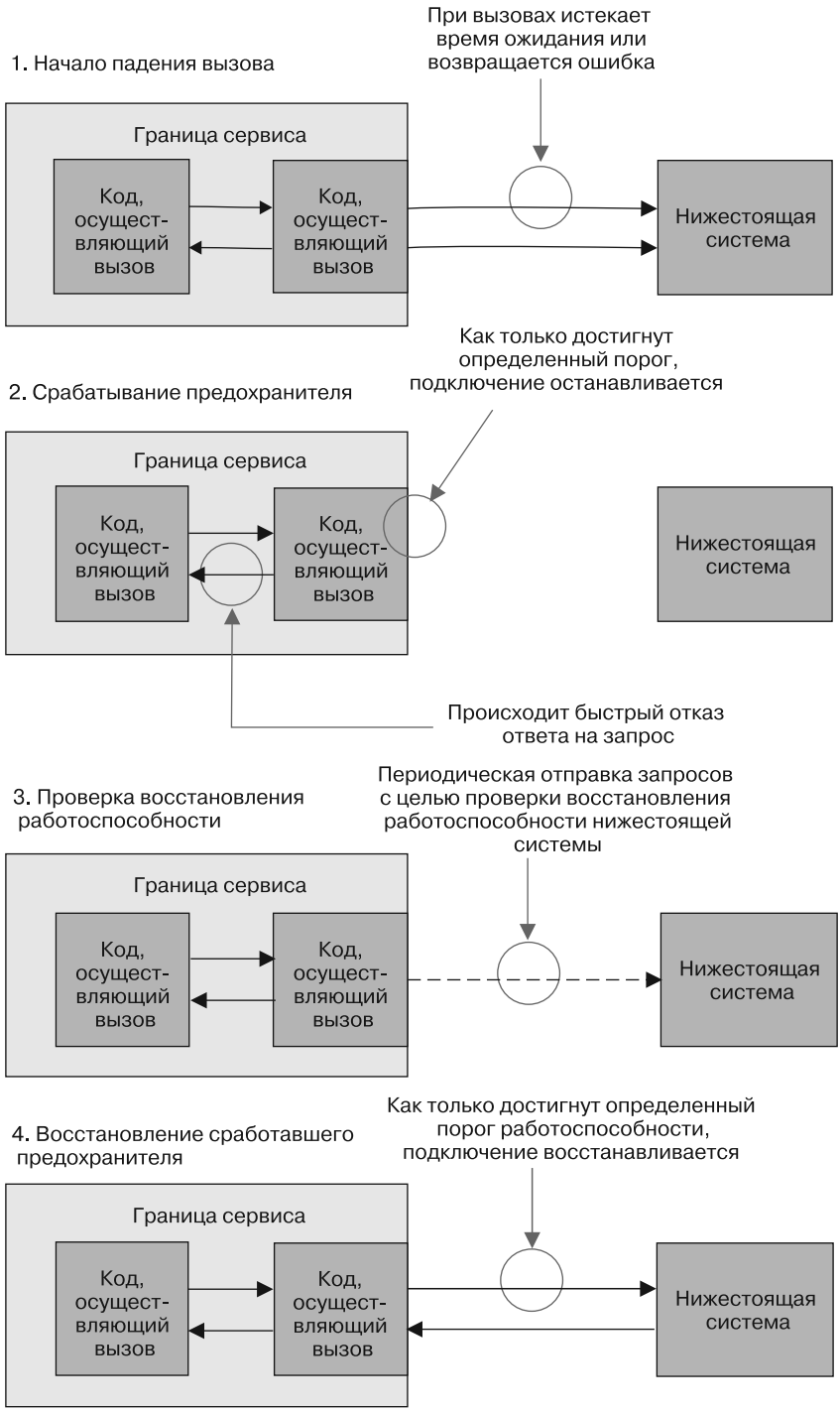


Рис. 11.2. Обзор предохранителей

Конкретная реализация предохранителя зависит от значимости *получившего отказ* запроса, но когда мне приходилось создавать его для HTTP-соединений, за сбой принимались либо истечение времени ожидания, либо возвращаемый код из серии 5xx HTTP. Таким образом, когда нижестоящий ресурс отказывал, или истекало время ожидания, или возвращались коды ошибок, после достижения определенного порога мы автоматически прекращали отправление трафика и быстро констатировали сбой. И в случае нормализации обстановки могли осуществлять повторный запуск в автоматическом режиме.

Установка правильных значений может вызвать затруднения. Вам ведь не хочется, чтобы предохранитель срабатывал слишком быстро, и в то же время не хочется слишком долго ждать его срабатывания. Более того, перед возобновлением отправки трафика хочется убедиться в том, что нижестоящий сервис восстановил нормальную работоспособность. Как и при выборе значений времени ожидания, я устанавливаю разумные параметры по умолчанию и использую их повсеместно, а затем изменяю для каждого конкретного случая.

При срабатывании предохранителя у вас есть выбор из нескольких вариантов. Один из них предполагает выстраивание запросов в очередь с последующей повторной попыткой их отправки. Для некоторых сценариев этот вариант вполне приемлем, особенно если вы выполняете работу, являющуюся частью асинхронного задания. Но если этот вызов был сделан как часть цепочки синхронных вызовов, то лучше будет, наверное, как можно скорее констатировать сбой. Это может означать распространение ошибки вверх по цепочке вызовов или более тонкое снижение уровня функциональности.

Располагая таким механизмом (аналогичным домашним предохранителям), мы можем воспользоваться им вручную, чтобы обезопасить свою работу. Например, если в ходе обычного обслуживания системы нужно выключить микросервис, можно вручную перевести в сработавшее положение предохранители зависимых систем, чтобы они смогли быстро констатировать сбой, пока микросервис находится в отключенном состоянии. После его возвращения в рабочее состояние мы можем восстановить сработавшие предохранители и все должно вернуться в нормальное рабочее состояние.

Переборки

В книге Release It! Нигард дает описание концепции *переборок*, применяемых в качестве способа изоляции от сбоев. В судостроении переборка является составной частью корабля, которую можно закрыть для защиты помещений корабля от поступления забортной воды. Если корабль дает течь, можно закрыть двери переборки. Часть корабля затопится водой, но все остальное будет не тронуто.

В понятиях архитектуры программ можно рассматривать множество различных переборок.

Возвращаясь к моему личному опыту, мы упустили шанс реализации переборки. Нам нужно было для каждого нижестоящего соединения использовать различные пулы соединений. Таким образом, при исчерпании одного пула соединений это не влияло бы на остальные соединения (рис. 11.3). Существовала бы гарантия того, что замедление работы нижестоящего сервиса повлияет только на один пул соединения, позволяя нормально обрабатывать другие вызовы.

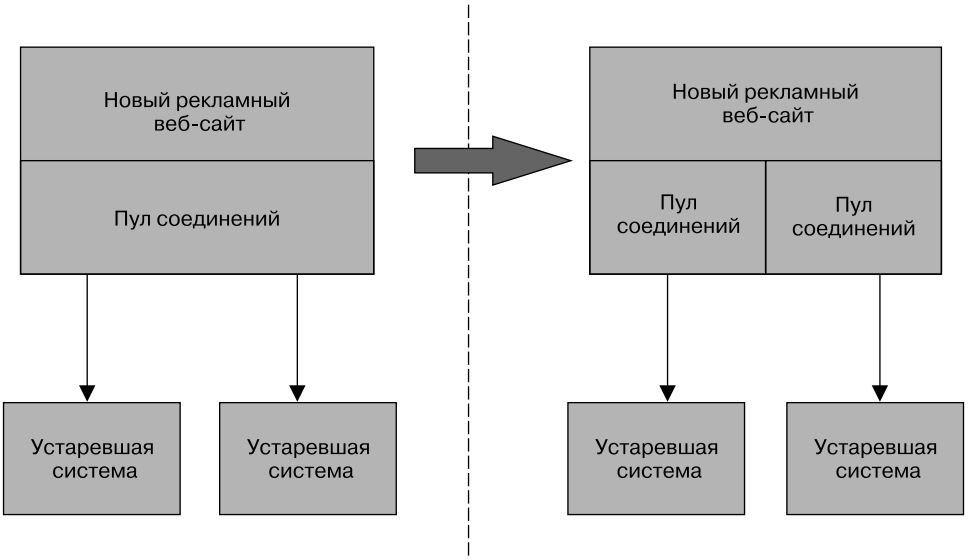


Рис. 11.3. Использование по одному пулу соединения для каждого нижестоящего сервиса с целью создания переборок

Еще одним способом реализации переборок может стать разделение проблем. Разделением функций на отдельные микросервисы можно уменьшить влияние сбоя в одной области на работу других областей.

Внимательно изучите все стороны вашей системы, от которых можно ожидать сбоя, как внутри микросервисов, так и между ними. Установлены ли между ними переборки? Советую начать с разделения пулов соединений, выделяя отдельный пул для каждого нижестоящего соединения. Но можно пойти и дальше и рассмотреть также возможность применения предохранителей.

Можно рассмотреть применение предохранителей в качестве автоматического механизма герметизации перегородки, то есть не только в качестве защиты потребителя от проблем, возникших в нижестоящей системе, но и в качестве потенциальной защиты нижестоящего сервиса от излишних вызовов, способных неблагоприятно воздействовать на него. Учитывая опасность каскадного сбоя, я бы порекомендовал обязательно применять предохранители для всех синхронных вызовов в адрес нижестоящих систем. Но вам не обязательно создавать собственные предохранители. У Netflix есть библиотека Hystrix с абстракцией предохранителя на JVM, поставляемой с эффективной системой мониторинга, есть и другие реализации для различных технологических стеков, например Polly для .NET или миксин circuit_breaker для Ruby.

Во многих отношениях переборки являются наиболее важной из этих трех схем. Настройки времени ожидания и предохранители помогают вам высвободить ресурсы при их истощении, а переборки в первую очередь могут обеспечить невозможность их истощения. Библиотека Hystrix, к примеру, позволяет реализовать переборки, которые фактически при определенных условиях отклоняют запросы, обеспечивая тем самым дальнейшее истощение ресурсов; этот прием называется

сбросом нагрузки. Иногда отклонение запроса является наилучшим способом уберечь важную систему от перегрузки и превращения в узкое место для множества вышестоящих сервисов.

Изолированность

Чем больше один сервис зависит от задействования других сервисов, тем больше благополучная работа одного сервиса влияет на выполнение задач другими сервисами. Использование технологий интеграции, позволяющих переводить нижестоящий сервер в режим автономной работы, может снизить вероятность влияния простоев, как плановых, так и внеплановых сбоев на вышестоящие сервисы.

Повышение изолированности сервисов друг от друга дает еще одно преимущество, заключающееся в существенно меньших потребностях в координации усилий между владельцами сервисов. Чем меньше эти потребности, тем большей автономностью будут обладать команды и тем больше возможность свободно распоряжаться своими сервисами и развивать их.

Идемпотентность

При проведении *идемпотентных* операций результат после первого применения не меняется, даже если операция последовательно выполняется еще несколько раз. Если операции обладают идемпотентностью, мы можем повторять вызов несколько раз без негативного воздействия. Это нам очень пригодится, если необходимо повторно воспроизвести сообщения, когда нет уверенности, что они обработаны. Это является весьма распространенным способом восстановления после ошибок.

Рассмотрим простой вызов с целью добавления баллов в результате размещения заказа одним из наших клиентов. Мы можем сделать вызов с полезной нагрузкой, показанной в примере 11.1.

Пример 11.1. Зачисление баллов на счет

```
<credit>
  <amount>100</amount>
  <forAccount>1234</account>
</credit>
```

Если этот вызов будет получен несколько раз, мы столько же раз зачислим 100 баллов. Получается, этот вызов не является идемпотентным. Но как показано в примере 11.2, при наличии дополнительной информации можно позволить банку бонусных баллов сделать этот вызов идемпотентным.

Пример 11.2. Добавление информации к зачислению баллов с целью придания идемпотентности этой операции

```
<credit>
  <amount>100</amount>
  <forAccount>1234</account>
  <reason>
```

```
    <forPurchase>4567</forPurchase>
  </reason>
</credit>
```

Теперь мы знаем, что это зачисление относится к конкретному заказу под номером 4567. Учитывая, что получить бонус за конкретный заказ можно только единожды, мы можем применить зачисление еще раз без увеличения общего количества баллов.

Этот механизм работает и при организации сотрудничества на основе событий и может быть особенно полезен при наличии нескольких экземпляров одного и того же вида сервиса, подписанного на события. Даже при сохранении сведений о том, какие события были обработаны, при некоторых формах доставки асинхронных сообщений могут создаваться небольшие окна, в которых одно и то же сообщение может попадать в поле зрения двух исполнителей. Обрабатывая события идемпотентным образом, мы гарантируем, что они не станут источником ненужных проблем.

Некоторые недопонимают эту концепцию, полагая, что последующие вызовы с такими же параметрами не смогут оказывать *какого-либо* влияния, оставляя нас в интересном положении. Например, нам по-прежнему хотелось бы, чтобы вызов был получен нашими журналами. Нужно записать время отклика на вызов и собрать данные для мониторинга. Ключевым моментом здесь является то, что рассматриваемые бизнес-операции, которые мы считаем идемпотентными, не распространяются на полное состояние системы.

Некоторые HTTP-глаголы, например GET и PUT, определены в HTTP-спецификации в качестве идемпотентных, но, чтобы это произошло, сервис должен обрабатывать их идемпотентным образом. Если вы начнете отказывать им в идемпотентности, а вызывающая сторона будет уверена в безопасности их повторного применения, может возникнуть запутанная ситуация. Следует запомнить, что сам факт использования HTTP в качестве основного протокола еще не означает, что вы все получите, не прилагая дополнительных усилий!

Масштабирование

В основном масштабирование систем выполняется по двум причинам. Во-первых, для того, чтобы легче было справиться со сбоями: если мы переживаем за отказ какого-либо компонента, то помочь сможет наличие такого же дополнительного компонента, не так ли? Во-вторых, для повышения производительности, что позволяет либо справиться с более высокой нагрузкой, либо снизить время отклика, либо достичь обоих результатов. Рассмотрим ряд наиболее распространенных технологий масштабирования, которыми можно будет воспользоваться, и подумаем об их применении к архитектурам микросервисов.

Наращивание мощностей

От наращивания мощностей некоторые операции могут только выиграть. Более объемный корпус с более быстрым центральным процессором и более эффективной

подсистемой ввода-вывода зачастую способны уменьшить задержки и повысить пропускную способность, позволяя выполнять больший объем работ за меньшее время. Но такая разновидность масштабирования, которую часто называют *вертикальным масштабированием*, может быть слишком затратной: иногда один большой сервер может стоить намного больше, чем два небольших сервера сопоставимой мощности, особенно когда вы начнете получать по-настоящему большие машины. Иногда само программное обеспечение не способно освоить доступные дополнительные ресурсы. Более крупные машины зачастую предоставляют в наше распоряжение больше ядер центральных процессоров, но подчас у нас нет программных средств, позволяющих использовать такое преимущество. Еще одна проблема заключается в том, что такая разновидность масштабирования не в состоянии внести весомый вклад в устойчивость сервера, если у нас только одна машина! Несмотря на это, такой подход может принести быстрый выигрыш, особенно если вы используете провайдер виртуализации, позволяющий легко и просто изменять объемы ресурсов виртуальных машин.

Разделение рабочих нагрузок

Как отмечалось в главе 6, наличие единственного микросервиса на каждом хосте, безусловно, предпочтительнее модели, предусматривающей наличие на хосте сразу нескольких микросервисов. Но изначально с целью снижения стоимости оборудования или упрощения управления хостом (хотя это спорная причина) многие принимают решение о сосуществовании нескольких микросервисов на одной физической машине. Поскольку микросервисы запускаются в независимых процессах, обменивающихся данными по сети, задача последующего их перемещения на собственные хосты с целью повышения пропускной способности и масштабирования не представляет особой сложности. Такое перемещение может повысить устойчивость системы, поскольку сбой одного хоста повлияет на ограниченное количество микросервисов.

Разумеется, мы могли бы воспользоваться необходимостью расширения масштаба для разбиения существующих микросервисов на части, чтобы успешнее справляться с нагрузкой. В качестве упрощенного примера представим, что наш сервис счетов позволяет создавать индивидуальные финансовые счета клиентов и управлять ими, а также выставляет API-интерфейс для запуска запросов с целью создания отчетов. Такая возможность запуска запросов существенно нагружает систему. Объем запросов считается некритическим и не требует сохранения поступающего за день потока запросов. Но возможность управления финансовыми записями является критической для наших пользователей, и мы не можем позволить, чтобы она функционировала со сбоями. Разделяя эти две возможности на отдельные сервисы, мы уменьшаем нагрузку на сервис важных счетов и вводим новый сервис отчета по счетам, спроектированный с учетом не только возможностей обработки запросов (возможно, с использованием технологий, рассмотренных в главе 4), но и того, что некритическую систему не обязательно развертывать в отказоустойчивом режиме, как того требует основной сервис счетов.

Распределение риска

Один из способов масштабирования с целью повышения отказоустойчивости заключается в выдаче гарантий того, что все яйца не сложены в одну корзину. Простейшим примером может послужить обеспечение того, что вы не поместили на одном хосте сразу несколько сервисов, где сбой окажет влияние на работу сразу нескольких сервисов. Но рассмотрим значение такого понятия, как *хост*. В большинстве возможных нынче ситуаций хост фактически является виртуальным понятием. Тогда что, если все мои сервисы находятся на разных хостах, но все эти хосты фактически являются виртуальными, запущенными на одной и той же физической машине? Если машина даст сбой, я могу потерять сразу несколько сервисов. Для уменьшения вероятности этого некоторые платформы виртуализации дают гарантии распределения хоста по нескольким физическим машинам.

Для внешних платформ виртуализации существует распространенная практика размещения корневого раздела виртуальной машины в одной сети хранения данных (SAN). Если SAN-сеть даст сбой, это может привести к сбою всех связанных с ее помощью виртуальных машин. SAN-сети отличаются масштабностью, дороговизной и проектированием в расчете на бессбойную работу. Я сталкивался со сбоями больших и дорогостоящих SAN-сетей как минимум дважды за последние десять лет, и каждый раз это имело довольно серьезные последствия.

Еще одной распространенной разновидностью сокращения вероятности сбоев является гарантия того, что не все ваши сервисы запускаются на одной и той же стойке дата-центра, или того, что сервисы распределены по более чем одному дата-центру. Если вы имеете дело с основным поставщиком услуг, то важно знать о предложении и планировании им соответствующего соглашения об уровне предоставления услуг (SLA). Если для вас допустимо не более четырех часов сбоев в квартал, а хостинг-провайдер может гарантировать всего лишь не более восьми часов, то вам придется либо пересмотреть SLA, либо придумать альтернативное решение.

AWS, к примеру, имеет региональную форму распределения, которую можно рассматривать как отдельные облака. Каждый регион, в свою очередь, разбит на две и более зоны доступности (AZ). Эти зоны в AWS являются эквивалентом дата-центра. Важно, чтобы сервисы были распределены по нескольким зонам доступности, поскольку инфраструктура AWS не дает гарантий доступности отдельно взятого узла или даже всей зоны доступности. Для своих вычислительных услуг эта инфраструктура предлагает только 99,95 % безотказной работы за заданный месячный период во всем регионе, поэтому внутри отдельно взятого региона рабочую нагрузку следует распределить по нескольким доступным зонам. Некоторых такие условия не устраивают, и вместо этого они запускают свои сервисы, также распределяя их по нескольким регионам.

Конечно же, нужно отметить, что провайдеры, давая вам *SLA-гарантии*, будут стремиться ограничить свою ответственность! Если несоблюдение условий с их стороны будет стоить вам клиентов и больших денежных потерь, нужно внимательно изучить контракты на предмет компенсаций. Поэтому я настоятельно рекомендую оценить влияние несоблюдения поставщиком взятых перед вами обяза-

тельств и подумать, стоит ли иметь про запас какой-нибудь план Б (или В). Мне, к примеру, приходилось работать не с одним клиентом, у которого имелась хостинг-платформа аварийного восстановления, использующая услуги другого поставщика, что снижало уязвимость от ошибок одной компании.

Балансировка нагрузки

Когда сервису нужна отказоустойчивость, вам понадобятся способы обхода критических мест сбоев. Для типичного микросервиса, выставляющего синхронную конечную HTTP-точку, наиболее простым способом решения этой задачи (рис. 11.4) будет использование нескольких хостов с запущенными на них экземплярами микросервиса, находящимися за балансировщиком нагрузки. Потребители микросервиса не знают, связаны они с одним его экземпляром или с сотней таких экземпляров.

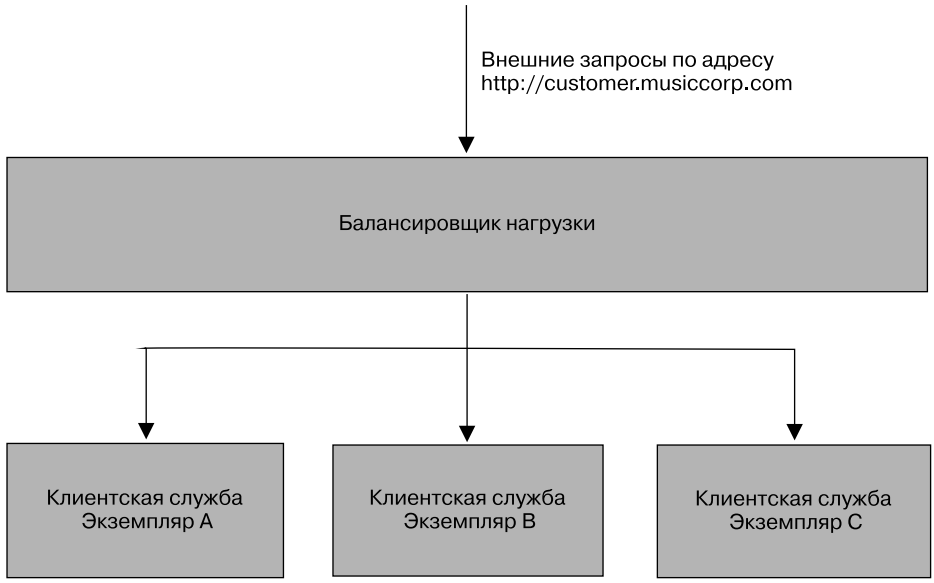


Рис. 11.4. Пример балансировки нагрузки с целью масштабирования количества экземпляров клиентского сервиса

Существуют балансировщики нагрузки всех форм и размеров, от больших и дорогих аппаратных приспособлений до балансировщиков на основе программных средств типа mod_proxy. У всех них общие основные возможности. Они распределяют поступающие к ним вызовы между несколькими экземплярами на основе определенного алгоритма, устраняя экземпляры, утратившие работоспособность, но не теряя при этом надежды на их возвращение при восстановлении нормального режима работы.

Некоторые балансировщики нагрузки предоставляют ряд полезных свойств. Одним из самых распространенных является возможность работы в качестве *оконечного SSL-устройства*, в котором входящие в балансировщик нагрузки

HTTPS-соединения преобразуются в HTTP-соединения, в качестве которых они и попадают в сам экземпляр. Исторически издержки на управление SSL были довольно большими, и когда этот процесс берет на себя балансировщик нагрузки, он оказывает вам существенную услугу. В настоящее время это сильно упрощает настройки отдельных хостов, на которых запускаются экземпляры. Но, как говорилось в главе 9, смысл использования HTTPS заключается в обеспечении неуязвимости запросов от взлома злоумышленниками на маршруте их передачи, поэтому при использовании конечной точки SSL мы потенциально отчасти открываем свои данные. Снизить угрозу можно, поместив все экземпляры микросервисов в единую VLAN-сеть (рис. 11.5). VLAN является виртуальной локальной сетью, изолированной таким образом, что поступающие извне запросы могут пройти только через маршрутизатор, а в данном случае маршрутизатор является также балансировщиком нагрузки с возможностью выполнения роли конечной точки SSL. Единственная линия связи с микросервисами, идущая с внешней стороны VLAN-сети, использует протокол HTTPS, а внутри сети повсеместно применяется протокол HTTP.

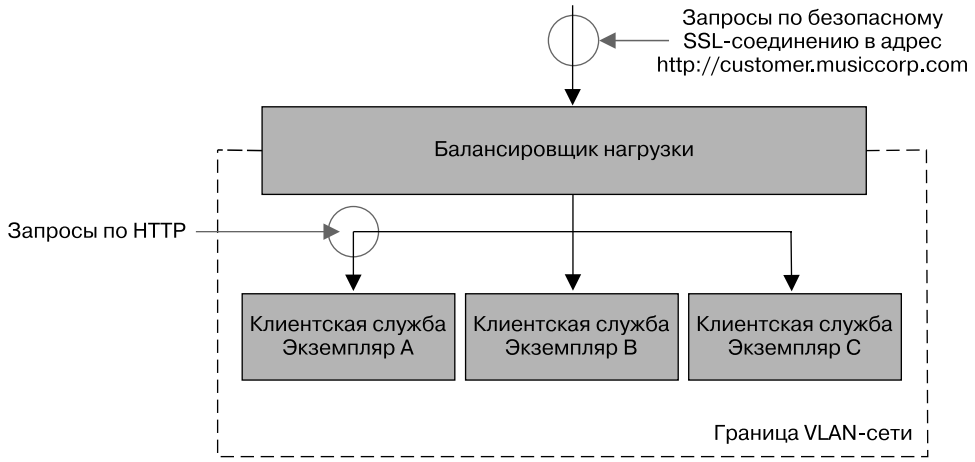


Рис. 11.5. Использование конечной точки HTTPS в балансировщике нагрузки с VLAN-сетью с целью повышения безопасности

AWS предоставляет балансировщики нагрузок с конечными точками HTTPS в форме ELB-балансировщиков (Elastic Load Balancer — гибкий балансировщик нагрузки), позволяющие для реализации VLAN-сети воспользоваться его безопасными группами или виртуальными закрытыми облаками (VPC). Такую же роль в качестве программного балансировщика нагрузки может сыграть программа вроде mod_proxy. У многих организаций имеются аппаратные балансировщики нагрузки, автоматизация которых может быть затруднена. При таких обстоятельствах я встану на защиту программных балансировщиков нагрузки, установленных *за* аппаратными балансировщиками, что даст командам свободу их перенастройки в соответствии с их запросами. Надо считаться с тем фактом, что аппаратные балансировщики нагрузки нередко выходят из строя, становясь единой точкой отказа!

Независимо от избранного вами подхода при рассмотрении вопроса о конфигурации балансировщика нагрузки относитесь к ней так же, как относились к конфигурации вашего сервиса: обеспечьте ее сохранение в системе управления версиями и возможность автоматического применения.

Балансировщики нагрузки позволяют нам добавлять дополнительные экземпляры микросервисов незаметно для любых потребителей сервиса. Это дает нам более широкие возможности управления нагрузкой и в то же время уменьшает влияние сбоя на одном из хостов. Но у многих, если не у большинства микросервисов будут какие-нибудь постоянные хранилища данных, возможно, база данных, находящаяся на другой машине. При наличии нескольких экземпляров микросервисов на разных машинах, но только одного хоста с запущенным экземпляром базы данных мы обрекаем эту базу данных на роль единого источника сбоев. Схемы, позволяющие справиться с этой проблемой, будут рассмотрены чуть позже.

Системы на основе исполнителей

Применение балансировщиков не является единственным способом разделения нагрузки среди нескольких экземпляров сервиса и уменьшения их хрупкости. В зависимости от характера операций столь же эффективной может быть и система на основе исполнителей. Здесь вся коллекция экземпляров действует с некоторым общим отставанием в работах. Это может быть целый ряд Hadoop-процессов или, возможно, некоторое количество процессов, прослушивающих общую очередь работ. Операции такого типа хорошо подходят для формирования пакетов работ или асинхронных заданий. Подумайте в данном ключе о таких задачах, как обработка миниатюр изображений, отправка электронной почты или создание отчетов.

Эта модель также хорошо работает при *пиковых* нагрузках, где по мере возрастания потребностей могут запускаться дополнительные экземпляры для соответствия поступающей нагрузке. Пока сама очередь работ будет сохранять устойчивость, эта модель может использовать масштабирование для повышения как пропускной способности работ, так и отказоустойчивости, поскольку становится проще справиться с влиянием отказавшего (или отсутствующего) исполнителя. Работа займет больше времени, но ничего при этом не потеряется.

Я видел, как это вполне успешно работает в организациях, имеющих в определенные периоды времени большие объемы незадействованных вычислительных мощностей. Например, по ночам для работы системы электронной торговли все имеющиеся машины вам не нужны, поэтому временно их можно задействовать для выполнения заданий по созданию отчетов.

Хотя в системах на основе исполнителей самим исполнителям высокая надежность и не нужна, система, содержащая предназначенную для выполнения работу. должна быть надежной. Справиться с этим можно, к примеру запустив круглосуточный брокер сообщений или такую систему, как Zookeeper. Преимущества такого подхода заключаются в том, что при использовании для достижения этих целей существующих программных средств наиболее сложную задачу за нас выполняет кто-то другой. Но нам по-прежнему требуется знать, как настроить и обслуживать эти системы, добиваясь от них безотказной работы.

Начинаем все заново

Архитектура, использовавшаяся сначала, может не стать архитектурой, используемой в дальнейшем, когда вашей системе придется обрабатывать совершенно разные объемы нагрузки. Как Джеф Дин (Jeff Dean) говорил в своей презентации *Challenges in Building Large-Scale Information Retrieval Systems* (конференция WSDM 2009), вы должны «закладывать в конструкцию возможность десятикратного роста, но планировать ее перезапись под стократный рост». Для поддержки следующего уровня роста в определенные моменты придется делать нечто весьма радикальное.

Вспомним историю Gilt, которую мы уже затрагивали в главе 6. В течение двух лет Gilt вполне устраивало монолитное Rails-приложение. Бизнес развивался успешно, что означало увеличение количества клиентов и рост объема нагрузки. В определенный переломный момент, чтобы справиться с возросшей нагрузкой, компании пришлось переделать приложение.

Переделка может означать разбиение монолита на части, что и было сделано для Gilt. Или выбор новых хранилищ данных, которые смогли бы лучше справиться с нагрузкой, что и будет рассмотрено нами в ближайшем будущем. Это также может означать применение новых технологий, например переход с синхронных систем «запрос — ответ» к системам на основе событий, применение новых платформ развертывания, смену всех технологических стеков или применение сразу всех этих нововведений.

Есть опасение, что люди поймут необходимость изменения архитектуры в момент достижения определенного порога масштабирования и примут это за предлог для создания с нуля системы под более широкий масштаб. Это может иметь весьма пагубные последствия. Запуская новый проект, мы зачастую не знаем в точности, что именно хотим создать, и не знаем, будет ли он успешен. Нам нужно иметь возможность быстрого проведения эксперимента, чтоб понять, какие функциональные возможности следует создавать. Если изначально попытаться создать систему под широкий масштаб, мы сразу же получим большой объем работ для обеспечения готовности к нагрузке, которой может никогда и не быть. Это отвлечет силы от более важных действий, например от выяснения того, захочет ли кто-нибудь вообще воспользоваться нашим продуктом. Эрик Рис (Eric Ries) рассказывал историю о том, как шесть месяцев было потрачено на создание продукта, который никто даже не загрузил. Он размышлял, что можно было даже установить ссылку на несуществующую веб-страницу, при щелчке на которой люди получали бы сообщение об ошибке 404, чтобы посмотреть, была ли вообще потребность в продукте, а вместо работы провести шесть месяцев на пляже и при этом получить более весомый результат!

Потребность во внесении в систему изменений, позволяющих ей справиться с расширением масштаба, нельзя считать провалом. Нужно считать это признаком успеха.

Масштабирование баз данных

Масштабирование микросервисов без сохранения состояния производится довольно просто. А что делать, если мы сохраняем данные в базе данных? Нам нужно знать,

как выполнять масштабирование и в таком случае. Различные типы баз данных требуют разных форм масштабирования, и понимание того, какая из этих форм подойдет наилучшим образом именно для вашего случая, гарантирует выбор нужной технологии баз данных с самого начала.

Доступность сервиса против долговечности данных

Изначально важно отделить понятие доступности сервиса от понятия долговечности самих данных. Нужно разобраться в том, что это два разных понятия и поэтому для них будут использоваться разные решения.

Например, я могу хранить копию всех данных, записанных в мою базу данных, в отказоустойчивой файловой системе. Если база данных откажет, данные не пропадут, поскольку у меня есть копия, но сама база данных станет недоступной, что может привести также к недоступности моего микросервиса. В более обобщенной модели будут задействованы резервы. Все данные, записанные в основную базу данных, будут копироваться в резервную базу данных, являющуюся точной копией основной. Если основная база данных даст сбой, мои данные окажутся в безопасности, но без механизма, который либо их вернет, либо повысит резервную базу до статуса основной, доступной базы данных у нас не будет, хотя сами данные будут в безопасности.

Масштабирование для считываний

Многие сервисы в основном занимаются считыванием данных. Вспомним сервис каталогов, который хранит информацию для выставленных на продажу товарных позиций. Мы добавляем записи для новых товарных позиций на весьма нерегулярной основе, и совсем неудивительно, что на каждую запись в каталог мы имеем по 100 считываний данных нашего каталога. К счастью, масштабирование для чтения дается значительно легче масштабирования для записи. Здесь большую роль может сыграть кэширование данных, которое вскоре будет рассмотрено более подробно. Еще одна модель предусматривает использования *реплик чтения*.

В системе управления реляционными базами данных (RDBMS), подобной MySQL или Postgres, данные могут копироваться из основного узла в одну или несколько реплик. Зачастую это делается с целью обеспечения безопасного хранения копий данных, но может использоваться также для распределения операций чтения. Сервис может направлять все запросы на запись к единственному основному узлу, но при этом распределять запросы на чтение между несколькими репликами, предназначенными для считывания данных (рис. 11.6). Резервное копирование из основной базы данных к репликам происходит через некоторое время после записи. Это означает, что при такой технологии считывания до завершения репликации данные могут быть *устаревшими*. Со временем операциям чтения станут доступны согласующиеся данные. Подобная настройка называется *согласованностью, возникающей по прошествии некоторого времени*, и если вы в состоянии справиться с временной несогласованностью, то ее можно признать довольно простым и весьма распространенным способом, содействующим масштабированию систем. Вскоре, когда дойдет очередь до теоремы CAP, мы рассмотрим его более подробно.

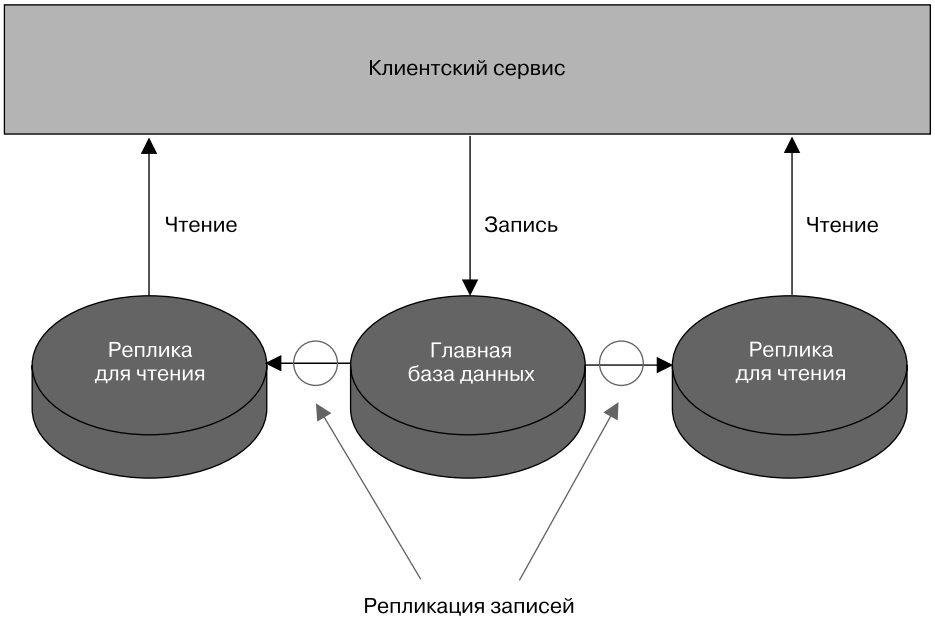


Рис. 11.6. Использование реплик для чтения с целью масштабирования операций считывания данных

Мода на использование реплик чтения в целях масштабирования появилась много лет назад, но сегодня я бы порекомендовал вам присмотреться в первую очередь к кэшированию, от которого можно получить существенно больше преимуществ с точки зрения производительности, зачастую потратив на это меньше времени и сил.

Масштабирование для производства записей

Масштабирование чтения дается сравнительно легко. А как насчет записей? Один из подходов предусматривает применение *фрагментации*. При этом используются несколько узлов базы данных. Берется часть данных, подлежащих записи, к ним применяется некая функция хеширования для получения ключа данных, и на основании результата работы функции определяется, куда эти данные отправлять. Рассмотрим весьма упрощенный (и совсем негодный) пример: представим, что клиентские записи диапазона A — M попадают в один экземпляр базы данных, а записи диапазона N — Z — в другой. Этим можно управлять самостоятельно в своем приложении, но некоторые базы данных, например Mongo, многое в этом плане делают за вас.

Сложность с фрагментацией операций записи данных заключается в управлении запросами. Когда смотришь на отдельно взятую запись, все представляется несложным, поскольку можно просто применить функцию хеширования, чтобы найти нужный экземпляр данных, а затем извлечь его из соответствующего фрагмента базы. А как быть с запросами, данные которых разбросаны по не-

скольким узлам, например предписывающими найти всех клиентов старше 18 лет? Если требуется запросить все фрагменты базы, то нужно либо запросить каждый отдельно взятый фрагмент и объединить ответы в памяти, либо иметь альтернативное хранилище для чтения, в котором доступны данные из обоих наборов. Зачастую отправкой запросов, распространяющихся на несколько фрагментов, управляет асинхронный механизм с использованием кэшируемых результатов. Например, в Mongo для выполнения таких запросов используются задания отображения и свертки (map/reduce jobs).

При использовании фрагментированных систем возникает вопрос: что случится, если понадобится добавить еще один узел базы данных? В прошлом для этого нужен был довольно длительный простой, особенно для крупных кластеров, поскольку могла потребоваться остановка работы всей базы данных и перебалансировка хранящейся в ней информации. Совсем недавно во многих системах появилась поддержка добавления фрагментов к не прекращающей работу системе, в которой перебалансировка данных осуществляется в фоновом режиме. К примеру, Cassandra справляется с этим очень хорошо. Добавление фрагментов к существующим кластерам — занятие не для слабых духом, так что все нужно тщательно проверить.

Фрагментация для операций записи может позволить увеличить масштаб, чтобы справиться с объемом записей, но при этом нисколько не улучшить отказоустойчивость. Если клиентские записи в диапазоне A — M всегда попадают в экземпляр X и экземпляр X окажется недоступен, доступ к записям A — M может быть утрачен. Здесь Cassandra предлагает дополнительные возможности, позволяющие гарантировать репликацию данных на несколько узлов *кольца* (так в Cassandra называется коллекция узлов).

Как можно было бы предположить на основании этого краткого обзора, масштабирование баз данных для проведения операций записи является той самой областью, где все слишком сильно усложняется и возможности различных баз данных начинают очень разниться. Мне часто встречались люди, которые меняют технологию баз данных, как только начинают сталкиваться с ограничениями, не позволяющими простым способом масштабировать имеющиеся у них объемы записи. Если такое произойдет и с вами, то зачастую самым быстрым способом решения проблемы станет приобретение более мощного оборудования, но при этом нужно все же присматриваться к таким системам, как Cassandra, Mongo или Riak, с целью поиска альтернативных моделей масштабирования, предлагающих более подходящие долговременные решения.

Совместно используемые инфраструктуры баз данных

В некоторых типах баз данных, например в традиционных RDBMS, понятия самой базы данных и схемы разделены. Это означает, что одна запущенная база данных может иметь несколько независимых схем, по одной для каждого микросервиса. Это может существенно помочь в сокращении количества машин, необходимых для запуска системы, но при этом появится весьма серьезная единая точка отказа.

Если такая инфраструктура базы данных даст сбой, это может повлиять сразу на несколько микросервисов, что потенциально может привести к катастрофическому перебою в работе. Если же вы воспользуетесь подобным типом настроек, следует обязательно оценить все риски. И нужно быть абсолютно уверенными в максимально возможной отказоустойчивости самой базы данных.

CQRS

Схема разделения ответственности на команды и запросы (CQRS) относится к альтернативной модели хранения и запроса информации. При использовании обычных баз данных одна и та же система применяется как для внесения изменений в данные, так и для запроса этих данных. А при использовании CQRS часть системы имеет дело с командами, которые отлавливают запросы на изменение состояния, в то время как другая часть системы работает с запросами на извлечение данных.

Команды бывают с запросами на внесение изменений в состояние. Эти команды проходят проверку, и если они работают, их применяют к модели. В командах должна содержаться информация об их намерениях. Они могут обрабатываться в синхронном или асинхронном режиме, позволяя различным моделям управлять масштабированием. К примеру, входящие запросы можно просто выстроить в очередь и обработать их чуть позже.

Ключевым выводом для нас является то, что внутренние модели, используемые для обработки команд и запросов, сами являются полностью разделенными. Например, я могу выбрать обработку и выполнение команд как событий, возможно, просто сохраняя команды в хранилище данных (этот процесс известен как подбор *источников событий*). Моя модель запроса может запрашивать источник событий и создавать прогнозы на основе сохраненных событий для сбора сведений о состоянии объектов домена или же просто забрать нужное из командной части системы для обновления различных типов хранилищ данных. Во многих отношениях мы получаем те же преимущества, что и от рассмотренных ранее реплик чтения, но при этом не требуется, чтобы резервное хранилище для реплик было таким же, как и хранилище данных, используемое для обработки изменений данных.

Такая форма разделения позволяет иметь различные типы масштабирования. Части нашей системы, занимающиеся командами и запросами, могут находиться в разных сервисах или на разном оборудовании и пользоваться совершенно разными типами хранилищ данных. Это может открыть для нас массу способов масштабирования. Можно даже поддерживать различные типы форматов чтения, имея несколько реализаций той части, которая занимается запросами, с возможной поддержкой графического представления данных или формы данных на основе пар «ключ — значение».

Но при этом следует иметь в виду, что схема такого рода весьма сильно отходит от модели, в которой все CRUD-операции обрабатываются в едином хранилище данных. Мне приходилось наблюдать, как несколько весьма опытных команд разработчиков бились над получением приемлемых результатов от использования данной схемы!

Кэширование данных

Кэширование довольно часто используется для оптимизации производительности посредством сохранения предыдущего результата какой-нибудь операции с тем, чтобы последующие запросы могли использовать это сохраненное значение, не затрачивая времени и ресурсов на повторное вычисление значения. В большинстве случаев кэширование проводится с целью исключения необходимости выполнения полного цикла обращения к базе данных или другим сервисам для ускорения обслуживания результата. При правильном использовании из кэширования можно извлечь огромные преимущества, выражающиеся в повышении производительности. Причина хорошего масштабирования технологии HTTP под обработку большого количества запросов кроется во встроенной концепции кэширования.

Даже при использовании единого монолитного веб-приложения вариантов того, где и как проводить кэширование, совсем немного. При использовании архитектуры микросервисов, где каждый сервис имеет собственные источник данных и поведение, мы располагаем намного большим количеством вариантов того, где и как проводить кэширование. При использовании распределенных систем кэширование задумывается либо на стороне клиента, либо на стороне сервера. Но какое лучше?

Кэширование на стороне клиента, прокси-сервере и стороне сервера

При кэшировании на стороне клиента результат кэширования сохраняется клиентом. Именно клиент решает, когда обращаться за свежей копией и нужно ли это делать. В идеале нижестоящий сервис будет давать подсказки, помогающие клиенту понять, что делать с ответом, чтобы он знал, когда выполнять новый запрос и надо ли это вообще. При прокси-кэшировании прокси-сервер помещается между клиентом и сервером. Хорошим примером может послужить использование обратного прокси-сервера или сети доставки контента (CDN). При кэшировании на стороне сервера ответственность за кэширование возлагается на сервер, возможно, с использованием таких систем, как Redis, или Memcache, или даже простой организацией кэша в памяти.

Какой из вариантов является наиболее подходящим, зависит от того, что вы пытаетесь оптимизировать. Кэширование на стороне клиента может помочь существенно сократить количество сетевых вызовов и стать одним из самых быстрых способов уменьшения нагрузки на нижестоящий сервис. При этом ответственность за порядок проведения кэширования возлагается на клиента, и если необходимо изменить этот порядок, развертывание изменений для нескольких потребителей может вызвать затруднения. Могут возникнуть трудности и с аннулированием просроченных данных, хотя вскоре мы рассмотрим некоторые применяемые для этого механизмы копирования.

При кэшировании с использованием прокси-сервера процесс непрозрачен как для клиента, так и для сервера. Зачастую это является весьма простым способом добавления кэширования к уже существующей системе. Если прокси-сервер создан

для кэширования общего трафика, он может также заниматься кэшированием для нескольких сервисов. Наиболее распространенным примером может послужить обратный прокси-сервер наподобие Squid или Varnish, который может кэшировать любой HTTP-трафик. Наличие прокси-сервера между клиентом и сервером становится причиной дополнительных сетевых скачков трафика, хотя в моей практике проблемы из-за этого возникали крайне редко, поскольку оптимизация производительности, осуществляемая благодаря кэшированию, перевешивала любые дополнительные сетевые издержки.

При кэшировании на стороне сервера процесс непрозрачен для клиентов, и им не о чем волноваться. Когда кэширование проводится близко к границе сервиса или внутри нее, это может упростить такие действия, как аннулирование данных или отслеживание и оптимизация кэш-попаданий. Когда имеется несколько типов клиентов, кэширование на стороне сервера может стать самым быстрым способом повышения производительности.

Для каждого общедоступного сайта, над которым мне приходилось работать, в итоге мы использовали сочетание всех трех подходов. Но бывали и такие распределенные системы, в которых удавалось обойтись вообще без кэширования. Но все это сводилось к знанию того, с какой нагрузкой следует справиться, насколько свежими должны быть ваши данные и что именно сейчас может сделать ваша система. Поначалу нужно просто понять, что в вашем распоряжении имеется целый ряд различных инструментальных средств.

Кэширование при использовании технологии HTTP

HTTP предоставляет ряд весьма полезных средств управления, помогающих нам проводить кэширование либо на стороне клиента, либо на стороне сервера, в которых было бы полезно разобраться, даже если сами вы не пользуетесь HTTP.

Во-первых, при применении HTTP в ответах клиентам мы можем воспользоваться инструкциями по управлению кэшированием — `cache-control`. В них клиентам говорится о том, должны ли они вообще использовать кэширование ресурсов, и если должны, то как следует проводить кэширование в секундах. У нас также есть возможность настройки заголовка `Expires` (истечения срока годности), где вместо указания длины кэшируемого содержимого указываются время и дата, при наступлении которых ресурс должен считаться несвежим и подлежащим повторному извлечению. Какая из настроек вам больше подойдет, зависит от характера совместно используемых ресурсов. Для стандартного статичного содержимого сайта, например для CSS или изображении зачастую хорошо подходит простая инструкция `cache-control` с указанием времени жизни информации (TTL). Однако, если заранее известно, когда поступит новая версия ресурса, разумнее будет воспользоваться Expires-заголовком. В первую очередь все это может принести большую пользу и избавить клиента от необходимости отправки запроса на сервер.

Кроме инструкций `cache-control` и использования Expires-заголовков, в нашем арсенале полезных свойств HTTP есть еще один вариант — метки объектов (Entity Tags, или ETags). ETag используется для определения того, изменилось значение ресурса или нет. Если я обновляю запись клиента, URI ресурса остается прежним,

но значение становится другим, поэтому я буду ожидать изменения ETag. Эффективность этого приема проявляется при использовании *условных GET-запросов*. При отправке GET-запроса можно указать дополнительные заголовки, сообщая сервису о необходимости отправки нам ресурса только в том случае, если он отвечает ряду критериев.

Представим, к примеру, что мы извлекаем клиентскую запись и ее ETag возвращается в виде `o5t6fkd2sa`. Чуть позже, возможно, из-за того, что инструкция по управления кэшированием данных предписывает нам, что ресурс должен рассматриваться как несвежий, мы хотим убедиться в том, что получаем самую свежую версию. При выдаче последующего GET-запроса мы можем передать условие об извлечении данных в случае несовпадения ETag — `If-None-Match: o5t6fkd2sa`. Тем самым серверу сообщается, что нам нужен ресурс по указанному URI, если у него уже имеется ETag, значение которого не соответствует указанному. Если уже есть новейшая версия, сервис отправляет нам ответ `304 Not Modified`, обозначающий, что мы уже располагаем самой последней версией. Если доступна более свежая версия, мы получаем ответ `200 OK` с изменившимся ресурсом и новой меткой ETag для него.

Тот факт, что эти средства управления встроены в столь широко применяемую спецификацию, означает, что мы можем воспользоваться преимуществом целого арсенала уже существующих программных средств, управляющих кэшированием. Такие обратные прокси-серверы, как Squid или Varnish, могут незаметно размещаться в сети между клиентом и сервером и сохраняя кэшируемое содержимое, и устанавливая для него срок истечения годности. Эти системы предназначены для очень быстрого обслуживания огромного количества параллельных запросов и являются стандартным средством масштабирования общедоступных сайтов. Такие сети доставки контента (CDN), как имеющаяся в AWS сеть CloudFront или Akamai, могут обеспечить маршрутизацию запросов к средствам кэширования, расположенным ближе к осуществившему вызов клиенту, гарантируя тем самым, что трафик, когда он понадобится, не проделает половину кругосветного путешествия. И что еще прозаичнее, справиться за нас с этой работой смогут клиентские библиотеки и клиентские средства кэширования, относящиеся к технологии HTTP.

ETags, Expires-заголовки и `cache-control` могут перекрывать функции друг друга, и если не проявить осторожность и принять решение о применении всех этих средств, то можно столкнуться с получением весьма противоречивой информации! Получить более глубокое представление о достоинствах тех или иных средств можно, прочитав книгу *REST In Practice* (O'Reilly) или изучив раздел 13 спецификации HTTP 1.1, где описывается, как эти различные управляющие средства реализуются как на клиентской, так и на серверной стороне.

Если принято решение воспользоваться HTTP в качестве межсервисного протокола, то наиболее подходящим вариантом будет кэширование на стороне клиента и сокращение потребностей клиента в полном цикле выполнения запроса. Если будет принято решение о выборе другого протокола, то нужно понять, когда и как можно предоставить подсказки клиенту, чтобы помочь ему разобраться в том, насколько долго он может пользоваться кэшированными данными.

Кэширование, проводимое для операций записи

Кэширование чаще всего выполняется для операций чтения, однако существует ряд обстоятельств, когда имеет смысл проводить кэширование и для операций записи. Например, если используется кэширование с отложенной записью (write-behind cache), запись можно вести в локальное устройство кэширования, а чуть позже данные будут сброшены в нижестоящий источник — возможно, в канонический источник данных. Это может пригодиться при резком возрастании количества операций записи или высокой вероятности многократной записи одних и тех же данных. При использовании буфера и потенциально пакетных записей кэширование с отложенной записью может поспособствовать дальнейшей оптимизации производительности.

Если используется кэширование с отложенной записью и записи практически постоянно попадают в буфер, то при недоступности нижестоящего сервиса можно будет выстроить очередь из записей и после восстановления доступности отправить их к нему.

Кэширование в целях повышения отказоустойчивости

Кэшированием можно воспользоваться для реализации отказоустойчивости в случае сбоев. Если при использовании кэширования на стороне клиента нижестоящий сервис становится недоступным, клиент может принять решение об использовании кэшированных, но потенциально несвежих данных. Для предоставления ранее использовавшихся данных можно также воспользоваться чем-нибудь вроде обратного прокси-сервера. Для некоторых систем лучше, чтобы они были доступны даже при использовании не совсем свежих данных, чем не возвращали вообще никаких результатов, но это вы будете делать на собственный страх и риск. Разумеется, если запрашиваемых данных в кэше не окажется, то мы уже ничем не сможем помочь, но есть способы смягчения подобной ситуации.

Технология, которую мне приходилось видеть в Guardian, а потом и в других местах, заключалась в периодическом медленном перемещении существующего живого сайта в создаваемую статическую версию сайта, которую можно задействовать в случае сбоя. Хотя эта перетянутая версия не такая свежая, как кэшированное содержимое, обслуживаемое на стороне живой системы, в крайних случаях оно может гарантировать, что версия сайта все же будет выведена на экран.

Скрытие источника

Если при использовании обычного кэша запрос не станет попаданием в кэш, он отправляется к источнику для извлечения свежих данных, а вызывающий данные блокируется в ожидании результата. Обычно такой исход вполне ожидаем. Но если происходит массовое непопадание в кэш, возможно, из-за сбоя машины (или группы машин), предоставляющей кэш, к источнику будет направлено большое количество запросов.

Для сервисов, обслуживающих широко кэшируемые данные, сам источник зачастую должен масштабироваться таким образом, чтобы справиться лишь с долей общего трафика, поскольку большинство запросов обслуживается из памяти средствами кэширования, находящимися перед источником. Если вдруг возникнет критическая ситуация из-за аннулирования целой области кэша, наш источник будет истерзан до смерти.

Один из способов защиты источника в подобной ситуации заключается в первую очередь в категорическом запрете запросам направляться к источнику. Вместо этого (рис. 11.7) сам источник по мере необходимости осуществляет заполнение кэша в асинхронном режиме. Если случается непопадание в кэш, выдается событие, которое может быть подхвачено источником и которое оповещает его о необходимости перезаполнения кэша. Следовательно, если аннулирован целый кусок, мы можем воссоздать кэш в фоновом режиме. Мы можем принять решение о блокировке исходного запроса в ожидании перезаполнения области, но это может вызвать претензии к самому кэшу и возникновение неблагоприятных последствий. Скорее всего, мы отдадим предпочтение сохранению стабильности системы, отклонив исходный запрос и сделав это без промедления.

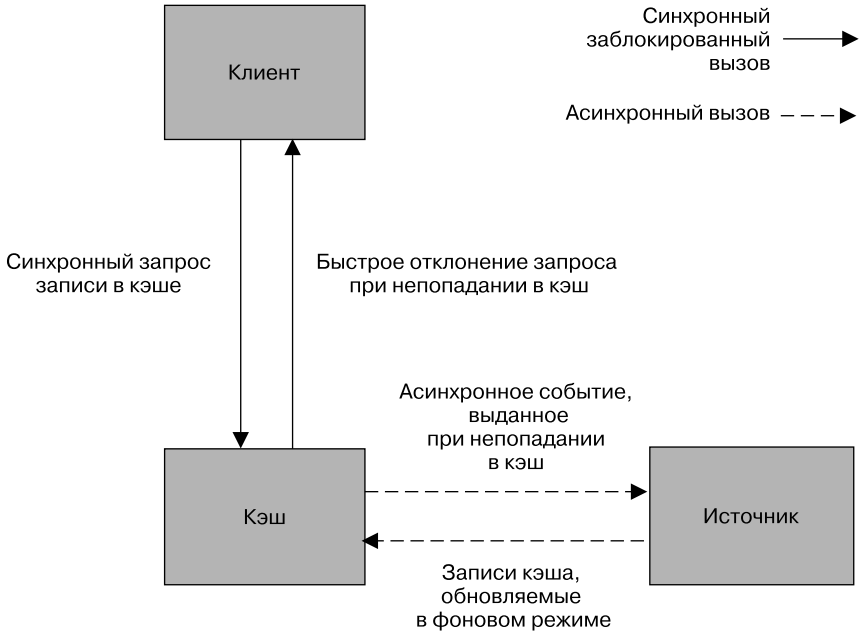


Рис. 11.7. Скрытие источника от клиента и заполнение кэша в асинхронном режиме

В некоторых ситуациях такой подход, возможно, не имеет смысла, но он может стать способом сохранения работоспособности всей системы в случае отказа ее частей. Быстро отклоняя запросы, не занимая ресурсы или не увеличивая время задержки, мы избегаем превращения сбоя в кэше в распространяющийся каскадный сбой и получаем шанс восстановить нормальный режим работы.

Не нужно ничего усложнять

Остерегайтесь излишне широкого применения кэширования! Чем больше средств кэширования между вашей системой и источником свежих данных, тем большим может оказаться объем устаревших данных и тем труднее может стать определение степени свежести тех данных, которые в итоге увидит клиент. С применением архитектуры микросервисов, где в цепочку вызовов вовлечены сразу несколько микросервисов, эта проблема может только усугубиться. Повторю еще раз: чем больше объем применения кэширования, тем труднее будет получить доступ к сведениям о свежести любой части данных. Поэтому, если вы считаете кэширование неплохой затеей, не нужно ничего усложнять, остановитесь на чем-нибудь одном и хорошенько подумайте, прежде чем что-нибудь к этому добавлять!

Отравление кэша: предостережение

При кэшировании мы думаем, что при его неправильном применении самое худшее, что может быть, заключается в кратковременном использовании устаревших данных. А что, если вам в итоге придется всегда пользоваться только устаревшими данными? Ранее я уже упоминал о проекте, над которым работал, где мы использовали приложение со strangler-шаблоном, чтобы легче было перехватывать вызовы к нескольким устаревшим системам с прицелом на постепенный отказ от их применения. Наше приложение вполне эффективно работало в качестве прокси-сервера. Трафик направлялся через приложение к устаревшим приложениям. На обратном пути выполнялись несколько служебных действий. Например, мы убеждались в том, что к результатам, полученным от устаревших приложений, применялись надлежащие HTTP-кэш-заголовки.

Однажды, почти сразу после обычного планового выпуска, начали происходить весьма странные вещи. Была допущена ошибка, при которой к небольшому поднабору страниц в коде вставки кэш-заголовка применялось логическое условие, в результате чего заголовок вообще не изменялся. К сожалению, незадолго до этого изменили и само нижестоящее приложение и в него был вставлен HTTP-заголовок постоянной свежести `Expires: Never`. Ранее это не производило никакого эффекта, поскольку заголовок переписывался. А теперь никакой перезаписи не происходило.

Для кэширования HTTP-трафика в приложении активно использовалось средство Squid, и мы заметили появившуюся проблему довольно быстро, поскольку видели, что средство Squid стало обходить больше запросов, направляющихся к серверам приложения. Мы внесли исправление в код кэш-заголовков и выдали новый выпуск, а также вручную вычистили соответствующую область Squid-кэша. Но этого оказалось недостаточно.

Как уже упоминалось, кэширование может проводиться в нескольких местах. Когда дело касается обслуживания контента для пользователей общедоступного веб-приложения, между вами и клиентом может быть несколько средств кэширования. Перед сайтом может быть не только что-нибудь вроде CDN-сети — кэширование может использоваться также некоторыми поставщиками интернет-услуг. Можете ли вы контролировать такие средства кэширования? И даже если можете,

остается еще одно средство кэширования, над которым у вас нет практически никакого контроля, — это кэш в браузере пользователя.

Страницы со свойством постоянной свежести `Expires: Never` застревают в средствах кэширования многих ваших пользователей и никогда не будут аннулированы, пока кэш не будет заполнен или пользователь не очистит его вручную. Понятно, что мы не можем инициировать ни то, ни другое событие; единственное, что можно сделать, — изменить URL-адреса таких страниц, чтобы инициировать их повторное извлечение.

Конечно же, кэширование может быть весьма эффективным, но при этом нужно представлять себе весь путь подвергаемых кэшированию данных от источника до пункта назначения, чтобы по-настоящему оценить все сложности и все места, где что-то может пойти не так.

Автоматическое масштабирование

Если вам посчастливилось получить полностью автоматизированное выделение виртуальных хостов и вы можете полностью автоматизировать развертывание экземпляров микросервисов, значит, у вас есть строительные блоки, позволяющие автоматически масштабировать микросервисы.

Например, у вас также может применяться масштабирование, запускаемое при известных тенденциях. Вам может быть известно, что пиковая нагрузка на систему наблюдается между 9:00 и 17:00, поэтому вы подключаете дополнительные экземпляры в 8:45 и выключаете их в 17:15. Если вы пользуетесь такими средствами, как AWS, в котором имеется очень хорошая встроенная поддержка автоматического масштабирования, выключение ненужных экземпляров поможет сэкономить деньги. Чтобы узнать, как изменяется нагрузка в течение дня и недели, понадобятся соответствующие данные. У некоторых разновидностей бизнеса имеются также вполне очевидные сезонные циклы, поэтому, для того чтобы сделать правильные выводы относительно объема вызовов, вам нужны ретроспективные данные.

В то же время можно быстро реагировать, подключая дополнительные экземпляры при обнаружении повышения нагрузки или сбоя экземпляра и удаляя экземпляры, когда надобность в них отпадет. Основным фактором здесь является знание того, насколько быстро вы сможете выполнить масштабирование, заметив восходящую тенденцию. Если вы знаете, что для обнаружения тенденции понадобится всего лишь пара минут, а масштабирование может занять минимум десять минут, значит, для преодоления этого разрыва нужно обзавестись дополнительными мощностями. В таком случае возникает необходимость в применении хорошего набора нагрузочных тестов. Ими можно воспользоваться для тестирования правил автоматического масштабирования. Если тесты, способные воспроизвести различные нагрузки для запуска масштабирования, отсутствуют, то действенность выбранных правил останется проверять только в производственном режиме. И негативные последствия неправильного выбора не будут слишком велики!

Хорошим примером бизнеса, для которого может потребоваться сочетание масштабирования на основе прогноза и на основе реагирования на изменение нагрузки, может послужить сайт. Наблюдая за последним новостным сайтом, над которым мне пришлось работать, мы заметили четкие дневные тенденции с ростом

нагрузки с утра до обеда и последующим ее спадом. Эта схема повторялась изо дня в день, а в выходные дни трафик был менее выраженным. Это позволяет выявить весьма четкую тенденцию, которая может управлять упреждающим масштабированием ресурсов путем их добавления (и высвобождения). В то же время какое-нибудь интересное событие может вызвать неожиданный всплеск нагрузки, требующий кратковременного более объемного выделения ресурсов.

Я считаю, что автоматическое масштабирование используется главным образом для того, чтобы справиться со сбоями экземпляров, а не для того, чтобы оперативно реагировать на условия нагрузки. В AWS вы можете определять правила вроде «в этой группе должно быть не менее пяти экземпляров», чтобы при сбое одного из них автоматически запускался новый экземпляр. Я видел, как такой подход приводил к веселой игре в «ладушки», когда кто-то забывал выключить правило, а затем пытался уменьшить количество экземпляров с целью проведения обслуживания и наблюдал за тем, как они все продолжали запускаться!

Полезны оба вида масштабирования, как на основе прогноза, так и на основе реагирования на изменение нагрузки, и оба они могут существенно повысить экономическую эффективность при использовании платформы, позволяющей платить только за использованные вычислительные ресурсы. Но они также требуют тщательного наблюдения за доступными вам данными. Я бы посоветовал использовать автоматическое масштабирование в первую очередь при возникновении сбоев в ходе сбора данных. Если потребуется приступать к масштабированию при изменении нагрузки, нужно проявить особую осторожность и не допустить слишком быстрого возврата мощностей. В большинстве случаев лучше иметь больше вычислительных мощностей, чем нужно, вместо того чтобы испытывать их острый дефицит!

Теорема CAP

Хотелось бы получить абсолютно все, но, к сожалению, мы знаем, что это невозможно. А что касается распределенных систем, подобных тем, что создаются с использованием архитектур микросервисов, есть даже математические доказательства того, что это невозможно. Может, вы уже слышали про теорему CAP, особенно при обсуждении достоинств различных типов хранилищ данных. Ее суть заключается в том, что в распределенных системах есть три компромиссных по отношению друг к другу свойства: *согласованность*, *доступность* и *терпимость к разделению*[1]. В частности, в теореме говорится, что при отказе удается сохранить только два из них.

Согласованность является характеристикой системы, означающей, что при обращении к нескольким узлам будет получен один и тот же ответ. Доступность означает, что на каждый запрос будет получен ответ. Терпимость к разделению является способностью системы справляться с тем фактом, что установить связь между ее частями порой становится невозможно.

После того как Эрик Брюер (Eric Brewer) опубликовал исходную гипотезу, идея получила математическое доказательство. Я не собираюсь в него углубляться не только потому, что это не относится к теме книги, но и потому, что не могу гаран-

[1] CAP — по первым буквам англоязычных названий этих свойств. — *Примеч. пер.*

тировать, что сделаю это правильно. Воспользуемся лучше несколькими практическими примерами, которые помогут разобраться в том, что за всем этим стоит, и в том, что теорема CAP является выжимкой из весьма логичного набора рассуждений.

Мы уже говорили о некоторых простых технологиях масштабирования баз данных. Возьмем одну из них, чтобы исследовать идеи, положенные в основу теоремы CAP. Представим, что сервис инвентаризации развернут на базе двух отдельных дата-центров (рис. 11.8). В каждом дата-центре экземпляры сервиса поддерживает база данных, и эти две базы данных обмениваются данными, стараясь синхронизировать их между собой. Операции чтения и записи осуществляются через локальный узел базы данных, а для синхронизации данных между узлами применяется репликация.

Теперь подумаем о том, что случится, когда что-нибудь откажет. Представим, что прекратило работу что-либо настолько простое, как сетевая связь между двумя дата-центрами. С этого момента синхронизация даст сбой. Записи, осуществляемые в основную базу данных в дата-центре DC1, не будут дублироваться на дата-центр DC2, и наоборот. Большинство баз данных, поддерживающих такие настройки, поддерживают также какую-либо разновидность технологии выстраивания очередей, чтобы впоследствии обеспечить возможность восстановления из этой ситуации. Но что произойдет до такого восстановления?

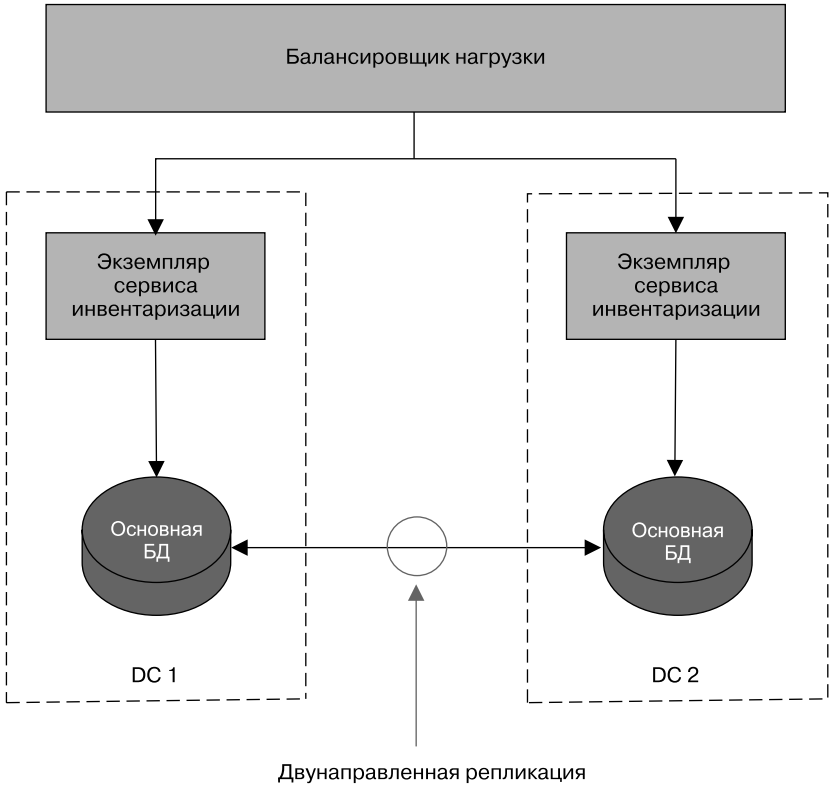


Рис. 11.8. Использование репликации двух основных баз для распределения данных между двумя узлами баз данных

Принесение в жертву согласованности

Предположим, что сервис инвентаризации нами полностью не отключен. Теперь при внесении изменений в данные в DC1 база данных в DC2 их не видит. Это означает, что любой запрос, обращенный к узлу инвентаризации в DC2, увидит потенциально устаревшие данные. Иными словами, система все еще *доступна* на обоих узлах, способных обслуживать запросы, и мы сохранили работоспособность системы, несмотря на *разделение*, но утратили *согласованность*. Зачастую это называется AP-системой. Мы не можем сохранить все три свойства.

Если в период данного разделения будет продолжено получение записей, значит, мы смирились с тем фактом, что через некоторое время записи будут рассинхронизированы. Чем дольше продлится разделение, тем труднее будет восстанавливать синхронизацию.

Реальность такова, что даже при отсутствии сетевого сбоя между узлами баз данных репликация данных не проводится мгновенно. Как уже говорилось, системы, которые готовы поступиться согласованностью для сохранения терпимости к разделению и доступности, называются *согласованными по прошествии некоторого времени*. Иначе говоря, ожидается, что в некоторый момент в будущем обновленные данные станут видны всем узлам, но это не произойдет немедленно, поэтому придется смириться с вероятностью того, что пользователи увидят устаревшие данные.

Принесение в жертву доступности

А что, если потребуется сохранить согласованность и отказаться вместо нее от чего-то другого? Итак, для сохранения согласованности каждый узел базы данных должен знать, что у него имеется такая же копия данных, как и у других узлов баз данных. Теперь при разделении, если узлы баз данных не могут связываться друг с другом, они не могут выполнять координацию для обеспечения согласованности. Мы не в состоянии гарантировать согласованность, поэтому единственным вариантом остается отказ от ответа на запрос. Иными словами, мы приносим в жертву доступность. Система согласована и терпима к разделению, или же можно сказать, что она приняла форму CP. В этом режиме сервису придется выработать меры снижения уровня функциональности до тех пор, пока не будет преодолено разделение и узлы баз данных не восстановят синхронизированность.

Обеспечить согласованность нескольких узлов довольно трудно. В распределенных системах сложно найти задачу труднее этой (если таковая вообще сможет найтись). Давайте немного порассуждаем на эту тему. Представим, что нужно прочитать запись из локального узла базы данных. Как узнать, что эта запись не устарела? Нужно обратиться с вопросом к другому узлу. Но мне также нужно попросить узел базы данных не разрешать обновление этой записи до тех пор, пока чтение не будет завершено, иными словами, для обеспечения согласованности нужно инициировать транзакционное чтение между несколькими узлами баз данных. Но люди, как правило, не связываются с транзакционным чтением, не так ли? Оно слишком медленное. Они запрашивают блокировки. Чтение может заблокировать всю систему. Для выполнения этой задачи всем согласованным системам требуется определенный уровень блокировки.

Как уже говорилось, для распределенных систем сбои не должны быть неожиданностью. Рассмотрим транзакционное чтение в наборе согласованных узлов. Я прошу удаленный узел заблокировать заданную запись, как только чтение будет инициировано. Завершаю чтение и прошу удаленный узел снять его блокировку. Но теперь я не могу с ним общаться. Что же произошло? Правильное применение блокировок вызывает серьезные затруднения даже в системе с единственным процессом, а правильно реализовать их в распределенных системах еще сложнее.

Помните, в главе 5 мы говорили о распределенных транзакциях? Основная причина затруднений при их реализации заключалась в проблеме обеспечения согласованности между несколькими узлами.

Обеспечить согласованность между несколькими узлами настолько трудно, что я буду *весьма настоятельно* рекомендовать при возникновении насущной потребности в этом не пытаться заниматься изобретательством. Лучше подобрать такое хранилище данных или такую службу блокировки, которые предлагают нужные характеристики. К примеру, такое средство, как Consul, которое вскоре будет рассмотрено подробнее, реализует совершенно согласованное хранилище пар «ключ — значение», разработанное для совместного использования конфигурации несколькими узлами. За лозунгом «Друзья не должны позволять своим друзьям писать собственные системы криптографии» должен следовать лозунг «Друзья не должны позволять своим друзьям создавать собственные распределенные согласованные хранилища данных». Если вы полагаете, что вам нужно написать собственное хранилище данных, обладающее свойствами CP, прочитайте сначала все статьи по данной теме, затем получите степень кандидата наук и потом подумайте, стоит ли тратить несколько лет на получение неработающего хранилища. А в это время я воспользуюсь чем-нибудь уже готовым или, что более вероятно, *очень постараюсь* создать вместо этого системы, обладающие свойствами AP и согласованностью, возникающей по прошествии некоторого времени.

А как насчет принесения в жертву терпимости к разделению?

Нам ведь нужно понять, какие два свойства из трех выбрать, не так ли? Мы уже получили приобретающую согласованность по прошествии некоторого времени AP-систему. У нас есть согласованная, но трудная в создании и масштабировании CP-система. Почему не создать еще и CA-систему? Итак, как же нам пожертвовать терпимостью к разделению? Если у системы отсутствует терпимость к разделению, она не сможет запускаться по сети. Иными словами, ей нужно быть единым процессом, выполняемым локально. CA-систем среди распределенных систем просто не бывает.

AP или CP?

Каков должен быть правильный выбор: AP или CP? В реальности нужно решать, *что от чего зависит*. При создании какой-нибудь системы всегда приходится идти на компромиссы. Нам известно, что AP-системы проще поддаются масштабированию и их легче создавать, и мы знаем, что CP-система потребует больших

трудозатрат из-за сложностей при поддержке распределенной согласованности. Но мы можем не понимать влияния этих компромиссов на бизнес. Подойдет ли для системы инвентаризации пятиминутная задержка с обновлением записи? Если ответ положительный, то выбор будет за AP-системой. А как насчет сведений о балансе клиента банка? Могут ли они быть устаревшими? Без знания контекста, в котором используется операция, мы не можем решить, как правильно поступить. Осведомленность о существовании теоремы CAP просто помогает понять, что такие компромиссы существуют, и разобраться в том, какие вопросы следует задавать.

Это не все или ничего

Не обязательно, чтобы вся наша система была либо AP, либо CP. Каталог может быть AP, поскольку в случае с ним нас не особо волнует устаревшая запись. А вот насчет сервиса инвентаризации может быть принято решение о его принадлежности к CP, поскольку мы не собираемся продавать клиенту отсутствующий товар, а затем приносить ему извинения.

Но даже отдельные сервисы не обязательно должны быть либо CP, либо AP.

Подумаем о сервисе остатка бонусных баллов, в котором сохраняются записи о количестве полученных клиентами бонусных баллов. Можно решить, что устаревание сведений об остатке этих баллов, демонстрируемых клиенту, нас особо не волнует, но что делать, когда речь пойдет об обновлении остатка и обязательном поддержании согласованности, чтобы убедиться в том, что клиенты не воспользовались бóльшим количеством баллов, чем им было доступно? К какому из типов отнести этот микросервис: CP, AP или же сразу к обоим? Вообще-то мы сделали следующее: применили компромиссы теоремы CAP к отдельным возможностям сервиса.

Еще одна сложность заключается в том, что ни согласованность, ни доступность не должны рассматриваться как либо все, либо ничего. Многие системы допускают куда более тонкие компромиссы. Например, применяя такое средство, как Cassandra, я могу идти на различные компромиссы для отдельных вызовов. Поэтому, если мне требуется строгое соблюдение согласованности, я могу выполнить чтение, ставящее блокировку до тех пор, пока не ответят все реплики, или пока не ответит определенный кворум реплик, или даже пока не ответит отдельный узел. Понятно, что, если в ожидании обратного отчета от всех реплик одна из них окажется недоступна, блокировка будет сохраняться довольно долго. Но если меня устроит простой кворум узлов, приславших отчет, для меня может быть приемлемо частичное отсутствие согласованности, которое поможет стать менее уязвимым к недоступности отдельной реплики.

Вам будут часто попадаться сообщения о людях, которым удалось *обойти* теорему CAP. Это неправда. Все, что им удается сделать, заключается в создании системы, где некоторые возможности относятся к категории CP, а некоторые — к AP. Математическое доказательство, подкрепляющее теорему CAP, остается незыблемым. Несмотря на неоднократные мои попытки обойти математику в школе, я понял, что сделать это невозможно.

И реальный мир

Многое, о чем мы говорим, относится к миру электроники — битам и байтам, сохраненным в памяти. Мы говорим о согласованности в какой-то манере, похожей на детские рассуждения. Мы представляем себе, что в пределах видимости созданной нами системы мы можем остановить мир и считаем, что во всем этом есть смысл. И еще многое из того, что мы создаем, — это просто отражение реального мира, и мы не в состоянии этим управлять, не так ли?

Вернемся к системе инвентаризации. Она отображается на физические товарные позиции реального мира. Мы в своей системе подсчитываем количество имеющихся альбомов. На начало дня было 100 копий альбома Give Blood группы The Brakes. Одну мы продали. Теперь у нас 99 копий. Все просто, не правда ли? Но что получится, если при отправке заказа кто-нибудь уронит копию на пол и случайно ее раздавит? Что произойдет? Наша система говорит «99», но на полке 98 копий.

А что, если мы создали нашу систему инвентаризации с прицелом на AP-свойства и впоследствии время от времени вынуждены выходить на контакт с пользователем и сообщать ему, что один из его товаров отсутствует на складе? Станет ли это самым худшим, что может случиться в этом мире? Несомненно, гораздо проще будет создать систему, выполнить ее масштабирование и убедиться в том, что все сделано правильно.

Мы должны признать, что независимо от степени возможной согласованности наших систем и от них самих разобраться абсолютно во всем, что может произойти, невозможно, особенно если мы ведем учет происходящего в реальном мире. Это одна из основных причин, по которой во многих ситуациях AP-системы в конечном счете берут верх. Кроме того что CP-системы сложны для построения, они все равно не в состоянии решить все наши проблемы.

Обнаружение сервисов

Когда количество микросервисов, находящихся в вашем распоряжении, станет солидным, вы непременно захотите получить сведения о том, где располагается каждый из них. Возможно, вам потребуется узнать, какой из них работает в той или иной среде, чтобы понять, что нужно отслеживать. Может, просто захочется узнать, где находится сервис счетов, чтобы использующие его микросервисы знали, где его найти. Или, может быть, вам просто потребуется упростить оповещение разработчиков — сотрудников организации, чтобы они знали, какие API-интерфейсы доступны, и не изобретали колесо заново. В широком смысле все эти сценарии можно назвать *обнаружением сервисов*. И как всегда бывает с микросервисами, в нашем распоряжении немало вариантов, которые можно взять на вооружение.

Все решения, которые будут рассматриваться, действуют в двух аспектах. Во-первых, они предоставляют экземпляру некий механизм для регистрации и заявления «Я здесь!». Во-вторых, они дают способ, позволяющий найти сервис после того, как он зарегистрируется. Но обнаружение сервисов дается труднее, когда рассматривается среда, в которой постоянно ведутся ликвидация и развертывание

новых экземпляров сервисов. В идеале хотелось бы, чтобы с этой задачей справлялось любое выбранное нами решение.

Рассмотрим некоторые наиболее распространенные средства поставки сервиса и оценим доступные варианты.

DNS

Лучше начать с простого. DNS позволяет нам связать имя с IP-адресом одной или нескольких машин. Мы можем, к примеру, решить, что сервис счетов всегда можно будет найти по адресу accounts.musiccorp.com. Затем у нас может быть запись, указывающая на IP-адрес хоста, на котором работает этот сервис, или, возможно, этот адрес при его разрешении будет указывать на балансировщик нагрузки, распределяющий нагрузку между несколькими экземплярами. Это означает, что нам в рамках развертывания сервиса придется заниматься обновлением этих входов.

При работе с экземплярами сервиса в других средах мне приходилось наблюдать эффективную работу шаблона домена на основе соглашений. Например, у нас может быть шаблон, определяемый как *<имя_сервиса>*-*<среда>*.musiccorp.com, дающий входы типа accounts-uat.musiccorp.com или accounts-dev.musiccorp.com.

Более совершенным способом управления различными средами является использование для разных сред серверов с разными доменными именами. Это позволяет предположить, что accounts.musiccorp.com — это место, где я всегда смогу найти сервис счетов, но данный адрес может разрешаться указаниями на различные хосты в зависимости от того, где я выполняю поиск. Если у вас уже есть свои среды, размещенные в разных сегментах сети, и вам удобно управлять собственными DNS-серверами и записями, это может стать отличным решением, но если вы не извлекаете из такого подхода никаких других преимуществ, можно счесть его слишком трудоемким.

У DNS множество преимуществ, и одно из основных заключается в том, что этот стандарт настолько хорошо продуман и имеет настолько богатый положительный опыт применения, что его поддерживают практически все технологические стеки. К сожалению, когда внутри организации существует несколько сервисов для управления DNS, для среды, в которой мы работаем с высокодоступными хостами, предназначены только некоторые из них, что существенно затрудняет обновление DNS-записей. В компании Amazon неплохо справляется с этой работой сервис Route53, но мне еще не попадался достаточно хороший, самостоятельно занимающий хост вариант, хотя (в чем мы вскоре убедимся) в этом вопросе нам сможет помочь такое средство, как Consul. Кроме трудностей с обновлением DNS-записей, ряд проблем может создать и сама спецификация DNS.

DNS-записи для доменных имен имеют указание *времени жизни информации* (TTL). Это время означает, как долго клиент может считать запись свежей. Когда мы хотим изменить хост, на который ссылается доменное имя, мы обновляем запись, но нужно учесть то обстоятельство, что у клиента будет храниться старый IP-адрес, *по крайней мере* пока не истечет TTL. DNS-записи могут кэшироваться во многих местах (даже JVM-машина будет кэшировать DNS-записи, пока вы не заставите ее не делать этого), и чем больше мест, в которых запись кэширована, тем более устаревшей она может быть.

Одним из способов обхода этой проблемы является содержание записи доменного имени для вашего сервиса, указывающего на балансировщик нагрузки, который (рис. 11.9), в свою очередь, указывает на экземпляры сервиса. При развертывании нового сервиса можно удалить старый из записи балансировщика нагрузки и вместо него добавить новый. Некоторые специалисты используют DNS в режиме круговой конкуренции, где сами DNS-записи ссылаются на группу машин. Эта технология считается крайне проблематичной, поскольку клиент скрыт от используемого хоста и потому не может справиться с остановкой маршрутизации трафика на один из хостов в случае его отказа.

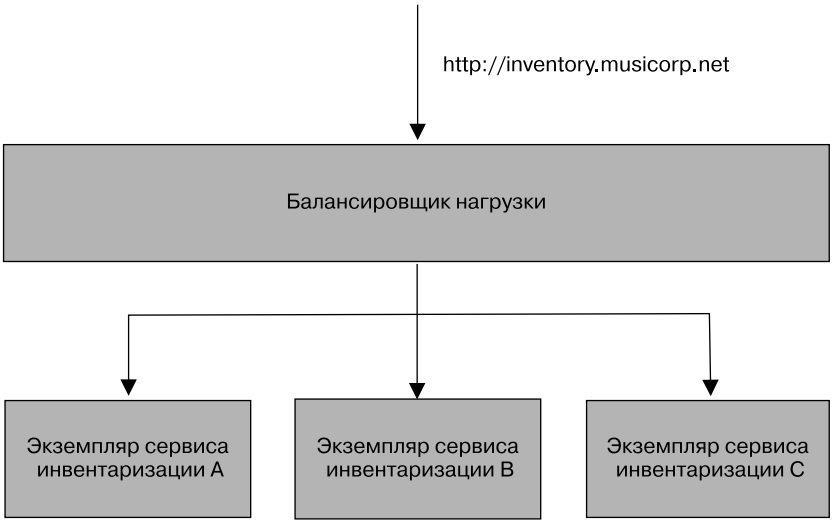


Рис. 11.9. Решение проблемы устаревших DNS-записей путем использования DNS для направления на балансировщик нагрузки

Как уже упоминалось, технология DNS хорошо продумана и пользуется весьма широкой поддержкой. Но у нее все же имеется один или два недостатка. Прежде чем выбрать что-нибудь более сложное, я все же посоветовал бы вам разобраться в том, насколько эта технология сможет подойти для решения именно ваших проблем. Когда у вас имеются только отдельные узлы, использования DNS, ссылающейся непосредственно на хосты, может оказаться вполне достаточно. Но в тех ситуациях, где нужно иметь более одного экземпляра на хост, лучше подойдет использование DNS-записей, разрешенных направлением трафика к балансировщику нагрузок, который может справиться с вводом в строй и с выводом из строя отдельных хостов.

Динамическая регистрация сервисов

Недостатки DNS как способа нахождения узлов в высокодинамичной среде привели к появлению множества альтернативных систем, в большинстве из которых используется самостоятельная регистрация сервисов с применением центрального

реестра, который, в свою очередь, предлагает возможность впоследствии выполнять поиск этих сервисов. Зачастую такие системы способны на большее, чем простое предоставление возможности регистрации сервиса и его обнаружения, что можно рассматривать и как полезные свойства, и как излишества. Эта область перенасыщена различными возможностями, поэтому, чтобы дать представление о том, что вам доступно, мы просто рассмотрим несколько вариантов.

Zookeeper

Средство Zookeeper изначально разрабатывалось как часть проекта Hadoop. Оно используется в невероятно большом количестве случаев, в том числе в качестве средства управления конфигурациями, синхронизации данных между сервисами, выбора лидера, выстраивания очереди сообщений и (что полезно для нас) в качестве службы имен.

Подобно многим похожим типам систем, для предоставления различных гарантий Zookeeper полагается на работу сразу на нескольких узлах кластера. Это означает, что вам следует ожидать запуска как минимум трех узлов Zookeeper. Основная доля интеллекта, заложенного в Zookeeper, приходится на обеспечение того, чтобы данные были безопасно реплицированы между этими узлами и чтобы при сбоях узлов сохранялась согласованность.

В своей основе Zookeeper для сохранения информации предоставляет иерархическое пространство имен. Клиенты могут вставлять в эту иерархию новые узлы, вносить в них изменения или запрашивать их. Более того, они могут добавлять к узлам наблюдателей, чтобы получать оповещение о внесении в них изменений. Это означает, что в этой структуре мы можем сохранять информацию о том, где находятся наши сервисы, и клиент будет оповещен о внесении изменений. Zookeeper зачастую используется в качестве главного хранилища конфигурации, поэтому вы также можете сохранять в нем конфигурацию, характерную для сервиса, что позволит вам выполнять такие задачи, как динамическое изменение уровней журнальной регистрации или выключение свойств работающей системы. Лично я стараюсь избегать использования таких систем, как Zookeeper, в качестве источника конфигурации, поскольку считаю, что они могут усложнить поиск причин конкретного поведения того или иного сервиса.

Само средство Zookeeper может делать универсальные предложения, и именно поэтому столь широк спектр его применения. Его можно считать просто реплицированным деревом информации с возможностью оповещения о вносимых изменениях. Это означает, что обычно для подстраивания под конкретный случай применения приходится создавать различные надстройки. К счастью, для этого существуют клиентские библиотеки для большинства языков программирования.

По большому счету, на сегодняшний день Zookeeper может считаться *устаревшим* и не предоставляет нам такого количества готовых функциональных возможностей для обнаружения сервисов, как более новые альтернативные средства. Тем не менее это давно испытанное и широко используемое средство. Правильно применить основные алгоритмы, реализуемые средством Zookeeper, весьма непросто.

К примеру, я знаю одного поставщика баз данных, который использовал Zookeeper только для выбора лидера с целью обеспечения правильного повышения статуса до основного узла при возникновении сбоя. Клиент чувствовал, что Zookeeper был слишком «тяжелым» средством, и потратил много времени на сглаживание шероховатостей в собственной реализации PAXOS-алгоритма, чтобы найти замену тому, что делало средство Zookeeper. Приходится довольно часто слышать, что не нужно создавать собственные криптографические библиотеки. Я расширю это утверждение, сказав, что не следует также создавать собственные распределенные системы координирования. Ведь сколько раз уже говорилось, что нужно использовать то, что уже есть и нормально работает.

Consul

Как и Zookeeper, средство Consul поддерживает и управление конфигурациями, и обнаружение сервисов. Но оно пошло дальше Zookeeper в части предоставления более широкой поддержки ключевых сценариев использования. Например, для обнаружения сервисов в нем показывается HTTP-интерфейс, и одной из убийственных особенностей Consul является фактическое предоставление готового DNS-сервера, а именно это средство может обслуживать SRV-записи, которые дают вам для заданного имени как IP-адрес, так и порт. Следовательно, если часть вашей системы уже использует DNS и может поддерживать SRV-записи, можете просто присоединиться к Consul и приступить к его использованию, не внося никаких изменений в уже существующую систему.

В Consul имеются и другие встроенные возможности, которые могут вам пригодиться, например проверка степени работоспособности узлов. Следовательно, Consul вполне может перекрыть возможности, которые предоставляют другие, специально предназначенные для этого инструменты мониторинга, хотя вы, скорее всего, воспользуетесь Consul в качестве источника этой информации, а затем разместите ее на более сложной инструментальной панели или в системе выдачи предупреждений. Но если в некоторых сценариях применения присущие Consul отказоустойчивость и сконцентрированность на обслуживании систем, интенсивно применяющие узлы краткосрочного пользования, приведут к тому, что этим средством в итоге будут заменены такие средства, как Nagios и Sensu, это меня не удивит.

В Consul для всего, от регистрации сервиса и отправки запросов в хранилище пар «ключ — значение» до вставки проверок степени работоспособности, используется RESTful HTTP-интерфейс. Это существенно упрощает интеграцию с различными технологическими стеками. Одной из особенностей, которая мне нравится в Consul, является то, что поддерживающая это средство команда выделила основную часть управления кластером. Serf, надстройкой для которого является Consul, занимается обнаружением узлов в кластере, управляет восстановлением после сбоев и выполняет оповещение. Затем Consul добавляет к этому обнаружение сервисов и управление конфигурациями. Такое разделение ответственности мне очень импонирует, что не должно стать для вас сюрпризом, учитывая темы, рассматриваемые на страницах этой книги!

Consul является совершенно новым средством, и из-за сложности используемых в нем алгоритмов я, как всегда, испытываю некоторые сомнения, рекомендуя его для столь важного задания. Тем не менее Hashicorp — команда, занимающаяся его поддержкой, несомненно, имеет солидный послужной список в создании весьма полезных технологий с открытым кодом (в виде как Packer, так и Vagrant). И я разговаривал с некоторыми специалистами, успешно использующими данное средство в производственном режиме. Исходя из этого, думаю, к нему стоит присмотреться.

Eureka

Разработанная компанией Netflix система с открытым кодом под названием Eureka не следует тенденциям, наблюдающимся в таких системах, как Consul и Zookeeper, и не пытается быть универсальным хранилищем конфигураций. Она имеет весьма узкую специализацию.

Eureka предоставляет основные возможности обеспечения сбалансированности нагрузки тем, что может поддерживать круговой поиск экземпляров сервисов. Эта система предоставляет конечную точку на REST-основе, поэтому вы можете создавать своих клиентов или использовать ее собственных Java-клиентов. Java-клиент предоставляет такие дополнительные возможности, как проверка состояния работоспособности экземпляров. Разумеется, при отказе от собственного клиента Eureka и переходе непосредственно к конечной точке на REST-основе вам придется заниматься всем этим самостоятельно.

За счет того что клиенты имеют дело с обнаружением сервисов напрямую, нам удается избежать использования отдельного процесса. Но это требует, чтобы сервис обнаружения был реализован каждым клиентом. В Netflix, занимающейся стандартизацией на JVM-машинах, это достигается тем, что Eureka используется всеми клиентами. Если вы являетесь более крупным полиглотом в области вычислительных сред, то для вас это может оказаться гораздо более сложной задачей.

Прокатка собственной системы

Один из подходов, которым я пользовался сам и использование которого видел в других местах, заключался в прокатке собственной системы. В одном из проектов мы интенсивно использовали инфраструктуру AWS, в которой предлагалось воспользоваться возможностью добавления к экземплярам меток. При запуске экземпляров сервиса я хотел применить метки, чтобы помочь определить, каким был этот экземпляр и для чего он использовался. Это позволило бы связать расширенные метаданные с заданным хостом, например:

- service = accounts *(сервис = accounts, то есть счета)*;
- environment = production *(среда = производство)*;
- version = 154 *(версия = 154)*.

Я мог использовать API-интерфейсы инфраструктуры AWS для запроса всех экземпляров, связанных с заданной учетной записью AWS, чтобы найти машины,

которые меня интересовали. В данном случае сохранением метаданных, связанных с каждым экземпляром, и предоставлением нам возможности их запроса занималась сама инфраструктура AWS. Затем для взаимодействия с этими экземплярами я создал инструментальные средства командной строки, чем существенно облегчил создание инструментальных панелей для отслеживания состояния, особенно при использовании идеи, заключавшейся в том, чтобы заставить каждый экземпляр сервиса показывать подробности проверки степени работоспособности.

Когда я делал это в последний раз, мы не заходили так далеко, чтобы заставлять сервисы использовать API-интерфейсы AWS для поиска их зависимостей, но не вижу причин, по которым вы не смогли бы это сделать. Разумеется, если вам нужно оповестить вышестоящие сервисы об изменения местонахождения нижестоящих сервисов, то придется делать это самостоятельно.

Не забывайте про людей!

Системы, которые рассматривались до сих пор, облегчали экземплярам сервисов проведение саморегистрации и выискивание других сервисов, с которыми нужно общаться. Но людям также иногда нужна эта информация. Какую бы систему вы ни выбирали, убедитесь в том, что у вас есть доступные инструментальные средства, позволяющие создавать отчеты и информационные панели в верхней части реестров, чтобы создать экспозицию для людей, а не только для компьютеров.

Документирующие сервисы

Разбиением систем на узкоспециализированные микросервисы мы надеемся показать множество стыков в виде API-интерфейсов, которые люди могут использовать для создания многих, надеюсь, замечательных вещей. Если обнаружение проведено правильно, мы знаем, где что есть. Но как узнать, чем все это занимается или как всем этим пользоваться? Один из очевидных вариантов заключается в применении документации, описывающей API-интерфейсы. Разумеется, документация может часто устаревать. В идеале хотелось бы обеспечить постоянное соответствие документации состоянию API-интерфейса микросервиса и облегчить просмотр этой документации, когда мы знаем, где находится конечная точка сервиса. Воплотить это в реальность пытаются две различные технологии, Swagger и HAL, и обе они заслуживают рассмотрения.

Swagger

Swagger позволяет описать API-интерфейс, чтобы можно было создать весьма функциональный пользовательский веб-интерфейс, позволяющий просматривать документацию и взаимодействовать с API-интерфейсом посредством браузера. Особенно подкупает возможность выполнения запросов: вы можете, к примеру, определить POST-шаблоны, разъясняя, какого сорта контент ожидается сервером.

Чтобы справиться со всем этим, Swagger нуждается в сервисе для выставления сопутствующего файла, соответствующего Swagger-формату. У Swagger имеется ряд библиотек для различных языков программирования, которые делают все это за вас. Например, для Java можно комментировать методы, соответствующие API-вызовам, а для вас будет создан соответствующий файл.

Мне нравится то впечатление, которое оставляет Swagger у конечного пользователя, но оно мало что дает для концепции дополнительных исследований на основе гиперсреды. И все же это очень хорошее средство показа документации, дающей представление о ваших сервисах.

HAL и HAL-браузер

Сам по себе язык гипертекстовых приложений (HAL) является стандартом, в котором описываются положения, касающиеся выставляемых нами элементов управления гиперсредой. Как уже говорилось в главе 4, элементы управления гиперсредой являются средствами, с помощью которых мы позволяем клиентам постепенно исследовать API-интерфейсы, чтобы применять возможности наших сервисов без задействования такой же сильной связанности, которая используется другими технологиями интеграции. Если вы решите применять стандарты гиперсреды, заложенные в HAL, то это позволит вам воспользоваться не только большим количеством клиентских библиотек для применения API-интерфейса (на момент написания данной книги в статье «Википедии», посвященной HAL, перечислялись 50 поддерживающих его библиотек для множества различных языков), но и HAL-браузером, предоставляющим способ исследования API с помощью браузера.

Как и Swagger, этот пользовательский интерфейс может применяться не только как живая документация, но и как средство, выполняющее вызовы самого сервиса. Хотя выполнение вызовов происходит не так же гладко. Если при использовании Swagger можно определять шаблоны для выполнения таких операций, как выдача POST-запросов, с HAL вы больше полагаетесь на собственные силы. В то же время присущая элементам управления гиперсредой эффективность позволяет намного продуктивнее исследовать API-интерфейс, демонстрируемый сервисом, поскольку дает возможность легко и просто следовать по ссылкам. Оказывается, браузеры великолепно справляются и с этими задачами!

В отличие от Swagger, все информация, необходимая для управления данной документацией и «песочницей», встроена в элементы управления гиперсредой. Но это обоюдоострый меч. Если вам уже приходилось использовать элементы управления гиперсредой, то больших усилий для показа HAL-браузера и предоставления клиентам возможности исследования вашего API-интерфейса не понадобится. Но если вам не приходилось пользоваться гиперсредой, то либо вы не сможете применять HAL, либо вам придется подгонять свой API-интерфейс под использование гиперсреды, что, скорее всего, станет занятием, нарушающим нормальную работу текущих потребителей.

Тот факт, что в HAL также дается описание стандартов гиперсреды с поддержкой клиентских библиотек, является дополнительным бонусом, и я полагаю, что

именно в этом кроется весомая причина, по которой среди тех, кто уже имеет опыт применения элементов управления гиперсредой, наблюдается больше сторонников использования HAL, чем сторонников применения Swagger в качестве способа документирования API-интерфейсов. Если вы используете гиперсреду, то я советую отдать предпочтение HAL, а не Swagger. Однако если вы используете гиперсреду, но не видите оснований для перехода на HAL, я настоятельно рекомендую пустить в дело Swagger.

Самостоятельно описываемая система

В ходе раннего развития сервис-ориентированной архитектуры появились такие стандарты, как Universal Description, Discovery и объединенный стандарт Integration (UDDI), помогающие людям разобраться в том, какие сервисы были запущены. Эти подходы были весьма тяжеловесными, что приводило к появлению альтернативных технологий, с помощью которых осуществлялись попытки разобраться в наших системах. Мартин Фаулер рассмотрел концепцию понятного человеку реестра, в рамках которой использовались намного более легкие подходы, предназначенные просто для того, чтобы люди могли записывать информацию о сервисах в организациях в чем-то таком же простом, как вики.

Получение описания нашей системы и характера ее поведения играет важную роль, особенно когда дело касается вопросов масштабирования. Мы уже рассмотрели несколько различных технологий, которые помогут нам обрести понимание непосредственно из нашей системы. Отслеживая состояние работоспособности нижестоящих сервисов наряду с отслеживанием идентификаторов взаимосвязанности, что помогает нам проследить цепочки вызовов, мы можем получить реальные данные в условиях взаимосвязанности сервисов. Используя такие системы обнаружения сервисов, как Consul, мы можем увидеть, где работают микросервисы. HAL позволяет нам увидеть, какие возможности размещены в данный момент в каждой отдельно взятой конечной точке, а страница проверки работоспособности и системы мониторинга позволяют узнать, каково состояние работоспособности как системы в целом, так и отдельных сервисов.

Всю эту информацию можно получить программным способом. Совокупность этих данных позволяет нам сделать понятный человеку реестр эффективнее простой вики-страницы, которая, несомненно, будет устаревать. Вместо этого мы будем применять его для задействования и отображения всей информации, выдаваемой системой. Создавая настраиваемые информационные панели, мы можем обобщить огромный массив информации, доступной для оказания помощи в осмыслении нашей экосистемы.

Во всех смыслах начинать следует с чего-нибудь настолько же простого, как статическая веб-странница или вики, на которой по крупицам собираются данные о живой системе. Но со временем нужно присматриваться к размещению все больших объемов информации. То, что к этой информации можно быстро получить доступ, является ключевым инструментом, позволяющим справиться с трудностями, возникающими из-за эксплуатации этих систем в режиме масштабирования.

Резюме

Микросервисы как подход к проектированию все еще довольно молоды, поэтому, хотя у нас уже есть вполне определенный опыт, от которого можно отталкиваться, я уверен, что в следующие несколько лет появятся более удачные схемы, позволяющие успешно справляться с их масштабированием. И тем не менее я надеюсь, что в этой главе был намечен ряд шагов, которыми можно будет воспользоваться в вашем путешествии по масштабированию микросервисов и которые станут для вас хорошим подспорьем.

Вдобавок к тому, что здесь было рассмотрено, я рекомендую прочитать замечательную книгу Майкла Нигарда (Michael Nygard) *Release It!*. В ней он делится с нами подборкой историй о сбоях системы и некоторыми схемами, помогающими успешно справиться со сбоями. Книгу действительно стоит прочитать (более того, я хотел бы заострить внимание на том, что она должна рассматриваться в качестве весьма важного пособия для тех, кто создает масштабируемые системы).

Мы уже усвоили довольно много основ и близки к завершению нашего разговора. В следующей, последней главе мы уделим внимание объединению всего изученного в единое целое и подытожим все, что нам удалось узнать при прочтении книги.

12 Коротко обо всем

В предыдущих главах было рассмотрено множество вопросов: от того, чем являются микросервисы, до способов определения их границ и от технологии интеграции до решения вопросов безопасности и мониторинга. У нас даже нашлось время на выяснение роли, отводимой в этой области архитектору. Осмысливать приходится весьма большой объем материала, и хотя сами микросервисы могут быть совсем небольшими, этого не скажешь о ширине размаха и влиянии, оказываемом их архитектурой. Поэтому в данной главе я постараюсь подвести итоги по ряду ключевых моментов, рассмотренных в книге.

Принципы микросервисов

Роль, которую могут играть принципы, мы рассматривали в главе 2. Принципы складываются из утверждений о том, как все должно делаться, и из разъяснения причин, по которым все должно делаться именно таким образом. Они помогают нам выстроить различные решения, которые приходится принимать при создании наших систем. Конечно же, вы должны определить собственные принципы, но я считаю, что есть смысл изложить мое видение того, как выглядят ключевые принципы микросервисных архитектур, обобщенный вид которых представлен на рис. 12.1. Эти принципы призваны помочь нам в создании небольших автономных сервисов, успешно выполняющих совместную работу. Все они по крайней мере единожды уже были рассмотрены в этой книге, поэтому ничего нового вы здесь не найдете, но ценность рисунка определяется тем, что в нем передана только суть.

Вы можете принять за основу сразу все эти принципы или же подстроить их под то, что имеет смысл принять именно в вашей организации. Но обратите внимание на ценность их использования в сочетании друг с другом: целостность может быть гораздо ценнее суммы частей. Итак, если вы решили отвергнуть один из них, убедитесь в том, что понимаете, что теряете.

Для каждого из этих принципов я пытался подобрать подтверждающие примеры из практики, которые были рассмотрены в данной книге. Но, как говорится, на них свет клином не сошелся: чтобы выработать принципы, можно воспользоваться собственным опытом, но те принципы, которые здесь изображены, могут для начала стать хорошим подспорьем.

Рис. 12.1. Принципы микросервисов

Модель, построенная вокруг бизнес-концепций

Опыт подсказывает, что интерфейсы, структурированные вокруг контекстов, связанных с бизнес-задачами, отличаются большей стабильностью, чем те, которые структурированы вокруг технических концепций. Моделированием областей, в которых работает наша система, мы не только пытаемся создать более стабильные интерфейсы, но и обеспечиваем повышение возможности более свободно отображать изменения в бизнес-процессах. Для определения потенциальных границ областей следует использовать *ограниченный контекст*.

Внедрение культуры автоматизации

Микросервисы усложняют системы главным образом из-за необходимости работать с огромным количеством активных компонентов. Применение культуры автоматизации является одним из основных способов справиться с этой проблемой, и в этом плане имеет смысл как можно раньше приложить максимум усилий к созданию инструментария для поддержки микросервисов. Поскольку удостовериться в работоспособности сервисов по сравнению с монолитными системами намного сложнее, важную роль в этом процессе играет *автоматизированное тестирование*. Также может помочь использование одинакового вызова командной строки для *повсеместного единообразного развертывания*, который может стать основной частью внедрения *непрерывной поставки* с целью обеспечения быстрой оценки качества работы в производственном режиме при каждой проверке.

Подумайте о возможности использования *определений среды* в качестве вспомогательного средства конкретизации отличий одной среды от другой без ущерба для возможности использования единообразного метода развертывания. Также подумайте о создании *настраиваемых образов*, позволяющих ускорить развертывание и включающих в себя создание полностью автоматизированных *неизменяемых серверов*, упрощающих понимание устройства ваших систем.

Сокрытие подробности внутренней реализации

Чтобы максимально увеличить возможность одного сервиса совершенствоваться независимо от всех других сервисов, крайне важно скрыть подробности его реализации. Содействие в этом могут оказать *ограниченные контексты*, поскольку они помогают сконцентрировать усилия на моделях, предназначенных для совместного использования и при этом обладающих скрытностью. Сервисы также должны *скрывать свои базы данных* во избежание попадания в одну из наиболее распространенных разновидностей связывания, которая может проявляться в традиционных сервис-ориентированных архитектурах и использовать для объединения данных от нескольких сервисов с целью создания отчетов *перекачку данных* или *перекачку данных о событиях*.

При любой возможности отдавайте предпочтение применению *технологически нейтральных API-интерфейсов*, чтобы получить свободу использования различных технологических стеков. Подумайте о применении технологии передачи репрезентативного состояния *REST*, оформляющей разделение подробностей внутренней и внешней реализации which. Те же самые идеи можно принять за основу и при использовании удаленных вызовов процедур (RPC).

Всесторонняя децентрализация

Для достижения максимальной автономности, допускаемой микросервисами, нужно постоянно выискивать возможности для передачи полномочий по принятию решений и управлению тем командам, которые владеют микросервисом. Этот процесс начинается с внедрения *самообслуживания* везде, где только можно, позволяя людям развертывать программные средства по мере надобности, делая как можно проще разработку и тестирование и исключая необходимость для отдельных команд выполнять эти действия.

Обеспечение того, что *команды владеют своими собственными сервисами*, является важным шагом на данном пути, возлагающим на команды ответственность за вносимые изменения и в идеале даже за принятие решения о том, когда именно выпустить такие изменения. Чтобы обеспечить возможность внесения изменений в сервисы, находящиеся во владении других команд, воспользуйтесь *семейственным открытым кодом*, но при этом помните, что для его реализации нужно будет поработать. *Подстраивайте команды под организацию*, чтобы гарантировать, что закон Конвея работает на вас и помогает вашей команде приобрести специализацию в создаваемых ею сервисах, сконцентрированных на решении конкретных бизнес-задач. Если требуется какая-либо всеобъемлющая ориентация,

попробуйте воспользоваться моделью *общего руководства*, при которой представители разных команд несут коллективную ответственность за развитие технической концепции системы.

Этот принцип применим и для архитектуры. Избегайте подходов вроде сервисных шин предприятия или оркестровых систем, которые могут привести к централизации бизнес-логики и появлению безмолвных сервисов. Чтобы гарантировать наличие соответствующей логики и данных на границах сервисов, которые помогают достигать нужного сцепления, *отдавайте предпочтение хореографическому, а не оркестровому принципу* и *безмолвным промежуточным программным средствам с интеллектуальными конечными точками*.

Независимое развертывание

Нужно всегда стремиться к тому, чтобы микросервисы могли развертываться и развертывались самостоятельно. Даже когда требуются изменения, нарушающие общий режим работы, нужно добиваться *совместного существования конечных точек разных версий*, позволяя потребителям со временем внести изменения. Тем самым мы сможем оптимизировать скорость выпуска новых свойств, а также повысить автономность работы команд, владеющих этими микросервисами, снимая с них обязанность постоянного согласования развертываний своих сервисов. Если используется интеграция на основе RPC, избегайте *при создании заглушек тесной связанности между клиентом и сервером* вроде той, что поддерживается технологией Java RMI.

Принимая модель *«по одному сервису на каждом хосте»*, вы уменьшаете число побочных эффектов, которые могут при развертывании одного сервиса влиять на другой, не связанный с ним сервис. Чтобы отделить развертывание от выпуска, сократив тем самым риск неудачного выпуска, присмотритесь к использованию технологий *сине-зеленого развертывания* или *канареечного выпуска*. Чтобы отловить изменения, нарушающие работу системы, еще до того, как они на нее смогут повлиять, воспользуйтесь *контрактами на основе запросов потребителей*.

Запомните, что возможность внесения изменений в отдельно взятый сервис и его выпуска для работы в производственном режиме без необходимости развертывания других сервисов с приостановкой работы системы должна стать нормой, а не исключением. Ваши *потребители должны сами решать, когда им нужно обновляться*, а вам следует к этому приспособиться.

Изолирование сбоев

Архитектура микросервисов может быть более устойчивой, чем монолитная система, но только если мы разбираемся в ситуации и имеем план на случай сбоев в части нашей системы. Если возможность и неизбежность сбоев вызова нижестоящего сервиса в расчет не принимаются, наша система может испытать катастрофический каскадный сбой и мы останемся с системой, еще более хрупкой, чем прежняя.

При использовании сетевых вызовов *не уподобляйте удаленные вызовы локальным*, так как из-за этого будут скрываться разные виды отказов. Поэтому убедитесь,

что при использовании клиентских библиотек не происходит чрезмерного применения удаленных вызовов.

Не нужно постоянно думать о том, что система должна быть *устойчивой к сбоям*, и при этом всегда и всюду ожидать сбоев. Обеспечьте соответствующую настройку *лимитов времени*. Разберитесь, когда и как для ограничения выхода из строя компонентов использовать *перегородки* и *предохранители*. Разберитесь в том, какое влияние окажет на клиента неправильная работа всего одной из частей системы. Узнайте, какими окажутся последствия разделения сети и будет ли правильно требовать в этой ситуации принести в жертву *доступность* или *согласованность* данных.

Всестороннее наблюдение

При оценке правильности функционирования системы мы не можем полагаться на наблюдение за поведением одного-единственного экземпляра сервиса или состоянием одной машины. Нам требуется более широкое представление о происходящем. Чтобы убедиться в правильном поведении системы, нужно использовать *семантический мониторинг* путем внедрения в вашу систему *искусственных транзакций* с целью имитации поведения реального пользователя. Следует *объединить свои журналы* и *объединить статистические данные*, чтобы при обнаружении проблемы можно было докопаться до ее источника. А когда дело дойдет до воспроизведения неприятных моментов в поведении системы или наблюдения за тем, как в вашей системе происходит взаимодействие компонентов в производственном режиме работы, используйте *идентификаторы взаимосвязанности*, позволяющие проводить трассировку вызовов внутри системы.

А когда не следует применять микросервисы?

Этот вопрос мне задавали довольно часто. Первая часть совета будет такой: чем хуже вы разбираетесь в заданной области, тем сложнее вам будет правильно определить ограниченные контексты для сервисов. Как уже говорилось, неправильное определение границ сервисов может привести к необходимости внесения множества изменений в совместную работу сервисов, что является весьма затратным делом. Поэтому, если вы взялись за монолитную систему, область применения которой вам непонятна, сначала уделите время изучению того, чем эта система занимается, а затем, прежде чем разбивать ее на сервисы, присмотритесь к определению четких границ модулей.

Разработка с чистого листа также дается весьма непросто. И дело даже не в том, что область, вероятно, также будет новой, а в том, что разбить на части что-либо уже существующее гораздо проще, чем что-то, чего у вас еще нет! Итак, повторю: присмотритесь сначала к созданию монолитной системы, а ее разбиением займитесь, как только добьетесь от нее стабильной работы.

Сложности, с которыми вы столкнетесь при работе с микросервисами, при масштабировании существенно усугубятся. Если в основном все делается вручную,

то при наличии одного-двух сервисов у вас все может получиться, ну а если сервисов будет пять или десять? Если придерживаться старых положений по мониторингу, где просто изучается статистика загруженности центрального процессора и памяти, этого может хватить только для работы с весьма небольшим количеством сервисов, но чем шире используется совместная работа сервисов, тем обременительнее станет процесс. Вас это начнет беспокоить по мере добавления сервисов, и я надеюсь, что приведенные в книге советы помогут вам заметить возникновение ряда подобных проблем и подскажут их конкретные решения. Ранее я упоминал о том, что компании REA и Gilt потратили время на создание инструментария и инструкций по управлению микросервисами еще до появления возможности их использования в большом количестве. Эти истории только подтвердили для меня важность постепенного расширения, чтобы разобраться в потребностях организации и возможностях внесения изменений, которые помогут вам внедрить микросервисы надлежащим образом.

Напутствие

Архитектуры микросервисов предоставляют вам более широкий выбор и возможность принятия большего количества решений. Принимать решение в этой области приходится намного чаще, чем в области монолитных систем. Абсолютно все решения правильными не бывают, это я заявляю со всей ответственностью. Следовательно, каков должен быть наш выбор, зная, что мы что-то поймем неправильно? Я бы посоветовал найти способы принятия каждого решения в самых скромных масштабах. Тогда, если оно окажется неверным, это повлияет лишь на небольшую часть вашей системы. Научитесь следовать концепции развивающейся архитектуры, при которой ваша система будет способна гибко подстраиваться под ситуацию и изменяться по мере обнаружения новых обстоятельств. Не замышляйте грандиозной перезаписи кода, лучше вместо этого сохраняйте гибкость системы путем серии изменений, вносимых в вашу систему в течение длительного времени.

Надеюсь, что теперь я поделился с вами достаточным объемом информации и опыта, чтобы помочь принять решение о том, насколько микросервисы подходят лично вам. Если вы поймете, что они вам нужны, то я надеюсь, что вы воспримете работу с ними как длительное путешествие, а не как стремление добраться до пункта назначения. Действуйте по нарастающей. Разбивайте свою систему на части, обучаясь в процессе работы. И вы привыкнете к этому: во многих отношениях тренировка во внесении постоянных изменений и в развитии ваших систем является гораздо более важным уроком, чем любой другой, преподнесенный мною в данной книге. Изменения неизбежны. Воспримите это как должное.

Сэм Ньюмен

Создание микросервисов

Перевел с английского Н. Вильчинский

Заведующий редакцией	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художник	*С. Заматевская*
Корректоры	*Т. Курьянович, Е. Павлович*
Верстка	*А. Барцевич*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Подписано в печать 17.12.15. Формат 70×100/16. Бумага писчая. Усл. п. л. 24,510. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает:
профессиональную, популярную и детскую нон-фикшн литературу

Заказать книги оптом можно в наших представительствах:

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: voronej@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: pitvolga@mail.ru

УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: тел./факс: +38 067 545-55-64; e-mail: sasha@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01;
e-mail: gv@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: sales@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс: (812) 703-73-72, (495) 234-38-15

Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате: Яндекс.Деньги, Webmoney и Kiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com)
- Можно оформить доставку заказа через почтоматы, (адреса почтоматов можно узнать на нашем сайте www.piter.com)

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM