Шаблоны проектирования для облачной среды

Облачные платформы призваны исполнить ваши заветные желания: их использование обещает вам практически нулевое время простоя, бесконечную масштабируемость, короткие циклы обратной связи, отказоустойчивость, контроль затрат... Но как этого добиться? Применяя конструкции для облачной среды, разработчики могут создавать гибкие, легко адаптируемые, веб-масштабируемые распределенные приложения, которые обрабатывают огромный пользовательский трафик и загрузку данных. Изучите фундаментальные шаблоны и методы, описываемые в книге, – и вы сможете успешно ориентироваться в динамичном, распределенном виртуальном мире облачных вычислений.
Используя реалистичные примеры и проверенные на опыте методы работы с облачными приложениями, данными, службами, маршрутизацией, автор – разработчик с 25-летним стажем – покажет вам, как проектировать и создавать программное обеспечение, которое прекрасно подходит для современных облачных платформ.

«Эта книга восполняет разрыв между теорией и практикой. Занимательно и познавательно».
— из предисловия Джина Кима, одного из авторов книги «Проект Феникс»

«Книга фокусируется на решении конкретных непростых задач и может использоваться как важное руководство при разработке современных проектов.»
— Дэвид Шмитц, Senacor Technologies

«Книга проливает свет на процесс создания самовосстанавливающихся распределенных отказоустойчивых веб-приложений, требующих минимального технического обслуживания..»
— Равиш Шарма, Stellapps Technologies

О чем идет речь в этой книге:

- жизненный цикл приложений для облачной среды;
- управление конфигурацией в масштабах облака;
- обновление без простоев; службы, имеющие версии и параллельное развертывание;
- обнаружение служб и динамическая маршрутизация;
- управление взаимодействием служб, включая повторные попытки и предохранители.

Корнелия Дэвис (Cornelia Davis) – вице-президент по развитию технологий в компании Pivotal Software. Последние 25 лет своей карьеры она посвятила созданию качественного программного обеспечения и, имея тягу к преподаванию, лично подготовила прекрасных разработчиков.

Для изучения книги читателю необходимы базовые навыки в области проектирования программного обеспечения и умение понимать код, написанный на Java или похожем языке.

ISBN 978-5-97060-807-4

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

Шаблоны проектирования для облачной среды

Корнелия Дэвис

Шаблоны проектирования для облачной среды

MANNING

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Cloud Native Patterns

DESIGNING CHANGE-TOLERANT SOFTWARE

CORNELIA DAVIS
Foreword by GENE KIM

MANNING
Shelter Island

Шаблоны проектирования для облачной среды

Проектирование программного обеспечения, устойчивого к изменениям

КОРНЕЛИЯ ДЭВИС
Предисловие ДЖИНА КИМА

ДМК ПРЕСС
издательство
Москва, 2020

УДК 004.42
ББК 32.972
Д94

Дэвис К.
Д94 Шаблоны проектирования для облачной среды / пер. с анг. Д. А. Беликова. – М.: ДМК Пресс, 2020. – 388 с.: ил.

ISBN 978-5-97060-807-4

Эта книга посвящена облачным платформам, которые обеспечивают многие преимущества – практически нулевое время простоя, бесконечную масштабируемость, короткие циклы обратной связи, отказоустойчивость и контроль затрат. Применяя конструкции для облачной среды, разработчики могут создавать гибкие, легко адаптируемые, веб-масштабируемые распределенные приложения, которые обрабатывают огромный пользовательский трафик и объем данных.

Автор рассматривает методы и шаблоны, ориентированные на приложения для облачной среды – с учетом их жизненного цикла, управления конфигурацией в масштабах облака, обновления без простоев.

Для работы с книгой читателю необходимы базовые навыки в области проектирования программного обеспечения и умение понимать код, написанный на Java или похожем языке.

Издание будет полезно всем, кого интересует развертывание систем на различных облачных платформах.

УДК 004.42
ББК 32.972

ISBN 978-1-617-29532-4 (анг.)
ISBN 978-5-97060-807-4 (рус.)

Моему мужу, Глену.
В тот день, когда я встретила тебя, вся моя жизнь изменилась

Моему сыну, Максу.

Содержание

Предисловие

На протяжении шести лет я имел честь работать с Николь Форсгрен и Джезом Хамблом над отчетом о состоянии DevOps (State of DevOps Report), в котором собраны данные более чем 30 000 респондентов. Одним из величайших откровений для меня стала важность архитектуры программного обеспечения: у высокопроизводительных команд были архитектуры, позволяющие разработчикам быстро и независимо разрабатывать, тестировать и развертывать программное обеспечение для клиентов, делая это безопасно и надежно.

Несколько десятилетий назад мы бы пошутили, сказав, что разработчики программного обеспечения были экспертами только в использовании Visio, создании диаграмм UML и генерации слайдов PowerPoint, на которые никто никогда не смотрел. Если когда-то так и было, то сейчас это точно не так. В наши дни компании одерживают победы и проигрывают на рынке благодаря программному обеспечению, которое они создают. И ничто не влияет на повседневную работу разработчиков больше, чем архитектура, в которой они должны работать.

Эта книга заполняет пробел, охватывая теорию и практику. В сущности, я думаю, что только очень небольшое число людей могло бы написать ее. Корнелия Дэвис обладает уникальной квалификацией. На протяжении нескольких лет, будучи аспирантом, она изучала языки программирования, развивая любовь к функциональному программированию и неизменяемости. В течение нескольких десятков лет она работала с крупными программными системами и помогала крупным компаниям, занимающимся разработкой программного обеспечения, достигать величия.

За последние пять лет я много раз обращался к ней за помощью и советами, часто по таким темам, как CQRS и Event Sourcing, LISP и Clojure (мой любимый язык программирования), опасности императивного программирования и состояния, и даже таким простым вещам, как рекурсия.

Корнелия не просто так начинает с шаблонов, что и делает эту книгу настолько полезной для чтения. Она начинает с основных принципов, а затем доказывает их обоснованность с помощью аргументации, иногда с помощью логики, а иногда с помощью блок-схем. Ее не устраивает одна лишь теория, поэтому затем она реализует эти шаблоны в Java Spring, итерация за итерацией, включая туда то, что вы узнали.

Я нашел эту книгу интересной и познавательной и узнал невероятное количество тем, о которых раньше у меня было лишь поверхностное представление. Теперь я полон решимости реализовать ее примеры в Clojure, желая доказать, что я могу применить эти знания на практике.

Я подозреваю, что вы объедините концепции, которые приведут вас в восторг и, возможно, даже поразят вас. Для меня одной из этих концепций была необходимость централизовать межсекторальные задачи либо с помощью аспектно-ориентированного программирования, либо sidecar-контейнеров Kubernetes или инъекций Spring Retry.

Я надеюсь, что вы найдете эту книгу полезной для чтения, как и я!

Джин Ким,
исследователь и один из авторов книг
The Phoenix Project, *The DevOps Handbook*
и *Accelerate*

От автора

Я начинала свою карьеру в области обработки изображений. Я работала с инфракрасными изображениями в отделе ракетных систем компании Hughes Aircraft, занимаясь такими вещами, как выделение границ и межкадровая корреляция (часть из этого можно найти в приложениях вашего мобильного телефона сегодня – все это было аж в 80-х!).

Одним из вычислений, которые мы часто выполняем при обработке изображений, является среднеквадратическое отклонение. Я никогда не стеснялась задавать вопросы, и один из вопросов, которые я часто задавала в то время, касался этого среднеквадратического отклонения. Коллега неизменно писал следующее:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}.$$

Но я знала формулу среднеквадратического отклонения. Черт возьми, за три месяца я писала ее уже, наверное, полдюжины раз. Я спрашивала: «Что дает нам знание среднеквадратического отклонения в этом контексте?» Среднеквадратическое отклонение используется для определения того, что является «нормальным», чтобы мы могли искать выбросы. Если я рассчитываю стандартное отклонение, а затем нахожу нечто, выходящее за рамки нормы, это признак того, что мой датчик неисправен и мне нужно выбросить кадр с изображением, или это показывает действия потенциального противника?

Какое все это имеет отношение к облачной среде? Никакого. Но это имеет отношение к шаблонам. Дело в том, что я знала схему – расчет среднеквадратического отклонения, – но из-за недостатка опыта в то время я боролась с тем, когда и зачем ее применять.

В этой книге я научу вас шаблонам для облачных приложений – и да, я покажу вам множество «формул», но гораздо больше времени я трачу на *контекст* – когда и для чего применять эти шаблоны. На самом деле шаблоны, как правило, не так уж и сложны (например, повтор запроса, описанный в главе 9, является простой концепцией, которую легко реализовать). Но выбрать, когда применять шаблон и как именно это сделать, может быть непросто. Существует так много понимания контекста, в котором вы будете применять эти шаблоны, и, честно говоря, этот контекст может быть сложным.

Так что же это за контекст? По сути, это одна из распределенных систем. Когда я начинала свою карьеру более 30 лет назад, я знала мало людей, которые работали над распределенными системами, и я не посещала занятия по распределенным системам в колледже. Да, были люди, которые работали в этой области, но, честно говоря, она была довольно нишевой.

Сегодня подавляющее большинство программного обеспечения является распределенной системой. Некоторые части вашего программного обеспечения работают в браузере, а другие – на сервере или, осмелюсь сказать, целой куче серверов. Эти серверы могут работать в вашем корпоративном центре обработки

данных, или они могут находиться в центре обработки темных данных в Прайневилле, штат Орегон, или же и там, и там. И все эти фрагменты взаимодействуют друг с другом по сети, возможно, через интернет, и, вероятно, данные вашего программного обеспечения также являются широко распределенными. Говоря проще, ПО для облачной среды – это распределенная система. Кроме того, все постоянно меняется – могут случаться проблемы с серверами, в сетях часто бывают простои, пусть даже кратковременные, а устройства хранения могут выходить из строя без предупреждения – однако ожидается, что ваше программное обеспечение будет работать. Это довольно сложный контекст.

Но его можно полностью подчинить себе! Цель этой книги – помочь вам понять этот контекст и предоставить вам инструменты для того, чтобы стать опытным архитектором и разработчиком программного обеспечения для облачных сред.

Никогда прежде я не была более интеллектуально стимулирована, чем сейчас. Во многом это связано с тем, что технологический ландшафт существенно меняется, и в центре внимания находятся облачные технологии. Мне очень нравится то, чем я зарабатываю на жизнь, и я хочу, чтобы все, особенно вы, получали удовольствие от написания программного обеспечения так же, как и я. Вот почему я и написала эту книгу: я хочу поделиться с вами сумасшедшими классными проблемами, над которыми мы работаем, и помочь вам на пути к решению этих проблем. Для меня большая честь иметь возможность сыграть даже небольшую роль в вашем пути к облачным технологиям.

Благодарности

Мое путешествие по облачным технологиям началось всерьез в 2012 году, когда мой начальник Том Магуайр попросил меня заняться моделью PaaS (Platform as a Service – Платформа как услуга). Будучи членами группы по архитектуре в офисе технического директора EMC, изучение новых технологий не было для нас чем-то новым, но, боже, у нас получилось! Я всегда буду благодарна Тому за этот импульс и за предоставленную мне возможность.

К началу 2013 года я уже знала достаточно, чтобы понять, что этим я буду заниматься в обозримом будущем, и с созданием компании Pivotal Software у меня было место для этой работы. В первую очередь я хочу поблагодарить Элизабет Хендриксон за то, что она пригласила меня на вечеринку Cloud Foundry, – даже когда я еще работала в EMC, – и за то, что познакомила меня с Джеймсом Уоттерсом. Я часто говорю, что лучший шаг в моей карьере – работа на Джеймса. Я благодарю его за те многочисленные возможности, которые он предоставил мне, за то, что он доверился мне и позволил мне максимально реализовать себя, за бесчисленные разговоры с высокой пропускной способностью, в ходе которых мы все вместе познавали облачную среду, и за то, что мы так подружились за последние шесть лет.

Я благодарна за то, что являюсь частью Pivotal с момента ее создания, где я проходила обучение наряду со многими яркими, преданными и отзывчивыми коллегами. Я хотела бы поблагодарить Элизабет Хендриксон, Джошуа МакКенти, Эндрю Клэй-Шафера, Скотта Яру, Феррана Роденаса, Мэтта Стайна, Рагвендера Арни и многих других (пожалуйста, простите меня, если я кого не упомянула) за то, что помогли мне учиться и за то, что разделили со мной шесть лучших лет моей жизни! Я также хотела бы поблагодарить Pivotal, в частности Иана Эндрюса и Келли Холл, за спонсирование мини-книги *Cloud-Native Foundations*.

Я столькому научилась у своих коллег, больше, чем могу себе представить. Спасибо каждому из вас. Но я бы хотела выделить Джина Кима. Я вспоминаю тот вечер, когда мы встретились (и еще раз благодарю Элизабет Хендриксон за ту роль, которую она сыграла, чтобы эта встреча стала возможной), и сразу же поняла, что мы будем сотрудничать на протяжении долгого времени. Я благодарю Джина за возможность поработать с ним на саммите DevOps Enterprise Summit, благодаря которому я познакомилась с большим количеством новаторов, работающих в самых разных компаниях. Я благодарю его за ободряющие и расширяющие сознание беседы и за то, что он написал предисловие к этой книге.

Конечно, я благодарю издательство Manning Publications за возможность написать эту книгу, и прежде всего Майка Стивенса, который помог мне перейти от праздного любопытства к тому, чтобы сделать реальный шаг в написании книги. Я очень благодарна своему редактору по развитию Кристине Тейлор. Она взяла начинающего автора, у которого на старте была мешанина идей из 20 глав, и раннюю главу длиной около 70 страниц, и помогла мне создать книгу, которая имеет структуру и реальную сюжетную линию. Она работала со мной более двух с половиной лет, подбадривая меня, когда я была в отчаянии, и поздравляла меня, когда

я была на пике своих достижений. Я также благодарю производственную команду, в том числе Шарон Уилки, Дейрдру Хиам, Нила Кролл, Кэрол Шилдс и Николь Берд, которые несут ответственность за, что нам удалось выйти на финишную прямую. И спасибо моим рецензентам – Бачиру Тихани, Карлосу Роберто Варгасу Монтеро, Дэвиду Шмитцу, Дониёру Улмасову, Грегору Зуровски, Джареду Дункану, Джону Гатри, Хорхе Иезекиилю Бо, Кельвину Джонсону, Кенту Р. Шпилнер, Лонни Сметана, Луису Карлосу Санчесу Гонсалесу, Марку Миллеру, Питеру Полу Селларсу, Равишу Шарма, Сергею Евсикову, Серхио Мартинесу, Шанкеру Джанакираману, Стефану Хеллвегеру, Вину Оо и Зородзайи Мукуя. Ваш отзыв оказал заметное влияние на конечный результат.

И больше всего я благодарю своего мужа Глена и сына Макса за их терпение, поддержку и неизменную веру в меня. Они, как и все остальные, являются теми двумя людьми, которые сделали это возможным не только благодаря тому, что оказывали мне поддержку в течение последних трех лет, но и помогали мне заложить основу за несколько десятилетий до этого. От всей души благодарю вас обоих. И за то, что ты, Макс, полюбил компьютеры так же сильно, как и я, и позволил мне стать зрителем в этой поездке, – это двойная шоколадная глазурь на торте «Death by chocolate» – спасибо!

Об этой книге

Кому стоит прочитать эту книгу

Переход в «облако» – это скорее о том, как вы разрабатываете свои приложения, нежели о том, где вы их развертываете. Эта книга представляет собой руководство по разработке надежных приложений, которые процветают в динамичном, распределенном, виртуальном мире облака. В ней представлена ментальная модель облачных приложений, а также шаблоны, методы и инструменты, поддерживающие их конструкцию. Здесь вы найдете реалистичные примеры и советы экспертов для работы с приложениями, данными, сервисами, маршрутизацией, и много чего еще.

По сути, это книга об архитектуре, в которой содержатся примеры кода для поддержания обсуждений, связанных с проектированием. Вы увидите, что я часто ссылаюсь на различия между шаблонами, которые я здесь описываю, и тем, как мы могли что-то делать в прошлом. Однако наличия опыта или даже знания шаблонов предшествующей эры не требуется. Поскольку я рассматриваю не только сами шаблоны, но и их мотивы и нюансы контекста, в котором они применяются, они могут оказаться очень полезными для вас, независимо от того, сколько лет вы занимаетесь разработкой программного обеспечения.

И хотя в этой книге приведено много примеров, содержащих код, это не книга по программированию. Она не научит вас программировать, если вы пока еще не знаете основ. Примеры кода написаны на Java, но с каким бы языком вы ни работали прежде, у вас не должно возникнуть проблем при чтении этой книги. Знание основ взаимодействия типа «клиент /служба», особенно через протокол HTTP, также полезно, но не обязательно.

Как устроена эта книга: дорожная карта

Эта книга состоит из 12 глав, разделенных на две части.

Часть I определяет облачный контекст и представляет характеристики среды, в которой вы будете развертывать свое программное обеспечение.

- Глава 1 дает определение термина *cloud-native* и отличает ее от облака. Он представляет мысленную модель, вокруг которой можно создать шаблоны, которые появятся позже: объектами этой модели являются *приложения/службы*, *взаимодействия* между службами и *данные*.
- Глава 2 посвящена облачным операциям – шаблонам и методам, используемым для поддержания работоспособности программного обеспечения для облачной среды в рабочем окружении во время неизбежных сбоев, которые обрушиваются на него.
- Глава 3 знакомит вас с облачной платформой, средой разработки и выполнения, которая обеспечивает поддержку и даже реализацию многих шаблонов, представленных во второй части книги. Хотя важно понимать все последующие шаблоны, вам не нужно реализовывать их все самостоятельно.

Во второй части подробно рассматриваются сами шаблоны.

- Глава 4 посвящена облачному *взаимодействию*, а также касается *данных*, знакомя вас с событийно-ориентированным обменом данными в качестве альтернативы привычному стилю «запрос/ответ». Хотя последнее практически повсеместно распространено в большинстве программных продуктов, событийно-ориентированный подход часто дает значительные преимущества сильно распределенному облачному программному обеспечению, и при изучении шаблонов важно учитывать оба протокола.
- Глава 5 посвящена облачным *приложениям/службам* и их связи с *данными*. В ней рассказывается, как развертывать приложения в качестве избыточных экземпляров часто в большом масштабе, для чего и как делать их не сохраняющими состояние, как привязать их к специальной службе с фиксацией состояния.
- В главе 6 рассказывается о том, как можно последовательно поддерживать конфигурацию приложений при развертывании большого числа экземпляров в широко распределенной инфраструктуре, а также говорится о правильном применении конфигурации приложений, когда среда, в которой они работают, постоянно меняется.
- Глава 7 охватывает жизненный цикл приложения и многочисленные способы обновления без остановки, включая последовательные обновления и сине-зеленые обновления.
- Глава 8 посвящена *взаимодействию* в облачной среде. Она фокусируется на том, как приложения могут находить нужные им сервисы (обнаружение сервисов), даже когда те постоянно перемещаются, и на том, как запросы в конечном итоге попадают в нужные сервисы (динамическая маршрутизация).
- Глава 9 сосредоточена на взаимодействии на стороне клиента. После объяснения необходимости избыточности взаимодействия и знакомства с повторными отправками запроса (при которых запросы повторяются, если они изначально были неудачными) в главе рассматриваются проблемы, которые могут возникнуть в результате неправильного применения повторных отправок, и способы избежать этих проблем.
- Глава 10 посвящена взаимодействию на стороне сервера. Даже если клиенты, инициирующие взаимодействие, делают это ответственно, служба все равно должна защищать себя от неправильного использования и от перегруженности трафиком. В этой главе рассматриваются шлюзы API и предохранители.
- Глава 11 посвящена *приложениям* и *взаимодействию*. В ней рассматриваются средства наблюдения за поведением и производительностью распределенной системы, составляющей ваше программное обеспечение.
- Глава 12 посвящена *данным* и имеет существенное влияние на *взаимодействие* между службами, составляющими ваше облачное программное обеспечение. В ней автор рассказывает о шаблонах, использующихся для разбиения того, что когда-то было монолитной базой данных, на распределенную структуру данных, в конечном итоге возвращаясь к событийно-ориентированным шаблонам, описанным в начале второй части книги.

О коде

Эта книга содержит много примеров исходного кода, как в пронумерованных листингах, так и внутри обычного текста. В обоих случаях исходный код форматируется с помощью шрифта фиксированной ширины, `вот так`, чтобы отделить его от обычного текста. Иногда код также выделяется **жирным шрифтом**, чтобы выделить фрагменты, на которые вам следует обратить внимание.

Во многих случаях исходный код был переформатирован; мы добавили разрывы строк и переработали отступы, чтобы обеспечить доступное пространство для страниц в книге. В редких случаях даже этого было недостаточно, и листинги содержат маркеры продолжения строки (➥). Кроме того, комментарии в исходном коде часто удалялись из листингов, когда описание кода приводилось в тексте. Многие листинги снабжены аннотациями, которые используются для выделения важных понятий.

Код, содержащийся в примерах этой книги, доступен для скачивания с вебсайта издательства Manning по адресу **https://www.manning.com/books/cloud-native-patterns** и на GitHub на странице **https://github.com/cdavisafc/cloudnative-abundantsunshine**.

Отзывы и пожелания

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

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

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** или **www.дмк.рф** на странице с описанием соответствующей книги.

Список опечаток

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

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

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Об авторе

Корнелия Дэвис – вице-президент по технологиям в компании Pivotal, где она работает над технологической стратегией для Pivotal и клиентов компании. В настоящее время она работает над тем, чтобы объединить различные модели облачных вычислений IaaS, aPaaS, CaaS и FaaS в комплексное предложение, позволяющее ИТ-организациям функционировать на самом высоком уровне.

Будучи ветераном отрасли с почти тридцатилетним опытом в обработке изображений, научной визуализации, работе с распределенными системами и архитектурами веб-приложений, а также облачными платформами, Корнелия имеет степень бакалавра и магистра компьютерных наук от Калифорнийского государственного университета, в Нортридже. Она также изучала теорию алгоритмов и языки программирования в университете Индианы.

Учитель в глубине души, Корнелия провела последние тридцать лет, занимаясь созданием передового программного обеспечения и его разработкой.

В свободное от работы время она занимается йогой или готовит на кухне.

Об иллюстрации на обложке

Рисунок на обложке книги озаглавлен «Облачение русской повитухи в 1764 году». Иллюстрация взята из опубликованной между 1757 и 1772 годом в Лондоне книги Томаса Джефферисса «Коллекция платьев разных народов, древних и современных» (четыре тома). На титульном листе указано, что это медные гравюры ручной работы, украшенные гуммиарабиком.

Томаса Джефферисса (1719–1771) называли «географом короля Георга III». Он был английским картографом, ведущим создателем карт своего времени. Томас гравировал и печатал карты для правительственных и других государственных учреждений и выпускал обширный спектр коммерческих карт и атласов, особенно касающихся Северной Америки. Его работа в качестве картографа пробудила интерес к местному дресс-коду, блистательно представленному в этой коллекции. Он был принят в тех землях, которые исследовал и наносил на карту. Увлечение далекими землями и путешествия ради удовольствия были относительно новым явлением в конце XVIII века, и такие коллекции, как эта, были популярны, знакомя как туристов, так и путешественников, сидящих в креслах, с жителями других стран.

Разнообразие рисунков в этом издании Джефферисса ярко свидетельствует об уникальности и индивидуальности народов мира около 200 лет назад. С тех пор дресс-код изменился, а богатое в ту пору разнообразие в зависимости от региона и страны исчезло. Сейчас часто трудно отличить жителей одного континента от другого. Возможно, пытаясь взглянуть на это с оптимизмом, мы обменяли культурное и визуальное разнообразие на более разнообразную личную жизнь – или на более разнообразную и интересную интеллектуальную и техническую жизнь.

В то время когда трудно отличить одну компьютерную книгу от другой, издательство Manning празднует изобретательность и инициативу компьютерного бизнеса с помощью обложек книг, основанных на богатом разнообразии жизни регионов двухвековой давности, которое ожило благодаря рисункам Джефферисса.

Часть I

КОНТЕКСТ ОБЛАЧНОЙ СРЕДЫ

Возможно, это звучит как клише, первая часть книги подготавливает основу для дальнейшей работы. Полагаю, что можно было бы сразу перейти к шаблонам, о которых, я уверена, вам не терпится узнать (обнаружение служб, предохранители и т. д.), но я хочу, чтобы вы поняли эти шаблоны на очень глубоком уровне, поэтому эти первые главы необходимы. Понимание контекста, в котором будут работать ваши приложения, инфраструктуры, а также более человеческих элементов позволит вам применять шаблоны наиболее эффективным образом. Ожидания ваших клиентов от ваших цифровых продуктов (постоянное развитие и нулевое время простоя) и то, как вы и ваши коллеги разрабатываете эти продукты (наделенные полномочиями команды и отсутствием инцидентов), имеют отношение к шаблонам проектирования, о которых вы, возможно, даже и не догадывались.

Одна из главных вещей, которые я делаю в первой главе, – это определение понятия *cloud-native*, проводя различие между ним и термином *cloud* (внимание: спойлер! Последний термин касается вопроса *где*, а предыдущий – отвечает на вопрос *как*, что действительно интересно). Я также устанавливаю ментальную модель, вокруг которой организована вторая часть книги.

Вторая глава посвящена работе с приложениями для облачной среды. Я слышу, что некоторые из вас думают: «Я – разработчик, мне не нужно об этом беспокоиться», но, пожалуйста, забудьте на мгновение о своем недоверии. Операционные методы, отвечающие требованиям некоторых ваших клиентов, тотчас же транслируются на требования к вашему программному обеспечению.

И наконец, в третьей главе я расскажу о платформах, которые удовлетворяют потребности разработки и эксплуатации. Хотя многие из шаблонов, которые я рассматриваю во второй части книги, абсолютно необходимы для создания высококачественного программного обеспечения, они не должны быть реализованы вами в полной мере; правильная платформа может оказать вам большую помощь.

Так что если у вас есть соблазн пропустить первую часть, не делайте этого. Я обещаю, что вложенные сюда инвестиции окупятся позже.

Глава 1

Вы продолжаете использовать это слово: определение понятия «cloud-native»

Это не вина Amazon. В воскресенье 20 сентября 2015 года в платформе Amazon Web Services (AWS) произошел серьезный сбой. При растущем числе компаний, работающих с критически важными рабочими нагрузками на AWS, даже с основными сервисами, ориентированными на клиентов, сбой в работе AWS может привести к далеко идущим последующим системным сбоям. В этом случае Netflix, Airbnb, Nest, IMDb и другие испытали простой, что отразилось на их клиентах и в конечном итоге на их бизнесе. Основное отключение длилось около пяти часов (или более, в зависимости от того, как считать), что привело к еще более длительным сбоям для клиентов AWS, которых это затронуло, прежде чем их системы восстановились.

Если вы компания Nest, вы платите AWS, потому хотите сосредоточиться на создании эффективного результата для своих клиентов, а не на проблемах инфраструктуры. В качестве части данной сделки AWS отвечает за поддержание своих систем и за то, чтобы вы также поддерживали функционирование своей компании. Если у AWS возникают простои, можно легко обвинить в этом Amazon.

Но вы ошибаетесь. Компания Amazon не виновата в сбое.

Подождите! Не отбрасывайте эту книгу в сторону. Пожалуйста, выслушайте меня. Мое утверждение раскрывает суть вопроса и объясняет цели книги.

Во-первых, позвольте мне прояснить кое-что. Я не утверждаю, что Amazon и другие облачные провайдеры не несут ответственности за нормальное функционирование своих систем; очевидно, что это так. И если провайдер не соответствует определенным уровням обслуживания, его клиенты могут найти альтернативу, и они это сделают. Поставщики услуг обычно предоставляют соглашение об уровне предоставления услуги (SLA). Amazon, например, предоставляет 99,95 % гарантии бесперебойной работы для большинства своих услуг.

Я говорю о том, что приложения, которые вы запускаете в конкретной инфраструктуре, могут быть более стабильными, чем сама инфраструктура. Как такое возможно? Это, друзья мои, как раз то, чему научит вас эта книга.

Давайте на минутку вернемся к перебоям в работе AWS, произошедшим 20 сентября. Netflix, одна из множества компаний, затронутых перебоями, является то-

повым сайтом в Соединенных Штатах, если судить по количеству потребленной пропускной способности (36 %). Но даже несмотря на то, что перерыв в работе Netflix затрагивает большое количество людей, по поводу произошедшего в AWS компания сказала следующее:

> *В Netflix действительно были кратковременные перебои с доступом в затронутом регионе, но мы обошли все существенные последствия, потому что упражнения с Chaos Kong готовят нас к подобным инцидентам. Проводя эксперименты на регулярной основе, которые симулируют региональный сбой, нам удается выявлять системные недостатки на ранней стадии и исправлять их. Когда US-EAST-1 стал недоступен, наша система была уже достаточно сильна, чтобы справиться с проблемой*[1].

Netflix смог быстро восстановиться после сбоя, став полностью функциональным всего лишь через несколько минут после начала инцидента. Netflix, попрежнему работающий на AWS, был полностью работоспособен, даже когда сбой продолжался.

ПРИМЕЧАНИЕ Как Netflix удалось так быстро восстановиться? Избыточность.

Ни один аппаратный компонент не может гарантированно работать в 100 % случаев, и поэтому, как это уже принято на протяжении некоторого времени, мы устанавливаем избыточные системы. Именно этим и занимается AWS, делает эти избыточные абстракции доступными для своих пользователей.

В частности, AWS предлагает услуги во многих регионах. Например, на момент написания этих строк ее платформа Elastic Compute Cloud (EC2) работает и доступна в Ирландии, Франкфурте, Лондоне, Париже, Стокгольме, Токио, Сеуле, Сингапуре, Мумбае, Сиднее, Пекине, Нинся, Сан-Паулу, Канаде и в четырех районах в Соединенных Штатах (Вирджиния, Калифорния, Орегон и Огайо). И в пределах каждого региона сервис дополнительно разбивается на многочисленные зоны доступности, которые сконфигурированы для изоляции ресурсов одной зоны от другой. Эта изоляция ограничивает последствия сбоя в одной зоне, распространяющегося на сервисы в другой зоне.

На рис. 1.1 изображены три региона, каждый из которых содержит четыре зоны доступности.

Приложения запускаются в зонах доступности и – вот важная часть – могут работать в нескольких зонах и нескольких регионах. Напомню, что минуту назад я утверждала, что избыточность является одним из ключей к безотказной работе.

Давайте разместим на рис. 1.2 логотипы в этой диаграмме, чтобы гипотетически обозначить запущенные приложения. (У меня нет четкого представления касательно того, как Netflix, IMDb или Nest развертывали свои приложения; это чисто гипотетически, но тем не менее показательно.)

---

[1] Для получения дополнительной информации о Chaos Kong см. статью «Chaos Engineering Upgraded» в блоге Netflix Technology (**http://mng.bz/P8rn**).

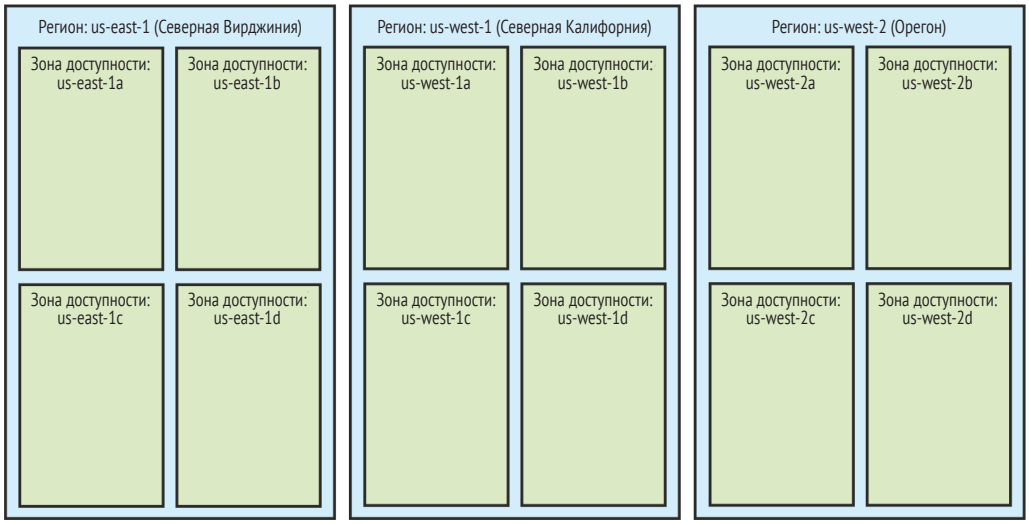


Рис. 1.1 ❖ AWS делит предлагаемые сервисы на регионы и зоны доступности. Регионы отображаются в географических областях, а зоны доступности обеспечивают дополнительную избыточность и изоляцию в пределах одного региона

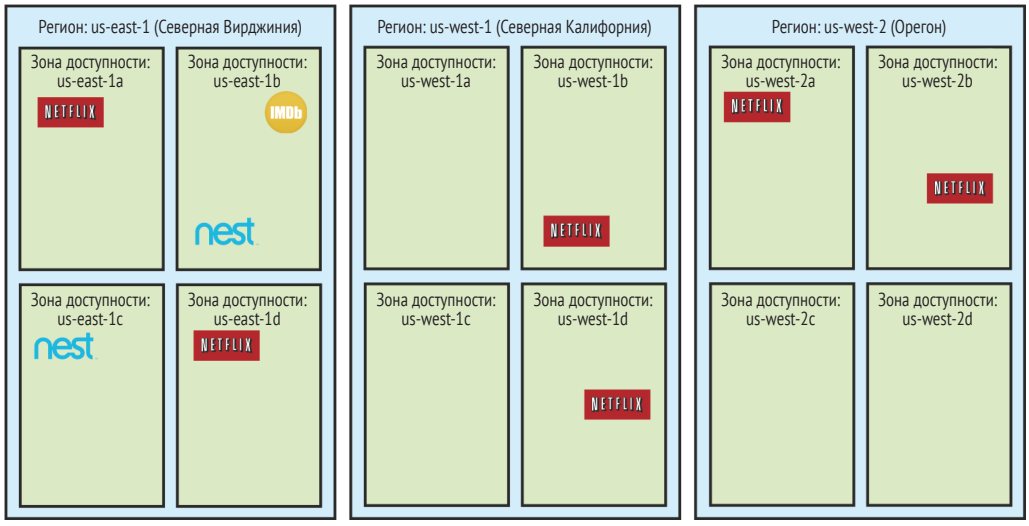


Рис. 1.2 ❖ Приложения, развернутые на AWS, могут быть развернуты в одной зоне доступности (IMDb) или в нескольких (Nest), но только в одном регионе, или в нескольких зонах доступности и нескольких регионах (Netflix), что обеспечивает различные профили устойчивости

На рис. 1.3 показано отключение в одном регионе, например отключение в сентябре 2015 года. В этом случае погас только us-east-1.

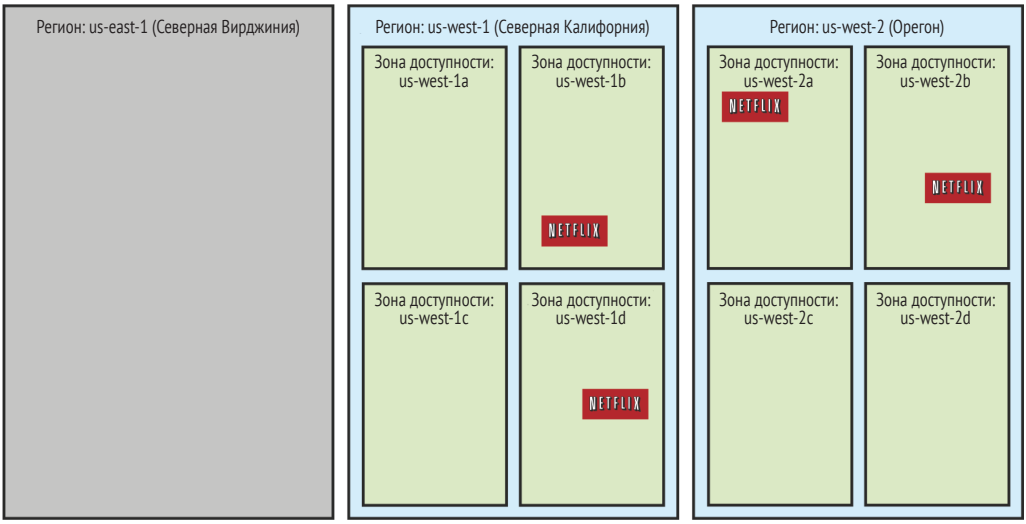


Рис. 1.3 ❖ Если приложения правильно спроектированы и развернуты, программное обеспечение может пережить даже обширный сбой, например сбой всего региона

На этом простом графике сразу видно, что Netflix удалось справиться с отключением намного лучше, чем другим компаниям; у него уже были приложения, работающие в других регионах AWS, и он смог легко перенаправить весь трафик на исправные экземпляры. И хотя похоже, что переключение на другие регионы не было автоматическим, Netflix предвидел (и даже испытал на практике!) возможный сбой, такой как этот, и спроектировал свое программное обеспечение и разработал свои методы для компенсации[1].

> **ПРИМЕЧАНИЕ** Программное обеспечение для облачной среды предназначено для прогнозирования сбоев и остается стабильным, даже если инфраструктура, в которой оно работает, испытывает перебои в работе или же изменяется.

Разработчики приложений, а также персонал службы поддержки и эксплуатации должны изучить и применить новые шаблоны и методы для создания и управления программным обеспечением для облачной среды, и эта книга учит этому. Вы, наверное, думаете, что это не ново и что компании в особенно критически важных сферах бизнеса, таких как финансы, уже некоторое время используют системы в режиме active-active, и вы правы. Но новизна состоит в том, каким образом это достигается.

В прошлом реализация таких способов аварийного переключения обычно была индивидуальным решением, связанным с развертыванием системы, которая изначально не была предназначена для адаптации к основным системным сбоям. Знания, необходимые для достижения требуемых соглашений о предоставлении услуги, часто ограничивались несколькими «рок-звездами». Создавались необыч-

[1] Для получения более подробной информации о восстановлении компании см. статью Ника Хита «Перебои в работе AWS: как Netflix выдержал бурю, готовясь к худшему» (**http://mng.bz/J8RV**).

ные механизмы проектирования, конфигурации и тестирования, чтобы системы реагировали на этот сбой соответствующим образом.

Разница между этим и тем, что делает Netflix сегодня, начинается с принципиального различия в философии. При прежних подходах изменение или неудача рассматривались как исключение. А Netflix и многие другие крупные интернет-компании, такие как Google, Twitter, Facebook и Uber, *воспринимают изменение или неудачу как правило*.

Эти компании изменили архитектуру своего программного обеспечения и методы разработки, чтобы проектирование на случай неисправности стало неотъемлемой частью процесса создания, разработки и управления программным обеспечением.

ПРИМЕЧАНИЕ Неудача – это правило, а не исключение.

1.1. Современные требования к приложениям

Опыт взаимодействия с помощью цифровых технологий больше не является для нас обузой. Он играет важную роль во многих или большинстве действий, в которых мы участвуем ежедневно. Эта повсеместность раздвинула границы того, что мы ожидаем от программного обеспечения, которое используем: мы хотим, чтобы приложения были всегда доступны, чтобы в них постоянно появлялись новые функции и чтобы они обеспечивали индивидуальный подход. Как сделать так, чтобы эти ожидания были оправданы, – вот на что нужно обратить внимание с самого начала жизненного цикла «от идеи до производства». Вы, разработчик, являетесь одной из сторон, ответственных за удовлетворение этих потребностей. Давайте рассмотрим ряд ключевых требований.

1.1.1. Нулевое время простоя

Сбой в работе AWS, случившийся 20 сентября 2015 года, демонстрирует одно из ключевых требований современного приложения: оно должно быть всегда доступно. Прошли те времена, когда допускались даже короткие окна обслуживания, в течение которых приложения были недоступны. Мир все время находится в режиме онлайн. И хотя незапланированные простои всегда были нежелательны, их влияние достигло поразительных высот. Например, в 2013 году Forbes подсчитал, что Amazon потерял почти 2 млн долл. из-за тринадцатиминутного незапланированного отключения[1]. Простои, не важно, запланированы они или нет, приводят к потере значительной части доходов и неудовлетворенности клиентов.

Но поддержание безотказной работы – это проблема не только оперативной группы. Разработчики программного обеспечения или архитекторы несут ответственность за создание системного дизайна со слабосвязанными компонентами, которые могут быть развернуты, чтобы позволить избыточности компенсировать неизбежные сбои, и с воздушными зазорами, которые препятствуют тому, чтобы эти сбои обрушивались на всю систему. Они также должны разрабатывать программное обеспечение, позволяющее выполнять запланированные мероприятия, такие как обновления, без простоев.

---

[1] Для получения более подробной информации см. статью Келли Клэй «Amazon.com падает, теряя $ 66 240 в минуту» на сайте Forbes (**http://mng.bz/wEgP**).

1.1.2. Сокращенные контуры обратной связи

Также очень важно умение часто выпускать код. В связи со значительной конкуренцией и постоянно растущими ожиданиями потребителей обновления приложений становятся доступными для клиентов несколько раз в месяц или в неделю либо в некоторых случаях даже несколько раз в день. Стимулирование клиента, безусловно, очень важно, но, возможно, самым большим драйвером этих непрерывных релизов является снижение риска.

С момента, когда у вас появляется идея для создания какой-то функции, вы берете на себя определенный уровень риска. Это хорошая идея? Смогут ли клиенты воспользоваться этим? Можно ли реализовать это более эффективным способом? Как бы вы ни пытались предсказать возможные результаты, реальность часто отличается от того, что вы ожидаете. Лучший способ получить ответы на такие важные вопросы – выпустить раннюю версию функции и получить отклики. Используя эти отклики, вы можете внести коррективы или даже полностью изменить направление. Частые релизы программного обеспечения сокращают контуры обратной связи и снижают риск.

Монолитные системы программного обеспечения, которые доминировали в последние несколько десятилетий, нельзя выпускать достаточно часто. Необходимо было тестировать слишком много тесно взаимосвязанных подсистем, созданных и протестированных независимыми командами как одно целое, прежде чем можно было бы применить нередко хрупкий процесс упаковки. Если дефект обнаруживался в конце фазы интеграционного тестирования, длительный и трудоемкий процесс начинался заново. Для достижения необходимой гибкости при отправке программного обеспечения в рабочее окружение необходимы новые архитектуры программного обеспечения.

1.1.3. Мобильная и мультидевайсная поддержка

В апреле 2015 года Comscore, ведущая компания, занимающаяся измерениями интернет-аудитории, опубликовала отчет, в котором говорится, что впервые количество пользователей, выходящих в интернет с помощью мобильных устройств, превзошло число тех, кто использует для этой цели настольные компьютеры[1]. Современные приложения должны поддерживать как минимум две платформы для мобильных устройств, iOS и Android, как и десктопы (которые по-прежнему используются значительной частью интернет-пользователей).

Кроме того, пользователи все чаще ожидают, что при работе с приложением они будут плавно переходить с одного устройства на другое во время своего путешествия по сети на протяжении дня. Например, пользователи могут смотреть фильм на Apple TV, а затем переходить к просмотру программы на мобильном устройстве, когда они едут на поезде в аэропорт. Более того, шаблоны использования на мобильном устройстве значительно отличаются от шаблонов настольного компьютера. Банки, например, должны быть в состоянии удовлетворять часто повторяющиеся обновления приложений от пользователей мобильных устройств, которые ожидают еженедельной выплаты заработной платы.

---

[1] Для ознакомления с кратким отчетом см. сообщение в блоге Кейт Дрейер от 13 апреля 2015 года на сайте Comscore (**http://mng.bz/7eKv**).

Для удовлетворения этих потребностей необходим правильный подход к разработке приложений. Базовые сервисы должны быть реализованы таким образом, чтобы они могли поддерживать все клиентские устройства, обслуживающие пользователей, а система должна адаптироваться к требованиям расширения и сжатия.

1.1.4. Устройства, подключенные к сети, также известные как интернет вещей

Интернет уже используется не только для подключения людей к системам, которые размещены в центрах обработки данных и обслуживаются оттуда. Сегодня миллиарды устройств подключены к интернету, что позволяет следить за ними и даже контролировать их с помощью других подключенных объектов. Один только рынок автоматизированных устройств для домашнего пользования, представляющий крошечную часть подключенных устройств, которые образуют интернет вещей (IoT), будет оцениваться в 53 млрд долл. к 2022 году[1].

Современные дома такого типа оснащены датчиками и устройствами с дистанционным управлением, такими как датчики движения, камеры, интеллектуальные термостаты и даже системы освещения. И все это чрезвычайно доступно; после разрыва трубы при температуре –26 градусов по Фаренгейту несколько лет назад у меня была скромная система, включающая в себя подключенный к интернету термостат и несколько датчиков температуры. На это я потратила менее 300 долл. Другие устройства с выходом в интернет включают в себя автомобили, бытовую и сельскохозяйственную технику, реактивные двигатели и суперкомпьютер, который большинство из нас носят с собой в карманах (смартфон).

Подключенные к интернету устройства меняют природу программного обеспечения, которое мы создаем, двумя основными способами. Во-первых, объем данных, передаваемых через интернет, резко увеличивается. Миллиарды устройств передают данные много раз в минуту или даже в секунду[2]. Секунда, чтобы получить и обработать эти огромные объемы данных. Для этого вычислительная основа должна значительно отличаться от тех, что были в прошлом. Она становится более распределенной при использовании вычислительных ресурсов, расположенных на «краю», ближе к тому месту, где находится устройство, подключенное к сети. Это различие в объеме данных и архитектуре инфраструктуры требует новых конструкций и методов программного обеспечения.

1.1.5. Управление с помощью данных

Принимая во внимание некоторые требования, которые я изложила к этому моменту, заставляю вас думать о данных более целостным способом. Объемы данных растут, источники становятся все более распространенными, а циклы разра-

---

[1] Вы можете прочитать больше о выводах, сделанных в ходе исследования Zion Market Research на сайте GlobeNewswire (**https://www.globenewswire.com/news-release/2017/04/12/959610/0/en/Smart-Home-Market-Size-Share-will-hit-53-45-Billion-by-2022.html**).

[2] Компания Gartner прогнозирует, что в 2017 году количество подключенных к интернету устройств во всем мире составит 8,4 млрд единиц; см. отчет Gartner на странице **https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016**.

ботки программного обеспечения сокращаются. В совокупности эти три фактора делают большую централизованную общую базу данных непригодной для использования.

Например, реактивный двигатель с сотнями датчиков часто отключается от центров обработки данных, в которых размещены такие базы данных, и ограничения полосы пропускания не позволяют передавать все данные в центр обработки данных в течение коротких окон, когда устанавливается соединение. Кроме того, общие базы данных требуют большого количества процессов и координации во множестве приложений для рационализации различных моделей данных и сценариев взаимодействия; это является серьезным препятствием для сокращения циклов релизов ПО.

Вместо единой общей базы данных эти требования к приложениям требуют создания сети небольших локализованных баз данных и программного обеспечения, которое управляет связями между данными в рамках этой федерации систем управления данными. Эти новые подходы приводят к необходимости разработки программного обеспечения и гибкости управления вплоть до уровня данных.

Наконец, все только что полученные доступные данные имеют небольшую ценность, если они не используются. Современные приложения должны все чаще использовать данные, чтобы предоставить клиенту более качественный результат с помощью более интеллектуальных приложений. Например, картографические приложения используют данные GPS от автомобилей и мобильных устройств, подключенных к сети, наряду с данными о проезжей части и местности, чтобы предоставить отчет о дорожном движении в режиме реального времени и инструкции по выбору маршрута. На смену приложениям прошлых десятилетий, в которых реализовывались тщательно разработанные алгоритмы, тщательно настроенные для ожидаемых сценариев использования, приходят приложения, которые постоянно пересматриваются. Они даже могут сами настраивать свои внутренние алгоритмы и конфигурации.

Эти требования *пользователей* – постоянная доступность, постоянное развитие и частые релизы, легкая масштабируемость и интеллектуальные возможности – нельзя удовлетворить с помощью систем проектирования программного обеспечения и управления прошлого. Но что же характеризует программное обеспечение, которое может соответствовать этим требованиям?

1.2. Знакомство с программным обеспечением для облачной среды

Ваше программное обеспечение должно работать 24 часа в сутки и семь дней в неделю. Вы должны иметь возможность часто выпускать релизы, чтобы давать своим пользователям мгновенное удовлетворение, которое они ищут. Мобильность и ваши пользователи в состоянии «always connected» обусловливают необходимость того, чтобы ваше программное обеспечение реагировало на бóльшие и более нестабильные объемы запросов, чем когда-либо прежде. А устройства, подключенные к сети («вещи»), образуют распределенную структуру данных беспрецедентного размера, которая требует новых подходов к хранению и обработке. Эти потребности наряду с доступностью новых платформ, на которых вы

можете запускать программное обеспечение, привели непосредственно к появлению нового архитектурного стиля программного обеспечения – программного обеспечения для облачной среды.

1.2.1. Определение понятия «cloud-native»

Что же характеризует *программное обеспечение для облачной среды* (cloud-native software)? Давайте еще проанализируем предыдущие требования и посмотрим, к чему они приведут. Рисунок 1.4 делает первые несколько шагов, перечисляя требования сверху и показывая причинно-следственные связи, идущие вниз. Приведенный ниже список объясняет детали:

- программное обеспечение, которое всегда работает, должно быть устойчивым к сбоям и изменениям инфраструктуры независимо от того, запланированы они или нет. Поскольку контекст, в котором оно работает, испытывает такого рода неизбежные изменения, программное обеспечение должно иметь возможность адаптироваться. При правильном построении, развертывании и управлении состав независимых элементов может ограничивать радиус взрыва любых возникающих отказов; это приводит вас к модульной конструкции. И поскольку вы знаете, что стопроцентной гарантии отсутствия сбоев не существует, вы включаете избыточность на протяжении всего проекта;
- ваша цель – частый выпуск релизов, а монолитное программное обеспечение этого не позволяет; слишком много взаимозависимых частей требует трудоемкой и сложной координации. В последние годы было убедительно доказано, что программное обеспечение, состоящее из компонентов меньшего размера, более слабосвязанных и независимо развертываемых и выпускаемых (их часто называют *микросервисами*), обеспечивает более гибкую модель выпуска;
- пользователи больше не ограничены доступом к цифровым решениям, когда они сидят за своими компьютерами. Им требуется доступ с мобильных устройств, которые они носят с собой 24 часа в сутки и семь дней в неделю. И неживые объекты, такие как датчики и контроллеры устройств, также всегда подключены к сети. Оба этих сценария приводят к приливной волне запросов и объемов данных, которые могут сильно колебаться, и поэтому требуют программного обеспечения, которое масштабируется динамически и продолжает функционировать адекватно.

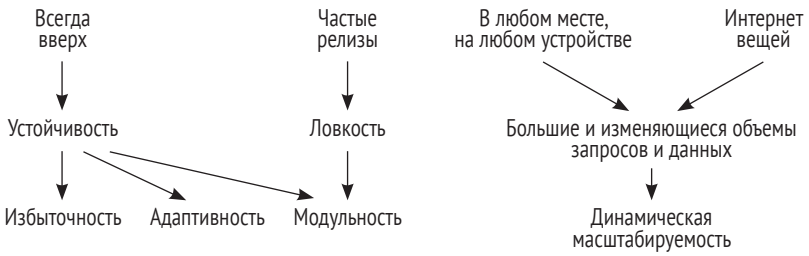


Рис. 1.4 ❖ Требования пользователей к программному обеспечению ведут разработку в направлении принципов архитектуры облачной среды

Некоторые из этих атрибутов имеют архитектурные последствия: получающееся в результате программное обеспечение состоит из развернутых независимых компонентов с избыточностью. Другие атрибуты относятся к методам управления, используемым для разработки цифровых решений: развертывание должно адаптироваться к изменяющейся инфраструктуре и к нестабильным объемам запросов. Взяв эту совокупность атрибутов как одно целое, давайте доведем данный анализ до конца. Посмотрите на рис. 1.5:

- программное обеспечение, созданное как набор независимых компонентов, развертываемое с использованием избыточности, подразумевает распространение. Если бы все ваши избыточные копии были развернуты близко друг к другу, вы бы подверглись большему риску локальных сбоев, имеющих далеко идущие последствия. Чтобы эффективно использовать имеющиеся у вас ресурсы инфраструктуры, когда вы развертываете дополнительные экземпляры приложения для обслуживания растущих объемов запросов, вы должны иметь возможность размещать их в широком диапазоне доступной инфраструктуры – возможно, даже в облачных сервисах, таких как AWS, Google Cloud Platform (GCP) и Microsoft Azure. В результате вы развертываете свои программные модули в высокой степени распределенным образом;
- адаптируемое программное обеспечение по определению «способно приспосабливаться к новым условиям», и условия, на которые я здесь ссылаюсь, – это условия инфраструктуры и набор взаимосвязанных программных модулей. Они неразрывно связаны друг с другом: по мере изменения инфраструктуры меняется программное обеспечение, и наоборот. Частые релизы означают частые изменения, а адаптация к нестабильным объемам запросов посредством масштабирования представляет собой постоянную корректировку. Понятно, что ваше программное обеспечение и среда, в которой оно работает, постоянно меняется.

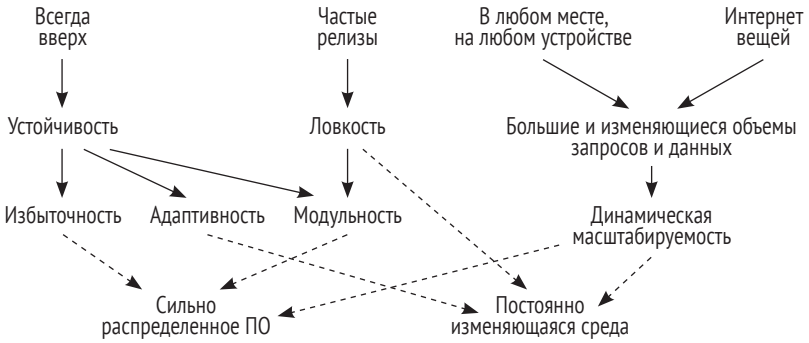


Рис. 1.5 ❖ Архитектурные и управленческие принципы приводят к основным характеристикам программного обеспечения для облачной среды: оно сильно распределено и должно работать в постоянно меняющейся среде, даже когда постоянно развивается

ОПРЕДЕЛЕНИЕ Программное обеспечение для облачной среды имеет высокую степень распространения и должно работать в постоянно меняющейся среде, и само оно постоянно меняется.

При создании программного обеспечения для облачной среды используется гораздо больше деталей (подробности заполняют страницы этого тома). Но в конечном счете все они возвращаются к этим основным характеристикам: высокая степень распределения и постоянное изменение. Это будет вашей мантрой, когда вы будете дальше читать книгу, и я буду неоднократно возвращать вас к чрезвычайному распределению и постоянным изменениям.

1.2.2. Ментальная модель программного обеспечения для облачной среды

Эдриан Кокрофт, бывший главный архитектор Netflix, а ныне вице-президент по стратегии облачной архитектуры в AWS, рассказывает о сложности управления автомобилем: будучи водителем, вы должны управлять автомобилем и перемещаться по улицам, стараясь не столкнуться с другими водителями, выполняющими те же сложные задачи[1]. Вы можете делать это только потому, что создали модель, которая позволяет вам понимать мир и управлять своим инструментом (в данном случае автомобилем) в постоянно меняющейся среде.

Большинство из нас использует ноги, чтобы контролировать скорость, и руки, чтобы установить направление, коллективно определяя нашу скорость. В попытке улучшить навигацию городские планировщики обдумывают планы улиц (Боже, помоги нам всем в Париже). А такие инструменты, как дорожные знаки и сигналы светофора в сочетании с правилами дорожного движения, дают вам основу, в которой вы можете рассуждать о путешествии, которое совершаете от начала до конца.

Написание программного обеспечения для облачной среды – также вещь сложная. В этом разделе я представляю модель, которая поможет навести порядок среди множества проблем, возникающих при написании программного обеспечения для облачной среды. Я надеюсь, что эта структура поможет вам понять ключевые концепции и методы, которые сделают вас опытным проектировщиком и разработчиком программного обеспечения для облачной среды.

Я начну с основных элементов программного обеспечения для облачной среды, которые вам наверняка знакомы. Они показаны на рис. 1.6.

Приложение реализует ключевую бизнес-логику. Здесь вы будете писать большую часть кода. Именно здесь, например, с помощью вашего кода можно будет принимать заказ клиента, проверять наличие товаров на складе и отправлять уведомление в отдел выставления счетов.

Приложение, конечно же, зависит от других компонентов, которые оно вызывает для получения информации или выполнения действий. Я называю их *сервисами*. Некоторые сервисы хранят *состояние*, например товары на складе. Другие могут быть приложениями, которые реализуют бизнес-логику другой части вашей системы, например выставление счетов клиентам.

Учитывая эти простые концепции, давайте теперь составим схему, обозначающую ПО для облачной среды, которое вы будете создавать. Посмотрите на рис. 1.7. У вас есть распределенный набор модулей, большинство из которых имеет несколько развернутых экземпляров. Видно, что большинство приложений также

[1] Послушайте, как Эдриан рассказывает об этом и других примерах сложных вещей на странице **http://mng.bz/5NzO**.

действует как сервисы, и, кроме того, некоторые сервисы явно сохраняют состояние. Стрелки показывают, как один компонент зависит от другого.

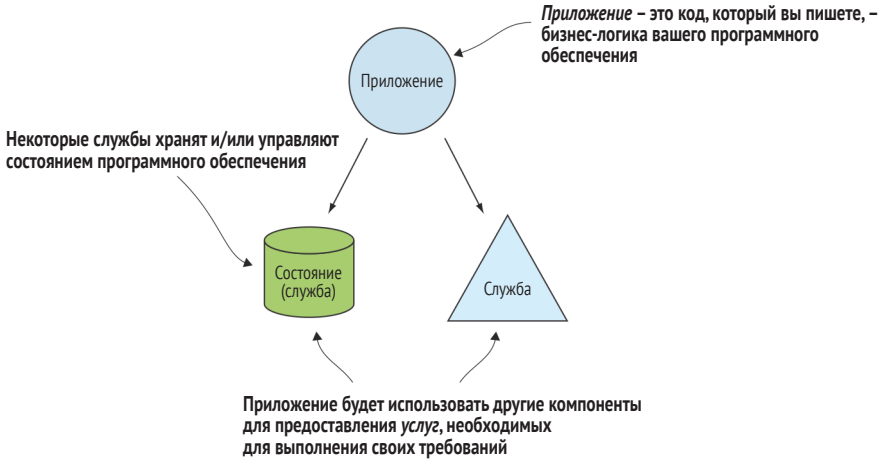


Рис. 1.6 ❖ Знакомые элементы базовой архитектуры программного обеспечения

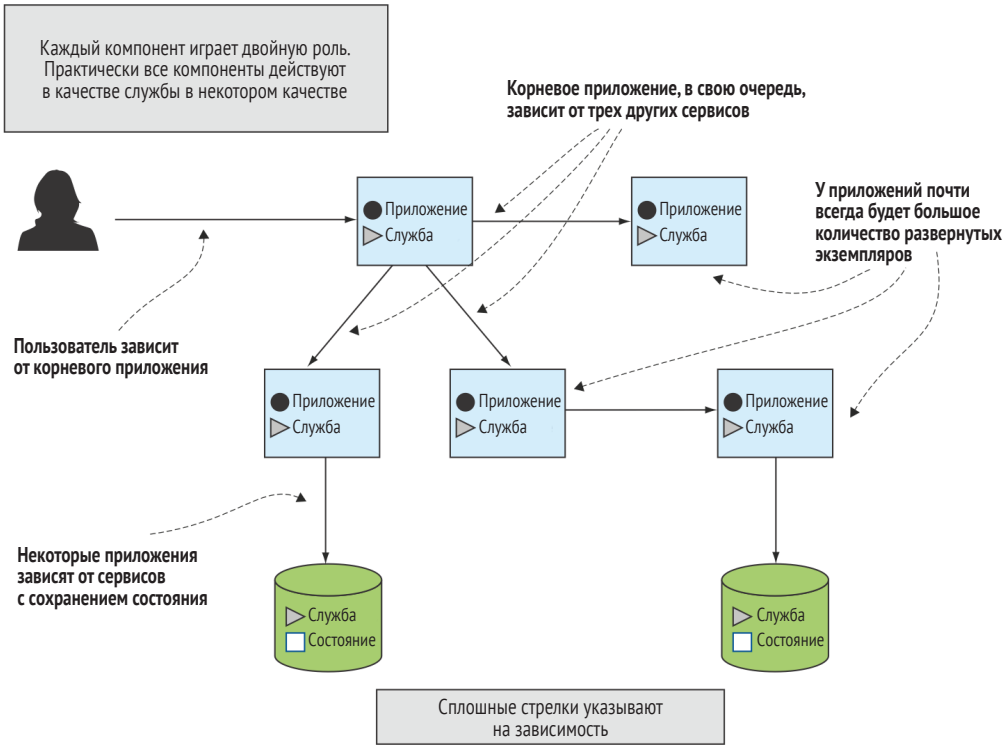


Рис. 1.7 ❖ Программное обеспечение для облачной среды использует знакомые концепции и обеспечивает экстремальное распространение с избыточностью повсюду и постоянные изменения

Данная диаграмма иллюстрирует несколько интересных моментов. Во-первых, обратите внимание, что фрагменты (прямоугольники и база данных или хранилище, значки) всегда имеют два обозначения: приложения и сервисы в прямоугольниках, а сервисы и состояние в значках хранилища. Я стала рассматривать простые понятия, показанные на рис. 1.7, как роли, которые выполняют различные компоненты вашего программного решения.

Вы заметите, что любой объект, на который направлена стрелка, указывающая, что этот компонент зависит от другого, является сервисом. Это верно – почти все это сервис. Даже к приложению, являющемуся основой схемы, идет стрелка от потребителя программного обеспечения. Приложения – конечно, это то, где вы пишете свой код. А мне особенно нравится сочетание аннотаций *сервис* и *состояние*, дающее понять, что у вас есть некоторые службы, которые не имеют состояния (службы без сохранения состояния, о которых вы наверняка слышали, помечены здесь как «приложение»), тогда как другие связаны с управлением состоянием.

И это подводит меня к определению трех частей программного обеспечения для облачной среды, изображенного на рис. 1.8:

- *приложение для облачной среды* – опять же, здесь вы будете писать код; это бизнес-логика вашего программного обеспечения. Реализация правильных шаблонов позволяет этим приложениям выступать в качестве добропорядочных граждан в составе вашего программного обеспечения; одно приложение редко бывает полноценным цифровым решением. Приложение находится на одном или другом конце стрелки (или на обоих) и, следовательно, должно реализовывать определенное поведение, чтобы иметь возможность участвовать в этих отношениях. Оно также должно быть построено таким образом, чтобы иметь возможность использовать облачные методы работы, такие как масштабирование, и позволять выполнять обновления;
- *данные в облачной среде* – это то место в вашем программном обеспечении, где находится состояние. Даже такая простая схема показывает заметное отклонение от архитектур прошлого, которые часто использовали централизованную базу данных для хранения состояния большой части программного обеспечения. Например, вы можете хранить профили пользователей, данные учетной записи, отзывы, историю заказов, информацию об оплате и многое другое в одной базе данных. Программное обеспечение для облачной среды разбивает код на множество более мелких модулей (приложений), и база данных аналогичным образом разлагается на составные части и распределяется;
- *взаимодействия в облачной среде* – программное обеспечение для облачной среды представляет собой совокупность приложений и данных, и то, как эти объекты взаимодействуют друг с другом, в конечном итоге определяет функционирование и качество цифрового решения. Из-за экстремального распределения и постоянного изменения, которое характеризует наши системы, эти взаимодействия во многих случаях значительно эволюционировали по сравнению с предыдущей архитектурой программного обеспечения, а некоторые шаблоны взаимодействия являются совершенно новыми.

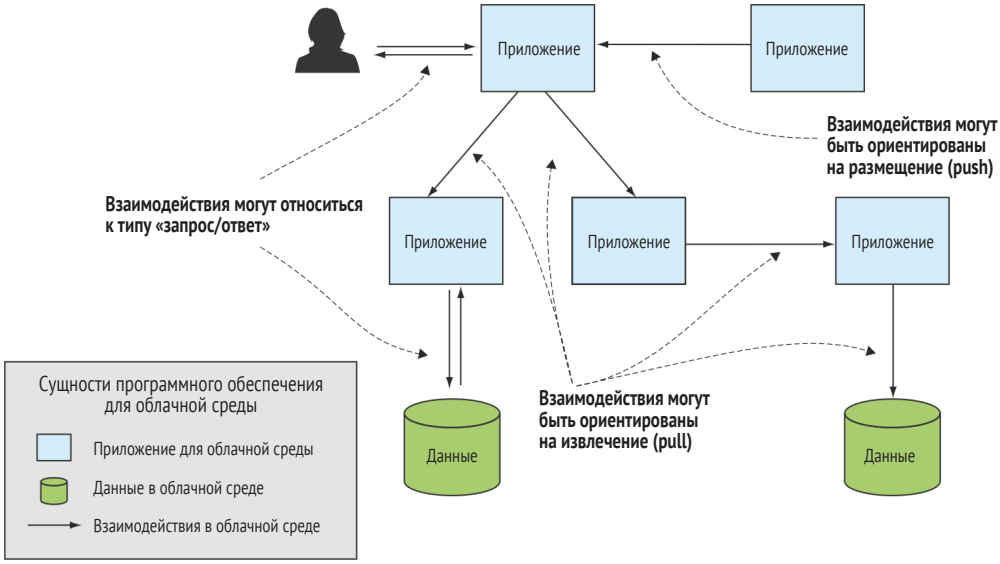


Рис. 1.8 ❖ Ключевые сущности в модели программного обеспечения для облачной среды: приложения, данные и взаимодействия

Обратите внимание, что хотя вначале я и говорила о сервисах, в конце концов, они не являются одной из трех сущностей этой ментальной модели. Во многом это связано с тем, что практически все является сервисом, как приложения, так и данные. Более того, я полагаю, что взаимодействие между сервисами даже интереснее, чем сам сервис в отдельности. Сервисы охватывают всю модель программного обеспечения для облачной среды.

Установив эту модель, давайте вернемся к современным требованиям к программному обеспечению, описанным в разделе 1.1, и рассмотрим их влияние на приложения, данные и взаимодействия вашего ПО для облачной среды.

Приложения для облачной среды

Среди проблем, связанных с такого рода приложениями, можно упомянуть следующие:

- их емкость увеличивается или уменьшается путем добавления или удаления экземпляров. Мы называем это горизонтальным масштабированием, и оно сильно отличается от моделей масштабирования, которые использовались в предыдущих архитектурах. При правильном развертывании наличие нескольких экземпляров приложения также обеспечивает уровни устойчивости в нестабильной среде;
- как только у вас появляется несколько экземпляров приложения, и даже когда каким-либо образом нарушается работа только одного экземпляра, сохранение состояния вне приложений позволяет вам наиболее легко выполнять действия по восстановлению. Вы можете просто создать новый экземпляр приложения и подключить его к любым сервисам с отслеживанием состояния, от которых он зависит;
- конфигурация приложения для облачной среды создает уникальные проблемы при развертывании множества экземпляров, а среды, в которых они

работают, постоянно меняются. Например, если у вас есть 100 экземпляров приложения, то те дни, когда вы могли перенести новую конфигурацию в известное место в файловой системе и перезапустить приложение, прошли. Добавьте к этому тот факт, что эти экземпляры могут перемещаться по всей вашей распределенной топологии. И применение таких устаревших методов к экземплярам, когда они распространяются по всей вашей распределенной топологии, было бы просто безумием;

- динамическая природа сред на базе облака требует изменений в способе управления жизненным циклом приложения (не жизненным циклом *разработки* программного обеспечения, а скорее запуском и закрытием реального приложения). Вы должны пересмотреть способы запуска, настройки, перенастройки и завершения работы приложений в этом новом контексте.

Данные в облачной среде

Итак, ваши приложения не сохраняют состояния. Но обработка состояния является не менее важной частью программного решения, и необходимость решения ваших проблем, связанных с обработкой данных, также существует в среде экстремального распространения и постоянных изменений. Поскольку у вас есть данные, которые должны сохраняться во время этих колебаний, обработка данных в облачной среде создает уникальные проблемы. Проблемы с данными в облачной среде включают в себя следующее:

- вам нужно разбить на части монолит данных. За последние несколько десятилетий компании потратили уйму времени, энергии и технологий на управление большими консолидированными моделями данных. Причиной стало то, что концепции, которые были актуальны во многих областях и, следовательно, реализованы во многих программных системах, лучше всего рассматривать централизованно как единый объект. Например, в больнице концепция пациента была актуальна во многих ситуациях, включая медицинскую помощь, выставление счетов, опросы опыта и т. д., и разработчики могли бы создавать единую модель, а часто и единую базу данных, для обработки информации о пациенте. Этот подход не работает в контексте современного программного обеспечения; он медленно развивается, становится ломким и в конечном итоге лишает внешне слабосвязанную структуру приложения своей ловкости и прочности. Необходимо создать распределенную структуру данных, подобно тому, как вы создавали распределенную структуру приложений;
- распределенная структура данных состоит из независимых, пригодных для использования баз данных (поддерживающих концепцию polyglot persistence), а также баз данных, которые могут действовать только как материализованные представления данных, где источник истины находится в другом месте. Кеширование – это ключевой шаблон и технология для разработки программного обеспечения для облачной среды;
- когда у вас есть объекты, которые существуют в нескольких базах данных, – например, «пациент», о котором я упоминала ранее, – вы должны решить, как синхронизировать информацию, общую для разных экземпляров;
- в конечном итоге в ходе рассмотрения состояния как результата ряда событий формируется ядро распределенной структуры данных. Шаблоны по-

рождения событий фиксируют события, связанные с изменением состояния, а унифицированный журнал собирает эти события и делает их доступными для членов распределения данных.

Взаимодействия в облачной среде

И наконец, когда вы соедините все части воедино, появится новый набор проблем, касающихся взаимодействий в облачной среде:

- доступ к приложению, когда оно имеет несколько экземпляров, требует определенного типа системы маршрутизации. Нужно обратиться к синхронному запросу/ответу, а также асинхронным шаблонам на основе событий;
- в сильно распределенной, постоянно меняющейся среде нужно учитывать попытки неудачного доступа. Повторная отправка запроса является существенным шаблоном в программном обеспечении для облачной среды, однако его использование может нанести ущерб системе, если им не управлять должным образом. Шаблон «Предохранитель» необходим при наличии повторной отправки запроса;
- поскольку программное обеспечение для облачной среды является составным, запрос одного пользователя обслуживается путем вызова множества связанных служб. Правильное управление программным обеспечением для облачной среды, чтобы гарантировать надлежащее взаимодействие с пользователем – задача управления композицией – каждой из служб и взаимодействие между ними. Метрики приложений и ведение журнала, то, что мы делаем десятилетиями, должно быть специализировано для новой настройки;
- одним из величайших преимуществ модульной системы является возможность более простого развития ее частей независимым образом. Но поскольку эти независимые фрагменты в конечном итоге объединяются в единое целое, протоколы, лежащие в основе взаимодействий между ними, должны соответствовать контексту облачной среды, – например система маршрутизации, которая поддерживает параллельное развертывание.

Данная книга охватывает новые и развитые шаблоны и методы для удовлетворения этих потребностей.

Давайте сделаем все это более конкретным, рассмотрев определенный пример. Это даст вам лучшее представление о проблемах, о которых я лишь кратко говорю здесь, и хорошее представление о том, куда я направляюсь.

1.2.3. Программное обеспечение для облачной среды в действии

Давайте начнем со знакомого сценария. У вас есть аккаунт в банке Волшебника. Часть времени вы общаетесь с банком, посещая местный филиал (если вы из поколения миллениалов, просто притворитесь со мной на мгновение ;-)). Вы также являетесь зарегистрированным пользователем приложения онлайн-банкинга. После получения только нежелательных звонков на домашний телефон (снова притворитесь ;-)) на протяжении большей части прошлого года или двух лет вы наконец решили отключить его. В результате вам необходимо обновить свой номер телефона в банке (и во многих других учреждениях).

Приложение для онлайн-банкинга позволяет вам редактировать свой профиль пользователя, который включает в себя ваш основной и любые резервные номера телефонов. После входа на сайт вы переходите на страницу профиля, вводите новый номер телефона и нажимаете кнопку **Отправить**. Вы получите подтверждение того, что ваши обновленные данные были сохранены, и ваш опыт взаимодействия на этом заканчивается.

Давайте посмотрим, как это могло бы выглядеть, если бы это приложение для онлайн-банкинга было спроектировано облачным способом. На рис. 1.9 показаны эти ключевые элементы:

- поскольку вы еще не вошли в систему, при доступе к приложению «*Профиль пользователя*» (1) вас перенаправят в приложение аутентификации. (2) Обратите внимание, что в каждом из этих приложений развернуто несколько экземпляров и что запросы пользователей отправляются в один из экземпляров с помощью маршрутизатора;
- как часть входа в систему приложение аутентификации создаст и сохранит новый маркер авторизации в службе с сохранением состояния (3);
- затем пользователь будет перенаправлен обратно в приложение «Профиль пользователя» с новым маркером авторизации. На этот раз маршрутизатор отправит запрос пользователя другому экземпляру приложения «Профиль пользователя». (4) (Внимание: спойлер! Так называемые sticky sessions (методы балансировки нагрузки, при которых запросы клиента передаются на один и тот же сервер группы) не подходят для программного обеспечения для облачной среды!);
- приложение «Профиль пользователя» проверит маркер авторизации, вызвав службу Auth API (5). Опять же, здесь несколько экземпляров, и запрос отправляется на один из них с помощью маршрутизатора. Напомним, что действительные маркеры хранятся в отслеживающей состояние службе Auth Token, которая доступна не только из приложения Auth, но и из любых экземпляров службы Auth API;
- поскольку экземпляры любого из этих приложений (User Profile или Auth) могут изменяться по любым причинам, должен существовать протокол для постоянного обновления маршрутизатора новыми IP-адресами (*);
- затем приложение «Профиль пользователя» отправляет нисходящий запрос в службу User API (6), чтобы получить данные профиля текущего пользователя, включая номер телефона. Приложение «Профиль пользователя», в свою очередь, отправляет запрос службе пользователя с отслеживанием состояния;
- после того как пользователь обновил свой номер телефона и нажал кнопку **Отправить**, приложение «Профиль пользователя» отправляет новые данные в *журнал событий* (7);
- в конце концов, один из экземпляров службы User API подхватит и обработает это событие (8), затем отправит запрос на запись в базу данных пользователей.

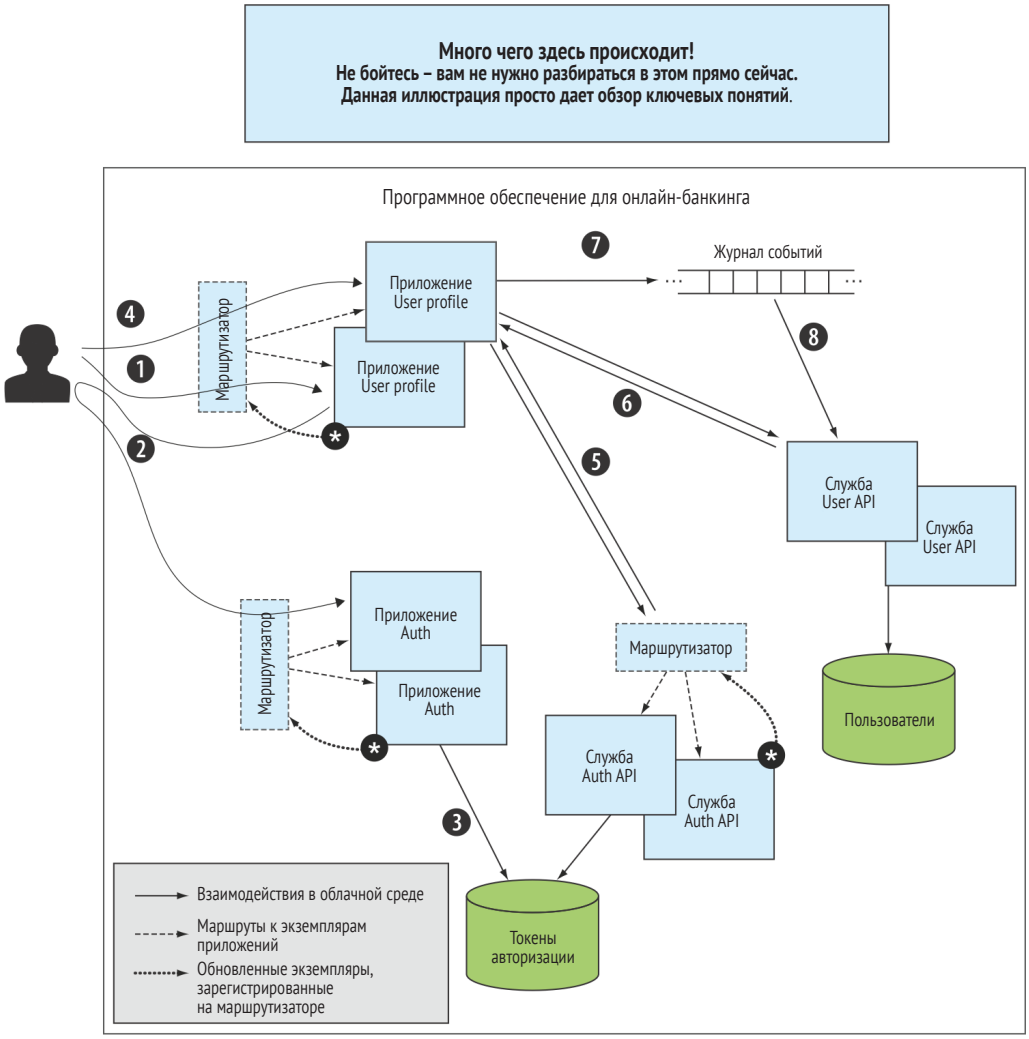


Рис. 1.9 ❖ Программное обеспечение для онлайн-банкинга представляет собой набор приложений и услуг передачи данных. Участвует множество типов протоколов взаимодействия

Да, тут и так уже много всего, но я хочу добавить еще больше.

Я не указала этого явно, но когда вы вернетесь в отделение банка и служащий проверит вашу текущую контактную информацию, вы будете ожидать, что у него высветится ваш новый номер телефона. Но программное обеспечение онлайн-банкинга и программное обеспечение служащего банка – это две разные системы. Так задумано. Это служит гибкости, устойчивости и многим другим требованиям, которые я назвала важными для современных цифровых систем. На рис. 1.10 показан данный набор продуктов.

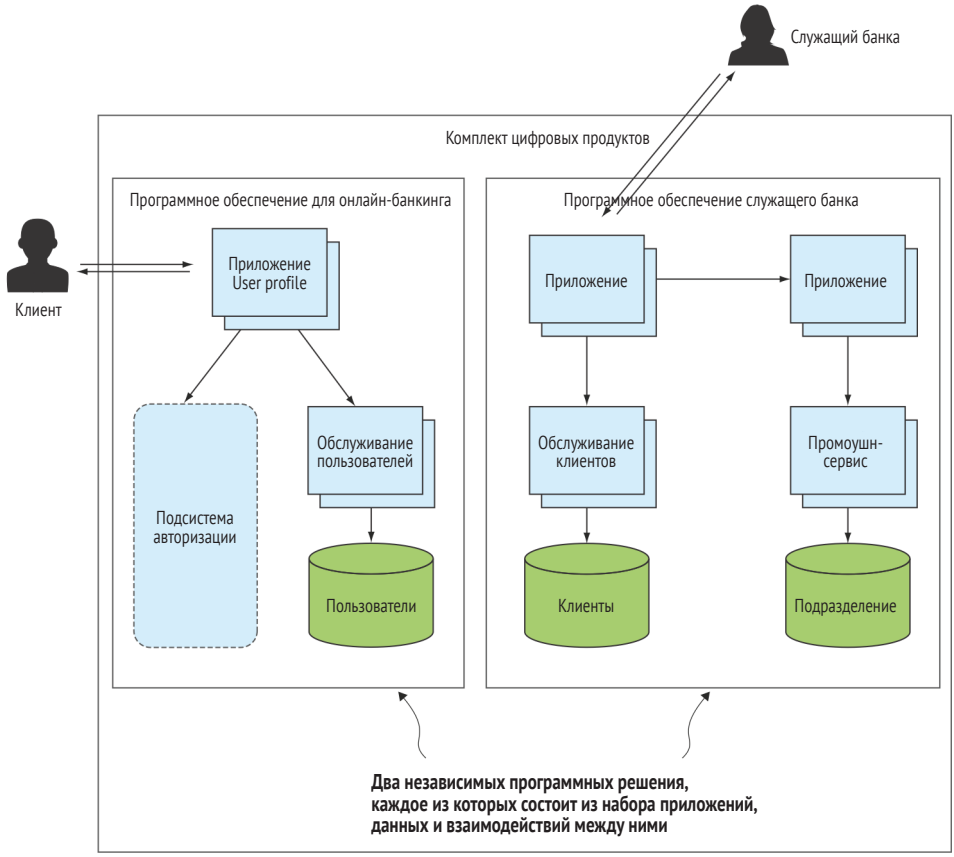


Рис. 1.10 ❖ То, что кажется пользователю единичным опытом взаимодействия с банком Волшебника, реализуется независимо разработанными и управляемыми программными ресурсами

Структура программного обеспечения банковского служащего ничем не отличается от программного обеспечения онлайн-банкинга; это набор приложений и данных для облачной среды. Но, как вы можете себе представить, каждое цифровое решение имеет дело с пользовательскими данными, или, скажем так, данными *клиентов*, и даже хранит их. При работе с программным обеспечением для облачной среды вы склонны к слабой связанности, даже когда имеете дело с данными. Это отражено в службе Users с фиксацией состояния в программном обеспечении для онлайн-банкинга и службе Customers в программном обеспечении служащего банка.

Таким образом, вопрос заключается в том, как согласовать общие значения данных в этих разнородных хранилищах. Как ваш новый номер телефона будет отображен в программном обеспечении банковского служащего?

На рис. 1.11 я добавила в нашу модель еще одну концепцию, которую я назвала «Координация распределенных данных». Здесь описание не подразумевает

каких-либо особенностей реализации. Я не предлагаю нормализованную модель данных, методы управления мастер-данными или какие-либо иные решения. На данный момент, пожалуйста, примите это как постановку проблемы. Обещаю, скоро мы разберем решения.

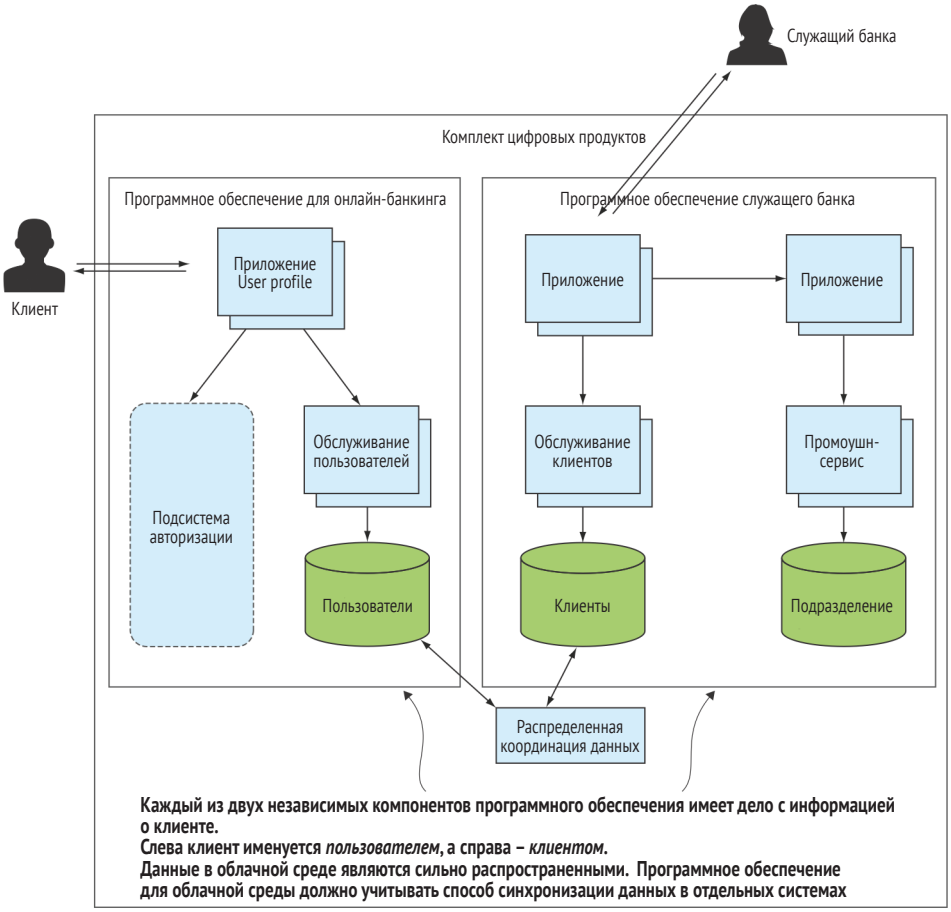


Рис. 1.11 ❖ Декомпозированная и слабо связанная структура данных требует методов для согласованного управления данными

Вот это да! Рисунки 1.9, 1.10 и 1.11 заняты, и я не ожидаю, что вы поймете в деталях все, что здесь происходит. То, что, как я надеюсь, вы узнали из этого примера, возвращает нас к ключевой теме ПО для облачной среды:

- программное обеспечение состоит из распределения множества компонентов;
- существуют протоколы, специально предназначенные для изменений, которые произошли в системе.

Мы рассмотрим все эти детали и многое другое в следующих главах.

1.3. Cloud-native и мир во всем мире

Я достаточно долго работаю в этой отрасли и была свидетелем нескольких технологических революций, обещающих решить все проблемы. Например, когда в конце 80-х годов появилось объектно-ориентированное программирование, некоторые вели себя так, как будто этот стиль программного обеспечения, по сути, писался сам. И хотя такие оптимистичные прогнозы не сбудутся, многие раскрученные технологии, без сомнения, привнесли улучшения во многие элементы программного обеспечения – простота построения и управления, надежность и многое другое.

Архитектуры программного обеспечения для облачной среды, часто именуемые микросервисами[1], сегодня в моде, но внимание: это также не приведет к миру во всем мире. И даже если бы они действительно стали доминировать (а я верю, что так и будет), их нельзя применять, где угодно. Давайте подробнее рассмотрим это чуть позже, но сначала поговорим об этом слове, *облако*.

1.3.1. Cloud и cloud-native

Повествование вокруг термина *облако* (*cloud*) может сбивать с толку. Когда я слышу, как владелец компании говорит: «Мы переходим в облако», – это часто означает, что они переносят некоторые или, может быть, даже все свои приложения в чужой центр обработки данных, такой как AWS, Azure или GCP. Эти облака предлагают тот же набор элементарных процедур, которые доступны в локальном центре обработки данных (компьютеры, хранилище и сеть), поэтому такой «переход в облако» может быть осуществлен с небольшими изменениями в программном обеспечении и методах, используемых в настоящее время локально.

Но этот подход не принесет гораздо большей устойчивости программного обеспечения, передовых методов управления или большей гибкости в процессах разработки программного обеспечения. Фактически, поскольку соглашения об уровне предоставления услуги для облачных сервисов почти всегда отличаются от тех, что предлагаются в локальных центрах обработки данных, скорее всего, вы столкнетесь со снижением эффективности во многих отношениях. Говоря кратко, переход к облаку не означает, что ваше программное обеспечение предназначено для облачной среды или что это будет демонстрировать ценности программного обеспечения cloud-native.

Как я уже говорила ранее в этой главе, новые ожидания потребителей и новые информационные контексты – те самые облачные – заставляют менять способ конструирования программного обеспечения. Принимая во внимание новые архитектурные шаблоны и методы работы, вы создаете цифровые решения, которые хорошо работают в облаке. Вы, возможно, скажете, что это программное обеспечение чувствует себя в облаке как дома. Оно абориген этой земли.

> **ПРИМЕЧАНИЕ** Понятие *cloud* (облако) – это то, *где* мы работаем. Понятие *cloud-native* используется, когда речь идет о том, как мы это делаем.

[1] Хотя я использую термин «микросервис» для обозначения архитектуры облачной среды, я не чувствую, что он охватывает две другие не менее важные сущности облачного программного обеспечения: данные и взаимодействия.

Понятие *cloud-native* используется, когда речь идет о том, как мы это делаем, означает ли это, что можно реализовать решения для облачной среды локально? Еще бы! Многие предприятия, с которыми я работаю, сначала так и делают в своих собственных центрах обработки данных. Это означает, что их локальная вычислительная инфраструктура должна поддерживать программное обеспечение и методы для облачной среды. Я рассказываю об этой инфраструктуре в главе 3.

Как бы это ни было здорово (и я надеюсь, что к тому времени, когда вы закончите читать эту книгу, вы будете думать так же), подход cloud-native подходит не для всех.

1.3.2. Что не относится к понятию «cloud-native»?

Я уверена, что вас не удивит тот факт, что не всякое программное обеспечение должно быть облачным. Когда вы будете изучать шаблоны, то увидите, что некоторые новые подходы требуют усилий, которые в противном случае могут быть не нужны. Если зависимая служба всегда находится в известном месте, которое никогда не меняется, вам не нужно будет реализовывать протокол обнаружения службы. А некоторые подходы создают новые проблемы, даже если они приносят значительную ценность. Отладка программного потока через кучу распределенных компонентов может быть сложным делом. Ниже перечислены три наиболее распространенные причины, по которым в вашей архитектуре программного обеспечения не используется облачная среда.

Во-первых, иногда программная и вычислительная инфраструктура не требуют использования облачной среды. Например, если программное обеспечение не является распределенным и редко изменяется, вы, вероятно, можете зависеть от уровня стабильности, о котором и не стоит предполагать, когда речь идет о современных крупных сетевых или мобильных приложениях. Например, код, встраиваемый во все большее количество физических устройств, таких как стиральная машина, может даже не иметь вычислительных ресурсов и ресурсов хранения для поддержки избыточности, что является ключом к этим современным архитектурам. В программном обеспечении моей рисоварки фирмы Zojirushi, которое регулирует время и температуру приготовления в зависимости от условий, о которых сообщают наружные датчики, не требуется, чтобы части приложения работали в разных процессах. Если какая-то часть программного или аппаратного обеспечения выйдет из строя, худшее, что может случиться, – это то, что мне придется оформить заказ, когда моя еда испортится.

Во-вторых, иногда общие характеристики программного обеспечения для облачной среды не подходят для рассматриваемой проблемы. Например, вы увидите, что многие новые шаблоны дают вам системы, которые в конечном итоге становятся согласованными; в вашем распределенном программном обеспечении данные, обновляемые в одной части системы, могут не сразу отражаться во всех частях системы. В конце концов, все будет совпадать, но может потребовать-

ся несколько секунд или даже минут, чтобы все стало согласованным. Иногда это нормально; например, это не такая уж большая проблема, если из-за перебоев в работе сети в рекомендациях к просмотру фильма не сразу показывается последняя оценка в пять звезд, которую поставил другой пользователь. Но иногда это не совсем нормально: банковская система не может позволить пользователю снять все средства и закрыть свой банковский счет в одном филиале, а затем разрешить дополнительное снятие средств через банкомат, поскольку обе системы на мгновение были отключены.

Возможная согласованность лежит в основе многих шаблонов для облачных сред. Это означает, что когда требуется строгая согласованность, эти конкретные шаблоны не могут использоваться.

И наконец, иногда у вас есть существующее программное обеспечение, которое не предназначено для облачной среды, и переписывать его не имеет смысла. В большинстве компаний, которым более двух десятилетий, часть ИТ-портфеля работает на мейнфреймах, и, хотите верьте, хотите нет, они могут продолжать выполнять этот код мейнфрейма еще пару десятилетий. Но это не просто код мейнфрейма. Большая часть программного обеспечения работает на множестве существующих ИТ-инфраструктур, которые отражают подходы к проектированию, предшествующие облаку. Следует переписывать код только тогда, когда это выгодно для бизнеса, и даже когда это так, вам, вероятно, придется расставлять приоритеты, обновляя различные продукты в вашем портфеле в течение нескольких лет.

1.3.3. Облачная среда нам подходит

Но это не тот случай, когда мы говорим: все или ничего. Большинство из вас пишет программы в условиях, где уже есть существующие решения. Даже если вы находитесь в завидном положении, создавая совершенно новое приложение, оно, вероятно, должно будет взаимодействовать с одной из этих существующих систем, и, как я только что отметила, большая часть уже работающего программного обеспечения вряд ли будет полностью cloud-native. Отличительной особенностью данного подхода является то, что в конечном итоге он представляет собой сочетание множества отдельных компонентов, и если некоторые из этих компонентов не воплощают самые современные шаблоны, полностью облачные компоненты по-прежнему могут взаимодействовать с ними.

Применение шаблонов для облачной среды там, где это возможно, даже если другие части вашего программного обеспечения используют более старые подходы к разработке, может принести мгновенную пользу. Например, на рис. 1.12 видно, что у нас есть несколько компонентов приложения. Служащий банка получает доступ к информации о счете через пользовательский интерфейс, который затем взаимодействует с API, стоящим перед приложением для мейнфреймов. При такой простой схеме развертывания, если сеть между службой Account API и этим приложением будет нарушена, клиент не сможет получить свои деньги.

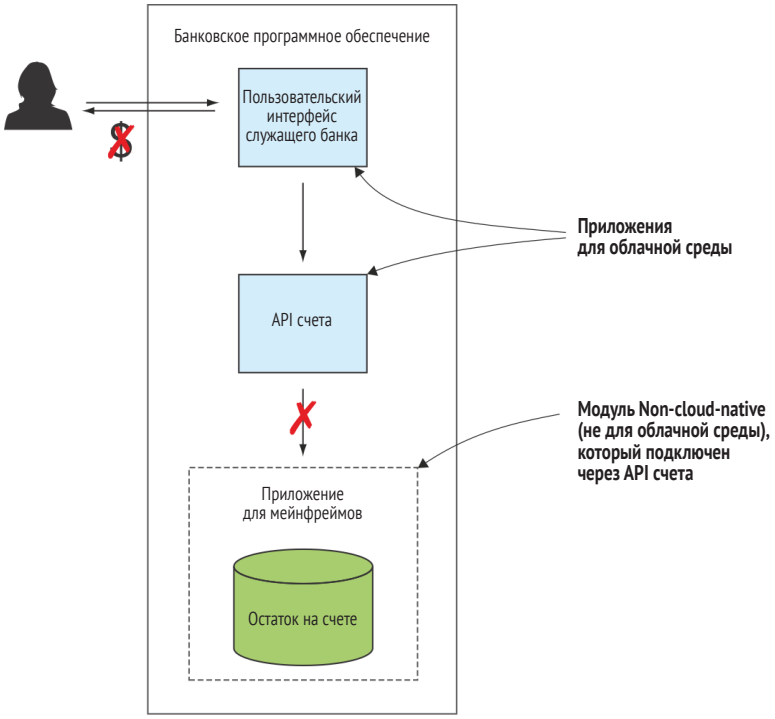


Рис. 1.12 ❖ Не рекомендуется распределять средства без доступа к источнику записи

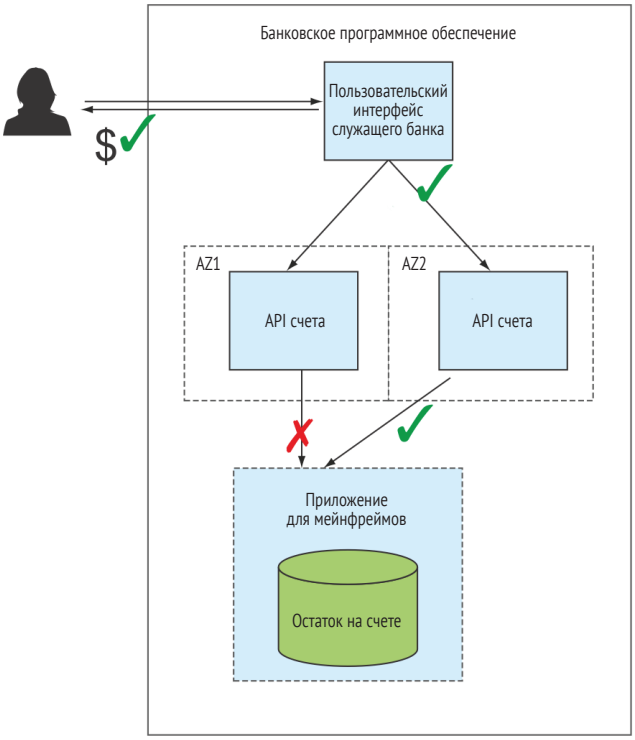
А теперь давайте применим несколько облачных шаблонов к частям этой системы. Например, если вы развернете множество экземпляров каждого микросервиса в многочисленных зонах доступности, сетевой раздел в одной зоне попрежнему разрешает доступ к данным мэйнфрейма через экземпляры сервисов, развернутые в других зонах (рис. 1.13).

Стоит также отметить, что если у вас есть устаревший код, который вы хотите реорганизовать, не нужно делать это одним махом. Netflix, например, реорганизовал все свое ориентированное на клиента цифровое решение в архитектуру для облачной среды как часть своего перехода в облако. И что в итоге? На переход ушло семь лет, но Netflix начал рефакторинг некоторых фрагментов своей монолитной клиент-серверной архитектуры в процессе, получая мгновенные преимущества[1]. Как и в предыдущем примере с банком, урок заключается в том, что даже во время миграции частичный переход в облако представляет ценность.

Создаете ли вы чистое новое приложение для облака, где применяете все новомодные шаблоны, или же извлекаете и создаете облачные части существующего монолита, вы можете рассчитывать на значительную выгоду. Хотя в то время

[1] Подробнее об этом см. статью Юрия Израилевского «Netflix: Завершение миграции в облако» (**https://media.netflix.com/en/company-blog/completing-the-netflix-cloud-migration**).

мы не использовали термин «cloud-native», отрасль начала экспериментировать с архитектурами, ориентированными на микросервисы, в начале 10-х годов этого века, и многие шаблоны были усовершенствованы в течение нескольких лет. Эта «новая» тенденция достаточно хорошо понята, и ее охват становится все более распространенным. Мы видели преимущества, которые дают эти подходы.



Развертывание нескольких экземпляров приложения в разных границах сбоев позволяет шаблонам, использующимся в приложениях для облачной среды, обеспечивать преимущества в гибридной (cloud-native и non-cloud-native) программной архитектуре

Рис. 1.13 ❖ Применение некоторых шаблонов, таких как избыточность и правильно распределенные развертывания, приносит пользу даже в том программном обеспечении, которое не полностью подходит для облачной среды

Я полагаю, что этот архитектурный стиль будет доминирующим в ближайшее десятилетие или два. Что отличает его от других причуд с меньшей стойкостью, так это то, что он возник в результате фундаментального сдвига в субстрате вычислений. Модели клиент–сервер, которые доминировали в последние 20–30 лет, впервые появились, когда вычислительная инфраструктура перешла с мейнфрейма туда, где стало доступным множество небольших компьютеров, и мы писали программное обеспечение, чтобы использовать преимущества этой вычислительной среды. Облачная среда также возникла как новый субстрат, который предлагает *программно-определяемые* вычисления, хранилища и сетевые абстракции, которые сильно распределены и постоянно меняются.

Резюме

- Приложения для облачной среды могут оставаться стабильными, даже если инфраструктура, в которой они работают, постоянно меняется или испытывает трудности.
- Основные требования к современным приложениям включают в себя обеспечение быстрой итерации и частых релизов, нулевого времени простоя и значительного увеличения объема и разнообразия подключенных устройств.
- Модель приложения для облачной среды имеет три ключевых объекта:
  - приложение;
  - данные;
  - взаимодействия.
- *Облако* – это то, где работает программное обеспечение. Cloud-native – как оно работает.
- Облачная среда не есть все или ничего. Некоторые программы, работающие в вашей компании, могут следовать множеству облачных архитектурных шаблонов, другие программы будут использовать более старую архитектуру, а иные будут гибридами (сочетать новые и старые подходы).

Глава 2

Запуск облачных приложений в рабочем окружении

О чем пойдет речь в этой главе:

- почему разработчики должны заботиться об эксплуатации;
- препятствия для успешного развертывания;
- устранение этих препятствий;
- реализация непрерывной доставки;
- влияние шаблонов облачной среды на операции.

Будучи разработчиком, вы хотите всего лишь создать программное обеспечение, которое понравится пользователям и которое обеспечит им максимальную отдачу. Когда пользователи хотят большего или у вас есть представление о чем-то, что вы хотели бы донести до них, вам бы хотелось создавать это и разрабатывать с легкостью. И вы хотите, чтобы ваше программное обеспечение хорошо работало в рабочем окружении, всегда было доступно и быстро реагировало.

К сожалению, для большинства компаний процесс развертывания программного обеспечения в рабочем окружении не является простым. Процессы, предназначенные для снижения риска и повышения эффективности, имеют непреднамеренный эффект, когда делается все наоборот, потому что они медлительны и неудобны в использовании. И после того как программное обеспечение развернуто, поддерживать его в рабочем состоянии в равной мере трудно. В результате нестабильности персонал службы поддержки постоянно находится в состоянии пожаротушения.

Даже с учетом хорошо написанного, законченного программного обеспечения по-прежнему трудно:

- развернуть это программное обеспечение;
- поддерживать его в рабочем состоянии.

Будучи разработчиком, вы можете подумать, что это не ваша проблема. Ваша задача – создать этот хорошо написанный кусок кода; развернуть его и поддерживать в рабочем окружении – работа других. Но ответственность за нынешнее хрупкое рабочее окружение лежит не на какой-либо конкретной группе или отдельном лице – «вина» лежит на системе, которая возникла из набора организационных и операционных практик, которые практически повсеместно распространены в отрасли. То, как определены команды и распределены обязанности,

способ, которым отдельные команды общаются друг с другом, и даже способ проектирования программного обеспечения – все это вносит вклад в систему, которая, откровенно говоря, терпит неудачу в отрасли.

Решение состоит в том, чтобы спроектировать новую систему, которая не рассматривает производственные операции как независимую сущность, а скорее связывает методы разработки программного обеспечения и архитектурные шаблоны с действиями по развертыванию и управлению программным обеспечением в рабочем окружении.

При разработке новой системы сначала следует понять, что является причиной самых серьезных неприятностей в текущей. После анализа препятствий, с которыми вы сталкиваетесь в настоящее время, вы можете создать новую систему, которая не только лишена проблем, но и приносит желаемые результаты, используя новые возможности, предлагаемые в облаке. Это обсуждение, которое затрагивает процессы и практики всего жизненного цикла доставки программного обеспечения, от разработки до промышленной эксплуатации. Будучи разработчиком программного обеспечения, вы играете важную роль в упрощении развертывания и управления программным обеспечением в рабочем окружении.

2.1. Препятствия

Вне всякого сомнения, обработка производственных операций – сложная и зачастую неблагодарная работа. Рабочее время обычно включает в себя работу до поздней ночи и в выходные дни, когда запланированы релизы программного обеспечения или когда происходят непредвиденные сбои. Нередко возникает значительный конфликт между группами, занимающимися разработкой приложений, и командами эксплуатации, которые обвиняют друг друга в неспособности адекватно обслуживать потребителей, используя передовой опыт взаимодействия с цифровыми технологиями.

Но, как я уже сказала, это не вина команд. Проблемы возникают из-за системы, которая непреднамеренно создает ряд барьеров на пути к успеху. Несмотря на то что каждая сложная ситуация уникальна, в ней участвует совокупность основных причин, некоторые моменты встречаются практически во всех компаниях. Они показаны на рис. 2.1, и их можно резюмировать следующим образом:

- *снежинки* – изменчивость в течение всего жизненного цикла разработки программного обеспечения (SDLC) приводит к проблемам при первоначальном развертывании, а также к недостаточной стабильности после запуска приложений. Проблема заключается в несоответствиях развертываемых программных артефактов и сред, в которых они развертываются;
- *рискованное развертывание* – ландшафт, в котором сегодня развертывается программное обеспечение, очень сложен, в нем участвует множество тесно связанных компонентов. Таким образом, существует большой риск того, что развертывание, влекущее за собой изменение в одной части этой сложной сети, вызовет колебания в других частях системы. А страх перед последствиями развертывания, в свою очередь, вызывает эффект ограничения частоты, с которой вы можете выполнять развертывание;
- *изменения – это исключение* – в течение последних нескольких десятилетий мы обычно писали и эксплуатировали программное обеспечение, ожидая,

что система, в которой оно работает, будет стабильной. Эта философия, вероятно, всегда была подозрительной. Но теперь, когда ИТ-системы являются сложными и сильно распределенными, это ожидание стабильности инфраструктуры абсолютно ошибочно[1]. В результате любая нестабильность в инфраструктуре распространяется на работающее приложение, что затрудняет поддержание его в рабочем состоянии;

- *нестабильность рабочего окружения* – и, наконец, поскольку развертывание в нестабильной среде обычно сопряжено с большими трудностями, частота развертываний в рабочем окружении ограничена.

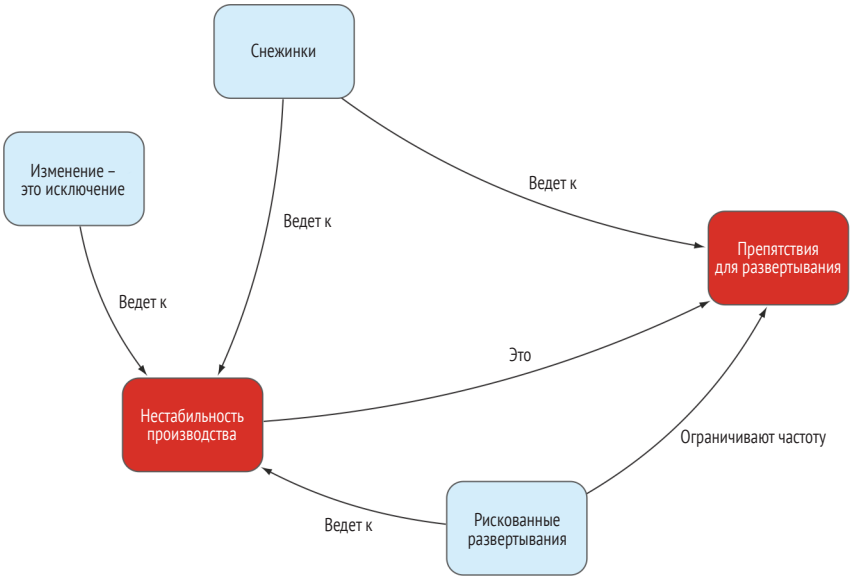


Рис. 2.1 ❖ Факторы, которые способствуют трудностям при развертывании программного обеспечения и поддержании его работоспособности в рабочем окружении

Давайте рассмотрим каждый из этих факторов подробнее.

2.1.1. Снежинки

«На моей машине это работает» – это обычная фраза, когда команда эксплуатации изо всех сил пытается выложить приложение в рабочее окружение и обращается к команде разработчиков за помощью. Я разговаривала с профессионалами из десятков крупных фирм, которые рассказывали о задержках продолжительностью в шесть, восемь или даже десять недель между моментом, когда программное обеспечение готово к выпуску, и моментом, когда оно будет доступно для пользователя. Одной из основных причин такой задержки является изменчивость в жизненном цикле разработки программного обеспечения. Эта изменчивость происходит по двум направлениям:

[1] Более подробную информацию можно найти в Википедии на странице **http://mng.bz/pgqw**.

- разница в средах;
- разница в развертываемых артефактах.

Без механизма обеспечения абсолютно одинаковой среды от разработки до тестирования, обкатки и промышленной эксплуатации программное обеспечение, работающее в одной среде, может случайно зависеть от того, чего не хватает в другой среде или оно отличается. Такое, например, происходит, когда существуют различия в пакетах, от которых зависит развертываемое программное обеспечение. Разработчик может быть строгим в отношении постоянного обновления всех версий Spring Framework, например даже в части автоматизации установок как части своих сценариев сборки. Серверы в рабочем окружении гораздо более управляемы, а обновления для Spring Framework выпускаются ежеквартально и только после тщательного аудита. Когда новое программное обеспечение попадает в эту систему, тесты больше не проходят, и, скорее всего, потребуется вернуться обратно к разработчикам, чтобы они использовали утвержденные производством зависимости.

Но различия в среде – не единственное, что замедляет развертывание. Слишком часто развертываемый артефакт также изменяется в жизненном цикле разработки программного обеспечения – даже когда специфические для среды значения жестко не прописаны в реализации (что никто из нас никогда не сделает, верно?). Файлы свойств часто содержат конфигурации, которые компилируются напрямую в развертываемый артефакт. Например, файл в формате JAR для вашего Java-приложения включает в себя файл application.properties, и если определенные параметры конфигурации выполняются непосредственно в этом файле – те, что различаются в зависимости от того, что это: разработка, тестирование или промышленная эксплуатация, – файлы JAR также должны быть разными в зависимости от того, идет ли речь о разработке, тестировании или промышленной эксплуатации. Теоретически единственное различие между каждым из этих файлов – это содержимое файлов свойств, но любая перекомпиляция или повторная упаковка развертываемого артефакта может, а часто так и происходит, непреднамеренно приводить и к другим различиям.

Эти снежинки не только негативно влияют на график первоначального развертывания; они также вносят большой вклад в операционную нестабильность. Например, допустим, у вас есть приложение, которое запущено в рабочем окружении при наличии примерно 50 000 одновременно работающих пользователей. Хотя это число обычно не колеблется слишком сильно, вам нужно пространство для роста. На этапе приемочного пользовательского тестирования (UAT) вы выполняете нагрузку с удвоенным объемом, и все тесты проходят. Вы развертываете приложение в рабочем окружении, и в течение некоторого времени все идет хорошо. Затем в субботу в 2 часа ночи вы увидите всплеск трафика. У вас внезапно появилось свыше 75 000 пользователей, и система перестала работать. Но, подождите, во время тестирования вы проверили до 100 000 одновременно работающих пользователей, так что же происходит?

Это разница в среде. Пользователи подключаются к системе через сокеты. Для соединений через сокеты требуются открытые файловые дескрипторы, а параметры конфигурации ограничивают количество файловых дескрипторов. В среде UAT значение, обнаруженное в /proc/sys/fs/file-max, составляет 200 000, а на рабочем сервере – 65 535. В ходе приемочного пользовательского тестирования вы не про-

веряли то, что вы бы увидели в рабочем окружении, из-за различий между UAT и рабочим окружением.

Дальше – хуже. После диагностики проблемы и увеличения значения в файле /proc/sys/fs/file-max все лучшие намерения персонала команды эксплуатации по документированию этого требования перекрываются чрезвычайной ситуацией; и позже, после настройки нового сервера, снова присваивается значение 65 535. Программное обеспечение устанавливается на тот сервер, и та же самая проблема в конечном счете снова поднимет свою уродливую голову.

Помните мгновение назад, когда я говорила о необходимости изменения файлов свойств между разработкой, тестированием, обкаткой и производством, и о влиянии, которое это может оказать на развертывание? Ну, допустим, у вас наконец-то все развернуто и работает, и теперь в схеме вашей инфраструктуры что-то меняется. Изменяется имя вашего сервера, URL- или IP-адрес, или же вы добавляете серверы для масштабирования. Если эти настройки среды находятся в файле свойств, нужно заново создать развертываемый артефакт, и вы рискуете получить дополнительные различия.

Хотя это может показаться экстремальным, и я надеюсь, что большинству компаний в какой-то степени удается справляться с этим хаосом, элементы генерации снежинок сохраняются во всех, кроме самых передовых, ИТ-отделах. Индивидуальные среды и пакеты развертывания явно вносят неопределенность в систему, но принятие того, что развертывание будет рискованным, само по себе является первейшей проблемой.

2.1.2. Рискованное развертывание

Когда планируется выпуск программного обеспечения в вашей компании? В нерабочее время, возможно, в 2 часа ночи в субботу? Такая практика является обычным делом вследствие одного простого факта: развертывание обычно сопряжено с опасностью. Для развертывания не является чем-то необычным, когда требуется простой во время обновления или простой возникает неожиданно. Простой – дорогое удовольствие. Если ваши клиенты не смогут заказать пиццу в режиме онлайн, они, скорее всего, обратятся к конкуренту, что приведет к прямой потере доходов.

В ответ на дорогостоящие простои компании внедрили множество инструментов и процессов, предназначенных для снижения рисков, связанных с выпуском программного обеспечения. В основе большинства этих усилий лежит идея о том, что необходимо выполнить целую кучу предварительных работ, чтобы свести к минимуму вероятность неудачи. За несколько месяцев до запланированного развертывания мы начинаем еженедельные встречи, чтобы спланировать «продвижение в верхнюю среду», и одобрения контроля изменений служат последней защитой от непредвиденных ситуаций в рабочем окружении. Возможно, самая высокая ставка с точки зрения персонала и ресурсов инфраструктуры – это процесс тестирования, который зависит от проведения пробных запусков в «точной копии рабочего окружения». В принципе, ни одна из этих идей не звучит безумно, но на практике эти упражнения в конечном итоге накладывают значительное бремя на сам процесс развертывания. Давайте рассмотрим один из этих методов более подробно в качестве примера: запуск тестовых развертываний в точной копии рабочего окружения.

Создание такой тестовой среды связано с большими затратами. Для начала требуется вдвое больше оборудования; добавьте к этому двойное количество программного обеспечения – и только капитальные затраты вырастут вдвое. Кроме того, существуют трудозатраты на поддержание соответствия тестовой среды с рабочим окружением, что осложняется множеством требований, таких как необходимость очистки производственных данных от информации, позволяющей установить личность, при генерации данных тестирования.

После создания тестовой среды необходимо тщательно организовать доступ к ней десяткам или даже сотням команд, которые хотят протестировать свое программное обеспечение, прежде чем приступить к выпуску. На первый взгляд может показаться, что это вопрос планирования, но количество комбинаций различных команд и систем быстро делает эту проблему неразрешимой.

Рассмотрим простой случай, когда у вас есть два приложения: система терминалов, которая принимает платежи, и приложение для обработки специальных заказов (SO), которое позволяет клиенту размещать заказ и оплачивать его с помощью системы терминалов. Каждая команда готова выпустить новую версию своего приложения, и им нужно выполнить предварительный тест. Как следует координировать деятельность двух этих команд? Один из вариантов – тестировать приложения по одному, и хотя последовательное выполнение тестов увеличит сроки, этот процесс будет относительно гибким, если с каждым из тестов все будет в порядке.

На рис. 2.2 показаны следующие два шага. Во-первых, четвертая версия приложения SO тестируется с первой версией (старой версией) системы терминалов. Если тест прошел успешно, четвертая версия приложения SO развертывается в рабочем окружении. И при тестировании, и в рабочем окружении теперь используется четвертая версия, и там, и там по-прежнему работает первая версия системы терминалов. *Тестовая* среда – это точная копия *рабочего окружения*. Теперь вы можете протестировать вторую версию системы терминалов, а когда все тесты будут пройдены, то сможете отправить эту версию в рабочее окружение. Оба обновления приложения завершены, при этом обе среды совпадают.

Но что произойдет, если тесты потерпят неудачу при обновлении системы SO? Очевидно, что у вас нет возможности развернуть новую версию в рабочем окружении. Но что теперь делать в тестовой среде? Вернетесь ли вы к третьей версии (что требует времени), даже если система терминалов не зависит от нее? Было ли это проблемой упорядочения, когда SO ожидала, что система терминалов уже будет работать на второй версии, прежде чем начнется тестирование? Сколько понадобится времени, прежде чем SO сможет вернуться в очередь для использования тестовой среды?

На рис. 2.3 показана пара альтернатив, которые быстро усложняются даже в этом игрушечном сценарии. В реальных условиях такая ситуация становится неразрешимой.

Моя цель состоит не в том, чтобы решить эту проблему, а в том, чтобы продемонстрировать, что даже очень упрощенный сценарий может быстро стать чрезвычайно сложным. Я уверена, что вы можете себе представить, что когда вы добавляете дополнительные приложения и/или пытаетесь параллельно тестировать новые версии нескольких приложений, процесс становится полностью неуправляемым. Среда, разработанная для того, чтобы гарантировать, что при развертывании программного обеспечения в рабочем окружении все будет работать хорошо, становится серьезной проблемой, и команды оказываются между двух

огней: им необходимо как можно быстрее доставить готовое программное обеспечение потребителю и сделать это с полной уверенностью. В конце концов, невозможно точно протестировать сценарии, с которыми вы столкнетесь в рабочем окружении, и развертывание остается рискованным делом.

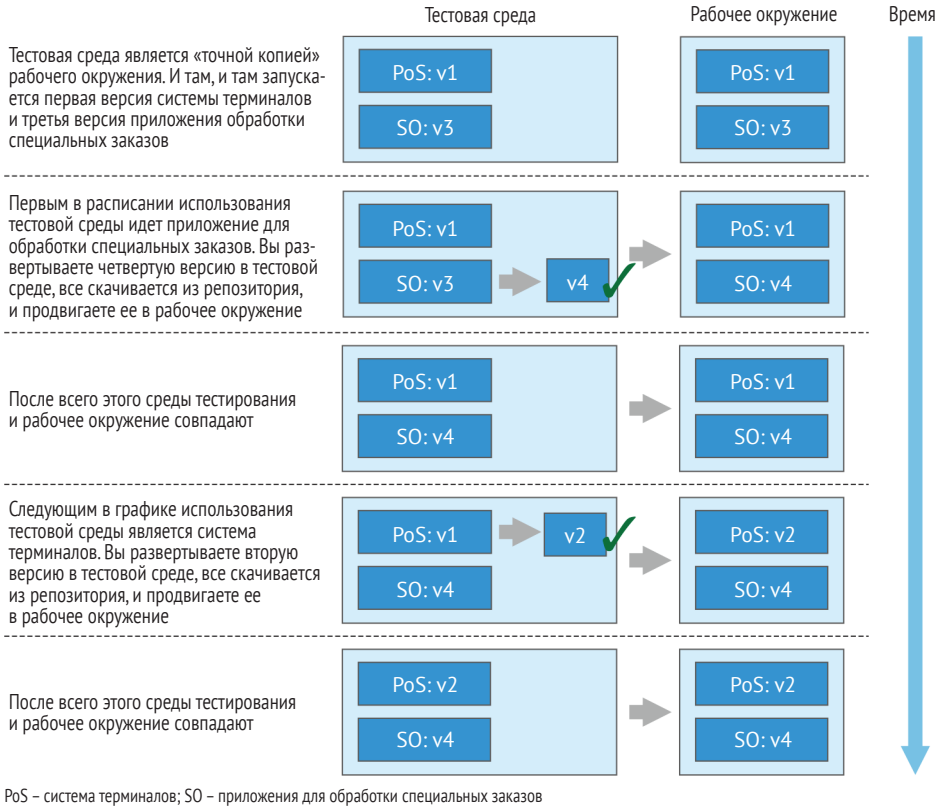


Рис. 2.2 ❖ Последовательное тестирование двух приложений не вызывает трудностей, когда все тесты успешны

У большинства компаний есть периоды времени, когда новые развертывания подобного рода запрещены. Для медицинских страховых компаний это период открытого набора. В электронной коммерции в США это месяц между Днем благодарения и Рождеством. Этот период времени также является священным для авиационной отрасли. Риски, которые сохраняются, несмотря на усилия по их минимизации, затрудняют развертывание программного обеспечения.

И из-за этой трудности программное обеспечение, работающее в рабочем окружении прямо сейчас, вероятно, останется там на некоторое время. Возможно, нам хорошо известно об ошибках или уязвимостях в приложениях и в системах, которые влияют на наш опыт работы с клиентами и деловые нужды, но мы должны бездействовать, пока не сможем организовать следующий релиз. Например, если у приложения выявлена утечка памяти, вызывающая периодические сбои, мы можем предварительно перезагружать это приложение через регулярные промежутки времени, чтобы избежать чрезвычайной ситуации. Но повышенная нагрузка на

это приложение может вызвать исключение типа Out of memory Exception раньше, чем ожидалось, и неожиданный сбой приведет к следующей аварийной ситуации.

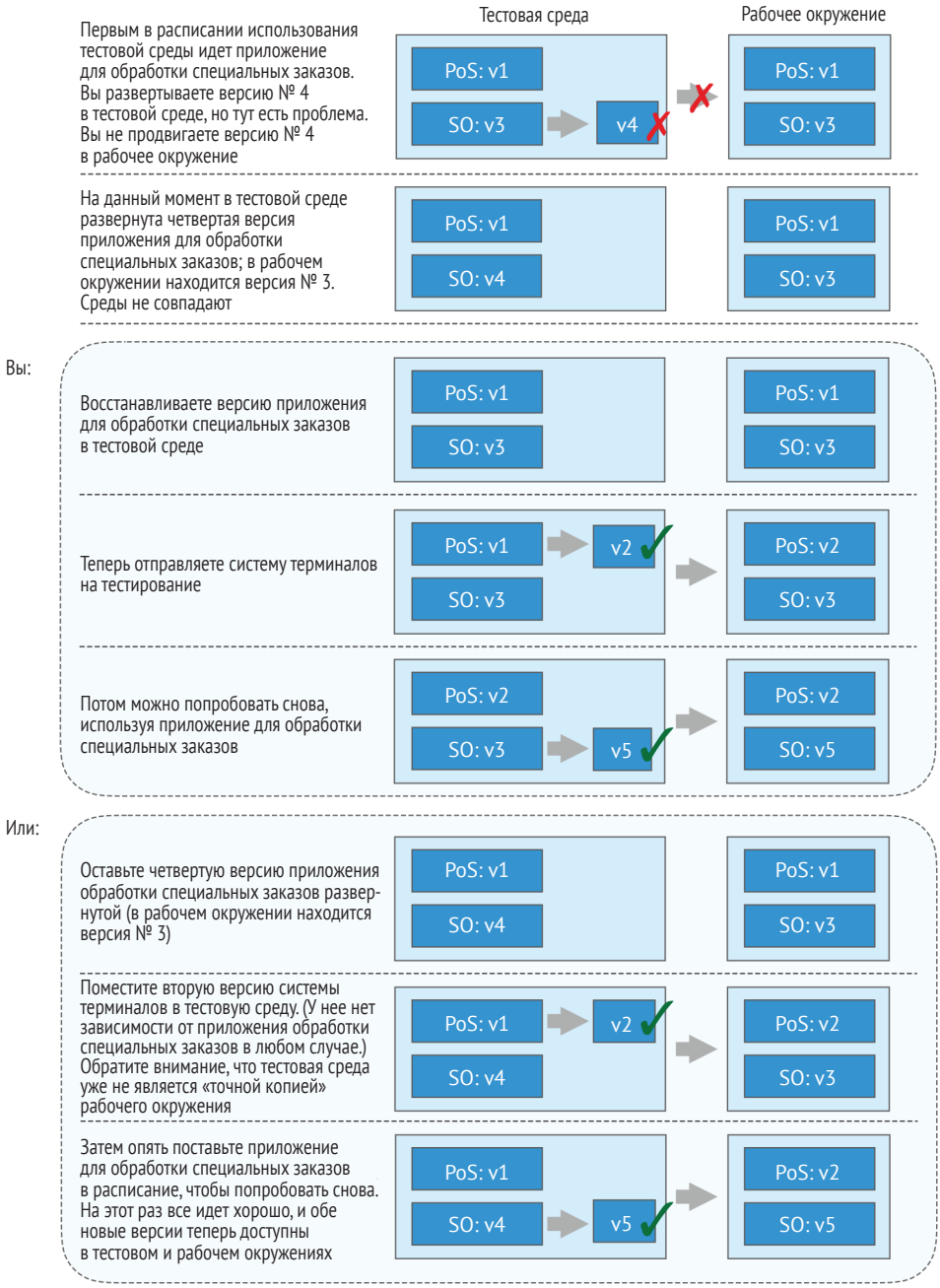


Рис. 2.3 ❖ Неудачный тест сразу же усложняет процесс экспериментального тестирования

Наконец, менее частые релизы приводят к увеличению размеров партии; развертывание приносит с собой много изменений, равно как и множество связей с другими частями системы. Все было прекрасно налажено, и это имеет интуитивный смысл, что развертывание, которое затрагивает множество других систем, с большей вероятностью приведет к чему-то неожиданному. Рискованные размещения оказывают непосредственное влияние на стабильность работы.

2.1.3. Изменение – это исключение

За эти годы у меня были десятки бесед с ИТ-директорами и их сотрудниками, которые выразили желание создавать системы, обеспечивающие дифференцированную ценность для их бизнеса и их клиентов, но вместо этого они постоянно сталкиваются с чрезвычайными ситуациями, которые отвлекают их внимание от этой инновационной деятельности. Я считаю, что причина, по которой эти люди находятся в постоянном состоянии пожарной тревоги, состоит в преобладающем складе ума этих давно устоявшихся ИТ-компаний: изменение – это исключение.

Большинство компаний осознало важность вовлечения разработчиков в начальное развертывание. Во время новых развертываний существует значительная доля неопределенности, и крайне важно привлечь команду, которая понимает, что реализация – это существенно. Но в какой-то момент ответственность за поддержание системы в рабочем состоянии полностью возлагается на команду эксплуатации, и информация о том, как справиться с задачей, предоставляется им в виде документации, в которой детально изложены возможные сценарии сбоев и их решение, и хотя в принципе это звучит хорошо, если поразмыслить, то можно предположить, что сценарии сбоев известны. Но в большинстве случаев это не так!

Команда разработчиков, отстраняющаяся от текущих операций, когда только что развернутое приложение было стабильным в течение заданного периода времени, тонко намекает на философию, согласно которой в какой-то момент времени наступает конец изменений – и с этого момента ситуация будет стабильной. Когда происходит нечто неожиданное, никто не знает, что делать дальше. Когда пресловутые постоянные изменения сохранятся, и я уже установила, что в облаке так и будет, системы продолжат испытывать нестабильность.

2.1.4. Нестабильность рабочего окружения

Все факторы, которые я рассматривала до сих пор, несомненно, мешают программному обеспечению работать нормально, но сама нестабильность рабочего окружения еще больше усложняет развертывание. Развертывание в и без того нестабильной среде неуместно; в большинстве компаний рискованное развертывание остается одной из основных причин поломки системы. Достаточно стабильная среда является предпосылкой для новых развертываний.

Но когда большая часть времени тратится на борьбу с пожарами, у нас остается мало возможностей для развертывания. Приведение тех редких моментов, когда производственные системы стабильны, в соответствии со сроками завершения сложных циклов тестирования, о которых я говорила ранее, и окна возможностей становятся еще меньше. Это порочный круг.

Как вы видите, написание программного обеспечения – это лишь начальный этап доставки цифровых технологий вашим клиентам. Обработка снежинок, позволение развертываниям становиться рискованными и рассматривать измене-

ния как исключение – все это усложняет задачу по запуску этого программного обеспечения в рабочем окружении. Дальнейшее понимание того, как эти факторы сегодня негативно влияют на работу, приходит в ходе изучения хорошо функционирующих организаций – компаний, рожденных в облаке. Когда вы применяете методы и принципы, как это делают они, вы разрабатываете систему, которая оптимизирует весь жизненный цикл поставки программного обеспечения, от разработки до бесперебойной работы.

2.2. Стимулирующие факторы

Компании нового поколения, которые достигли совершеннолетия после рубежа веков, придумали, как улучшить ситуацию. Google был великим новатором и вместе с другими интернет-гигантами разработал новые способы работы. Имея в своем распоряжении примерно 2 млн серверов, работающих в центрах обработки данных по всему миру, Google никак не смог бы управлять ими, используя методы, которые я только что описала. Существует другой способ.

На рис. 2.4 представлен эскиз системы, которая почти противоположна плохой системе, описанной мною в предыдущем разделе. Цели заключаются в следующем:

- простые и частые выпуски в производство;
- стабильность и предсказуемость.

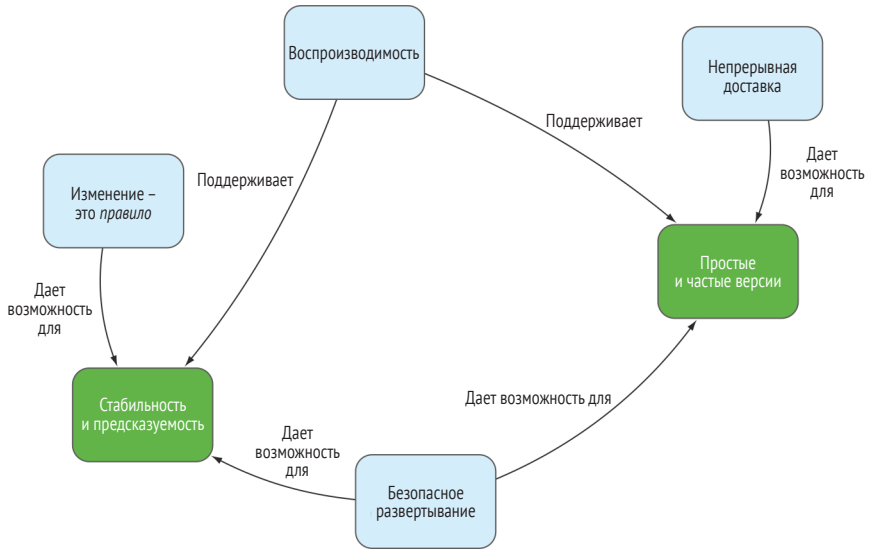


Рис. 2.4 ❖ Явное внимание к этим четырем факторам способствует разработке эффективной, предсказуемой и стабильной системы

Вы уже знакомы с обратной стороной некоторых факторов:

- в то время как снежинки ранее способствовали медлительности и нестабильности, повторяемость подтверждает обратное;

- в то время как рискованные развертывания способствуют нестабильности рабочего окружения и появлению проблем, возможность безопасного развертывания обеспечивает гибкость и стабильность;
- замена методов и конструкций программного обеспечения, которые зависят от неизменной среды, на те, что ожидают постоянных изменений, радикально сокращает время, затрачиваемое на борьбу с чрезвычайными ситуациями.

Но, глядя на рис. 2.4, вы заметите новый объект с надписью «Непрерывная доставка». Компании, которые достигли наибольшего успеха с помощью новой модели ИТ-операций, изменили дизайн своих процессов жизненного цикла разработки программного обеспечения, используя непрерывную доставку в качестве основного драйвера.

Это заметно сказывается на легкости, с которой может происходить развертывание, и это преимущества для всей системы.

В данном разделе я сначала объясню, что такое непрерывное развертывание, как базовые изменения в жизненном цикле разработки программного обеспечения активируют его и каковы положительные результаты. Затем я вернусь к трем другим ключевым факторам и подробно опишу их основные атрибуты и преимущества.

2.2.1. Непрерывная доставка

Amazon – возможно, самый экстремальный пример частых релизов. Говорят, что для **www.amazon.com** код в эксплуатацию вводится в среднем ежесекудно и ежедневно. Вы можете усомниться в необходимости таких частых релизов для своей компании, и конечно же, вероятно, вам не нужно выпускать программное обеспечение 86 000 раз в день. Но частые релизы стимулируют деловую гибкость и поддержку, а это – показатели сильной компании.

Вначале позвольте мне определить *непрерывную доставку*, указав на то, чем она не является. Непрерывная доставка не означает, что каждое изменение кода развертывается в среде эксплуатации. Скорее, это означает, что самая свежая версия программного обеспечения может быть *развернута* в любое время. Команда разработчиков постоянно добавляет новые возможности к реализации, но с каждым дополнением они гарантируют, что программное обеспечение готово к отправке, выполняя полный (автоматизированный!) цикл тестирования и упаковывая код для релиза.

На рис. 2.5 показан этот цикл. Обратите внимание, что в циклах нет этапа «упаковки» после этапа «тестирования». Вместо этого оборудование для упаковки и развертывания встроено непосредственно в процесс разработки и тестирования.

Рис. 2.5 ❖ Не каждый цикл разработки/тестирования приводит к отправке; и напротив, каждый цикл приводит к созданию программного обеспечения, которое готово к отправке. Доставка становится бизнес-решением

Сравните это с более традиционной практикой разработки программного обеспечения, изображенной на рис. 2.6. Гораздо более длительный цикл запускается с большим количеством программных разработок, что добавляет очень много функций к реализации. После добавления заранее определенного набора новых возможностей расширенный этап тестирования завершается, и программное обеспечение готовится к выпуску.

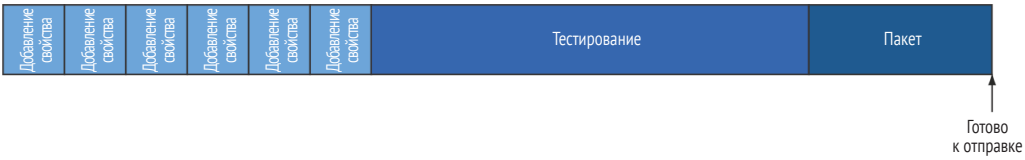


Рис. 2.6 ❖ В традиционном жизненном цикле доставки программного обеспечения большая часть работы по разработке и длительный цикл тестирования планируются заранее, прежде чем создавать артефакты, которые затем можно выпустить в промышленную эксплуатацию

Предположим, что промежутки времени, показанные на рис. 2.5 и 2.6, одинаковы и что начало каждого процесса находится слева, а точка «Готово к отправке» находится в крайнем правом положении.

Если вы посмотрите только на этот самый правый момент времени, то не увидите большой разницы в результатах; примерно одни и те же функции будут доставлены примерно в одно и то же время. Но если вы начнете копаться в деталях, то увидите значительные различия.

Во-первых, при первом подходе решение о том, когда произойдет следующий выпуск программного обеспечения, может зависеть от компании, а не от сложного, непредсказуемого процесса разработки программного обеспечения. Например, допустим, вы узнали, что конкурент планирует выпустить продукт, похожий на ваш, в течение двух недель. В результате компания принимает решение, что вы должны незамедлительно выпустить свой продукт. Компания говорит: «Давайте выпустим сейчас!» На рис. 2.7 наложение этого момента времени на две предыдущие диаграммы демонстрирует резкий контраст.

Использование методологии разработки программного обеспечения, которая поддерживает непрерывную доставку, позволяет немедленно выпустить готовое к отправке программное обеспечение третьей версии (выделено курсивом). Правда, приложение пока еще не имеет всех запланированных функций, но конкурентное преимущество того, чтобы быть первым на рынке, предлагая продукт, обладающий некоторыми из этих функций, может быть значительным. Глядя на нижнюю половину рисунка, видно, что компании не повезло. Этот ИТ-процесс является скорее блокирующим фактором, нежели стимулирующим, и продукт конкурента выйдет на рынок первым!

Этот процесс также дает еще один важный результат. Когда готовые к отправке версии часто становятся доступными для клиентов, это дает вам возможность собрать отзывы, используемые для улучшения последующих версий продукта. Следует обдумать использование отзывов, собранных после выпуска более ранних версий, для исправления ложных предположений или даже полного измене-

ния курса при выпуске последующих версий. Я была свидетелем того, как многие Scrum-проекты потерпели неудачу, потому что они строго придерживаются планов, определенных в начале проекта, не позволяя результатам предыдущих версий изменить эти планы.

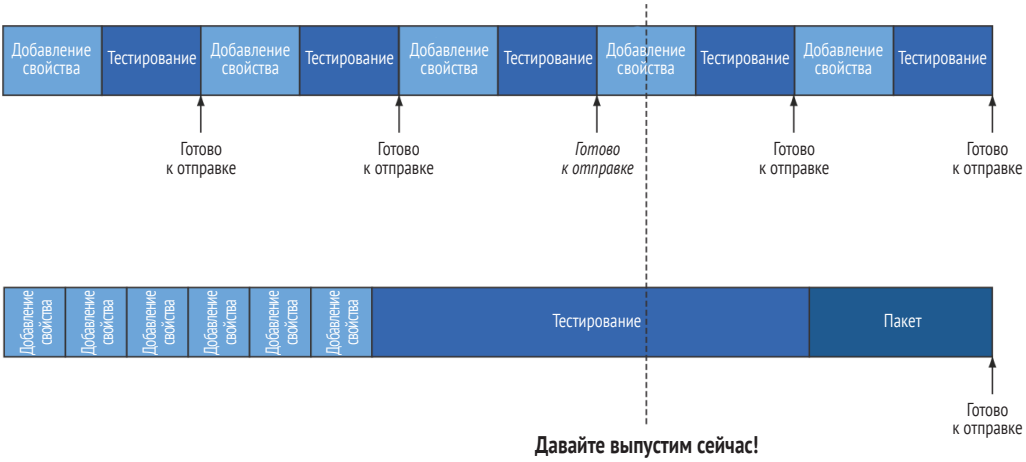


Рис. 2.7 ❖ Непрерывная доставка связана с тем, что бизнес-драйверы, а не ИТ-готовность компании, могут определять, когда поставляется программное обеспечение

Наконец, давайте признаем: мы не способны оценить время, необходимое для создания программного обеспечения. Отчасти виной тому – наш врожденный оптимизм. Обычно мы рассчитываем на счастливый путь, при котором код работает как положено сразу после первой записи. (Да, если так, то мы сразу видим абсурдность этого, а?) Мы также предполагаем, что полностью сосредоточимся на поставленной задаче; мы будем сокращать код весь день, каждый день, пока не добьемся цели. И нас, вероятно, заставляют соглашаться на агрессивные графики, обусловленные потребностями рынка или другими факторами, из-за чего мы обычно будем отставать от графика еще до того, как начнем.

Непредвиденные проблемы реализации возникают всегда. Скажем, вы недооцениваете влияние задержки в сети на одну часть вашей реализации, и вместо простого обмена запросами/ответами, который вы запланировали, теперь вам нужно реализовать гораздо более сложный протокол асинхронной связи. И пока вы внедряете очередной набор функций, вы отдаляетесь от новой работы, чтобы поддерживать расширения для уже выпущенных версий программного обеспечения. И редко когда ваши растянутые цели соответствуют и без того сложному графику.

Влияние этих факторов на процесс разработки старого образца заключается в том, что вы пропускаете запланированный этап выпуска. На рис. 2.8 показан идеализированный план выпуска программного обеспечения в первом ряду. Второй ряд показывает фактическое количество времени, потраченного на разработку (дольше, чем планировалось), а в последних двух рядах показаны альтернативы того, что вы можете сделать. Один из вариантов – придерживаться за-

планированного этапа выпуска, сжимая этап тестирования, безусловно, за счет качества программного обеспечения (этап упаковки обычно нельзя сократить). Второй вариант – сохранить стандарты качества и перенести дату выпуска. Ни один из этих вариантов не радует.

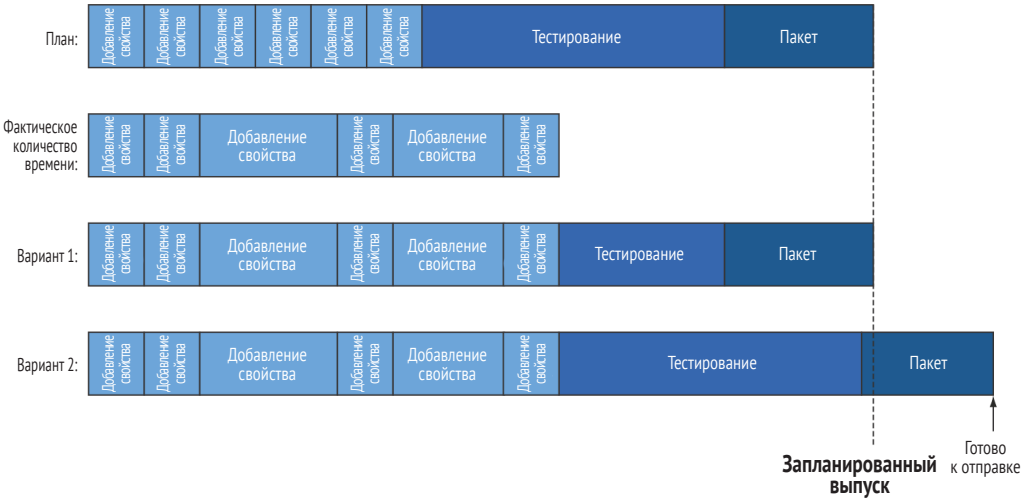


Рис. 2.8 ❖ Когда график разработки сдвигается, нужно выбирать между двумя неприятными вариантами

Сравните это с эффектами, которые «непредвиденные» задержки разработки оказывают на процесс, реализующий множество более коротких версий. На рис. 2.9 вы снова видите, что запланированный этап выпуска ожидается после шести версий. Когда фактическая реализация занимает больше времени, чем ожидалось, вы видите, что у вас есть новые возможности. Вы можете приступить к выпуску по расписанию с более ограниченным набором функций (вариант 1) либо выбрать небольшую или более длительную задержку для следующего релиза (варианты 2 и 3). Суть в том, что компании предоставляется гораздо более гибкий и приемлемый набор опций. И когда с помощью системы, которую я представляю в этом разделе, вы делаете развертывание менее рискованным и, следовательно, выполняете его чаще, то можете завершить два этих релиза в быстрой последовательности.

В итоге все длительные циклы выпуска вносят большой риск в процесс доставки цифровых продуктов клиентам. Бизнесу не хватает возможности контролировать, когда продукты поступают на рынок, и компания в целом часто находится в неловком положении, пытаясь компенсировать краткосрочное рыночное давление долгосрочными целями качества программного обеспечения и способностью развиваться.

ПРИМЕЧАНИЕ Короткие циклы освобождают систему от напряжения. Непрерывная доставка позволяет бизнес-драйверам определять, как и когда продукты поступают на рынок.

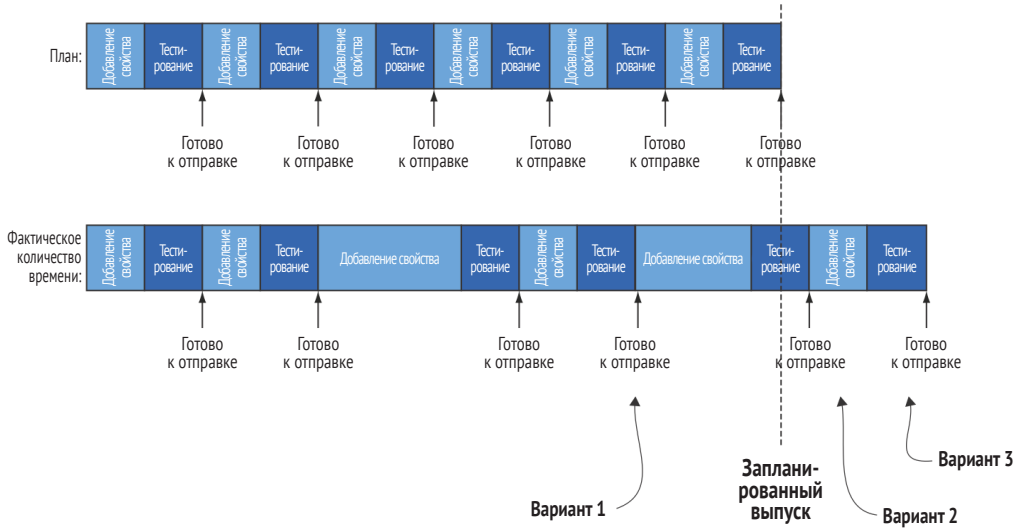


Рис. 2.9 ❖ Более короткие итерации, разработанные для непрерывной доставки, обеспечивают быстрый процесс выпуска при сохранении качества программного обеспечения

Сначала я говорила о непрерывной доставке, и относительно подробно, потому что она действительно лежит в основе новой, функциональной системы разработки программного обеспечения. Если ваша компания пока еще не использует такие методы, это то место, куда следует направить свои первоначальные усилия. Без таких изменений ваша способность изменить способ доставки программного обеспечения на рынок ограничена. И даже структура программного обеспечения, которое вы создаете, о чем и рассказывается в этой книге, связана с этими методами искусно и напрямую. Архитектура программного обеспечения – это то, о чем эта книга, и мы рассмотрим ее подробно.

Давайте вернемся к рис. 2.4 и изучим другие факторы, которые помогают нам в осуществлении наших целей: простые, частые релизы и стабильность программного обеспечения.

2.2.2. Повторяемость

В предыдущем разделе я говорила о вредном влиянии изменчивости, или, как мы это часто называем, *снежинок*, на работу информационно-технологического процесса. Они затрудняют развертывание, потому что вы должны постоянно приспосабливаться к различиям в среде, в которой вы развертываете, и в изменчивости артефактов, которые вы развертываете. Та же самая непоследовательность чрезвычайно осложняет поддержку работоспособности сразу после запуска в рабочем окружении, потому что каждая среда и фрагмент программного обеспечения подвергаются особой обработке каждый раз, когда что-то меняется. Отклонение от известной конфигурации – это постоянная угроза стабильности, когда вы не можете надежно воссоздать конфигурацию, работавшую до того, как произошел сбой.

Когда вы превращаете этот негатив в позитив в своей стимулирующей системе, ключевым понятием является повторяемость. Это аналогично этапам сборки: каждый раз, когда вы прикрепляете руль к автомобилю, вы повторяете один и тот же процесс. Если в некоторых параметрах условия совпадают (я подробнее остановлюсь на этом чуть позже) и выполняется один и тот же процесс, результат предсказуем.

Преимущества повторяемости для наших двух целей – развертывание и поддержание стабильности – огромны. Как вы видели в предыдущем разделе, итеративные циклы важны для частых релизов, и, избавляясь от изменчивости в процессе разработки/тестирования, время для предоставления новой возможности в рамках итерации сокращается. И после запуска в производство, независимо от того, реагируете вы на сбой или увеличиваете емкость для обработки больших объемов, способность абсолютно предсказуемо уничтожать развертывания избавляет систему от огромных нагрузок.

Как же достичь этой нужной нам повторяемости? Одним из преимуществ программного обеспечения является то, что его легко изменить и его можно быстро сделать гибким. Но именно это и побудило нас создавать снежинки в прошлом. Для достижения необходимой повторяемости нам нужно быть дисциплинированными. В частности, необходимо сделать следующее:

- контролировать среду, в которой вы будете развертывать программное обеспечение;
- контролировать развертываемое программное обеспечение – также известное как *развертываемый артефакт;*
- контролировать процессы развертывания.

Контроль среды

На сборочной линии вы контролируете среду, выстраивая собираемые детали и инструменты, используемые для сборки, точно таким же образом – вам не нужно искать торцевой гаечный ключ на три четверти дюйма каждый раз, когда он вам нужен, потому что он всегда находится в одном и том же месте. В программном обеспечении вы используете два основных механизма для последовательного выстраивания контекста, в котором выполняется реализация.

Сначала нужно начать со стандартизированных образов машин. При создании среды вы должны последовательно начинать с известной отправной точки. Во-вторых, изменения, примененные к этому базовому образу, чтобы установить контекст, в котором развертывается ваше программное обеспечение, *должны быть закодированы*. Например, если вы начинаете с базового образа Ubuntu, а для вашего программного обеспечения требуется комплект разработчика Java Development Kit (JDK), вы запишете сценарий установки JDK в базовый образ. Термин, часто используемый для этого понятия, – *инфраструктура в виде кода*. Когда вам нужен новый экземпляр среды, вы начинаете с базового образа и применяете сценарий, и у вас гарантированно всегда будет одна и та же среда.

После того как все изменения в окружающей среде будут установлены, они также должны контролироваться в равной степени. Если оперативный персонал регулярно подключается через протокол SSH и вносит изменения в конфигурацию, то та строгость, которую вы применили при настройке систем, ничтожна. Можно использовать многочисленные методы для обеспечения контроля после первоначального развертывания. Вы можете запретить доступ по SSH в работающую

среду, или, если вы это сделаете, автоматически переведите компьютер в автономный режим, как только кто-то использует протокол SSH. Последний вариант является полезным шаблоном, который позволяет кому-то войти в окно, чтобы исследовать проблему, но не разрешает вносить какие-либо потенциальные изменения, чтобы повлиять на работающую среду. Если необходимо внести изменения, единственный способ сделать это – обновить стандартный образ машины, а также код, который применяет к нему среду выполнения, – все это контролируется системой управления версиями или чем-то похожим.

Лица, отвечающие за создание стандартизированных образов машин и инфраструктуры в виде кода, могут быть разными, но, будучи разработчиком приложения, важно, чтобы вы использовали такую систему. Методы, которые вы применяете (или не применяете) на ранних этапах жизненного цикла разработки программного обеспечения, заметно влияют на способность компании эффективно развертывать и управлять этим программным обеспечением в рабочем окружении.

Контроль за развертываемым артефактом

Давайте на мгновение признаем очевидное: в средах всегда есть различия. В рабочем окружении ваше программное обеспечение подключается к вашей действующей клиентской базе данных, которая, например, находится по адресу **http://prod.example.com/cutomerDB**; при обкатке оно подключается к копии этой базы данных, которая была очищена от информации, позволяющей установить личность, и находится по адресу **http://staging.example.com/cleansedDB**; а во время начальной разработки может существовать фиктивная база данных, доступ к которой осуществляется по адресу **http://localhost/mockDB**. Очевидно, что учетные данные различаются в зависимости от среды. Как вы учитываете такие различия в коде, который вы создаете?

Я знаю, что вы не кодируете жестко такие строки непосредственно в своем коде (верно?). Вероятно, вы настраиваете свой код и помещаете эти значения в некий тип файла свойств. Для начала неплохо, но часто остается одна проблема: файлы свойств, а следовательно, значения параметров для различных сред часто компилируются в развертываемый артефакт. Например, в настройке Java файл application.properties нередко включается в файл формата JAR или WAR, который затем развертывается в одной из сред. В этом и заключается проблема. Когда специфичные для среды параметры компилируются, файл JAR, который вы развертываете в тестовой среде, отличается от JAR-файла, который вы развертываете в рабочем окружении; см. рис. 2.10.

Как только вы создадите разные артефакты для разных этапов жизненного цикла разработки программного обеспечения, повторяемость может быть нарушена. Дисциплина для контроля изменчивости этого программного артефакта, гарантирующая, что единственное различие в артефактах – это содержимое файлов свойств, теперь должна быть имплантирована в сам процесс сборки. К сожалению, поскольку файлы JAR разные, у вас больше нет возможности сравнивать хеши файлов, чтобы убедиться, что артефакт, который вы развернули в окружении для обкатки, точно такой же, как и тот, что вы развернули в рабочем окружении. И если что-то меняется в одной из сред, и изменяется одно из значений свойств, необходимо обновить файл свойств, что означает новый развертываемый артефакт и новое развертывание.

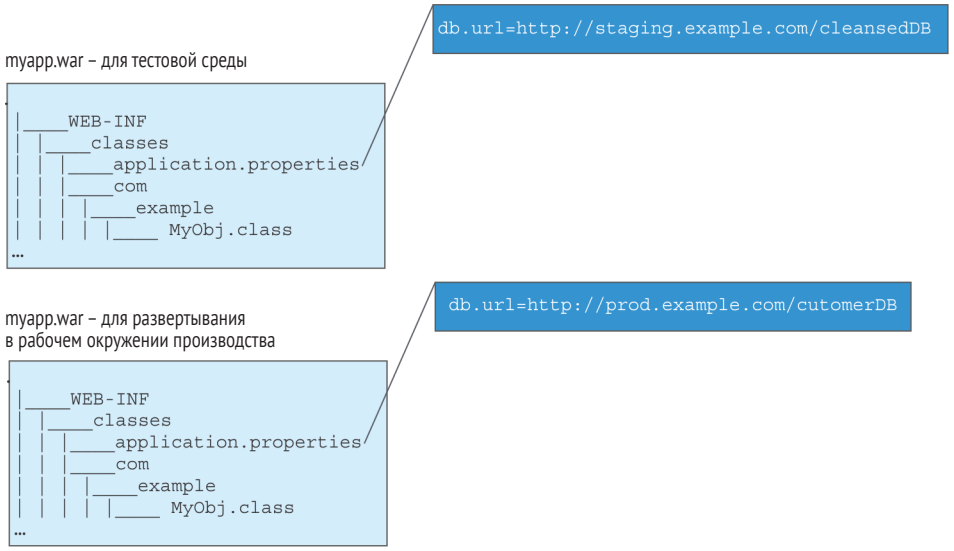


Рис. 2.10 ❖ Даже когда специфические для среды значения организованы в файлы свойств, включая файлы свойств в развертываемом артефакте, в жизненном цикле программного обеспечения у вас будут разные артефакты

Для эффективных, безопасных и повторяемых производственных операций важно, чтобы во всем жизненном цикле разработки программного обеспечения использовался один развертываемый артефакт. Файл JAR, который вы создаете и запускаете с помощью регрессионных тестов во время разработки, является *точно таким же* файлом JAR, развернутым в тестовой среде, окружении для обкатки и рабочем окружении. Чтобы это произошло, код должен быть правильно структурирован. Например, файлы свойств не содержат специфических для среды значений, но вместо этого определяют набор параметров, для которых впоследствии могут быть введены значения. Затем вы можете связать значения с этими параметрами в соответствующее время, извлекая значения из правильных источников. Вы как разработчик должны создавать реализации, которые должным образом абстрагируют изменчивость среды. Это позволяет создать один развертываемый артефакт, который можно использовать на всем жизненном цикле разработки программного обеспечения, привнося гибкость и надежность.

Контроль над процессом

Установив согласованность среды и дисциплину создания единого развертываемого артефакта, который будет использоваться в течение всего жизненного цикла разработки программного обеспечения, остается лишь обеспечить, чтобы эти фрагменты были собраны воедино контролируемым и воспроизводимым образом. На рис. 2.11 показан желаемый результат: на всех этапах жизненного цикла разработки программного обеспечения вы можете уверенно штамповать точные копии такого количества работающих модулей, сколько необходимо.

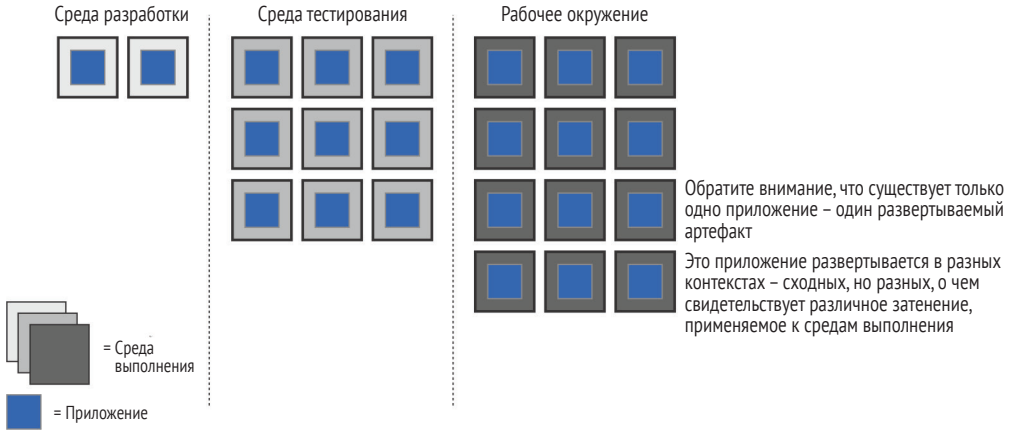


Рис. 2.11 ❖ Желаемый результат – возможность последовательно создавать приложения, работающие в стандартизированных средах. Обратите внимание, что приложение одинаково для всех сред; среда выполнения стандартизируется на этапе жизненного цикла программного обеспечения

На этом рисунке нет снежинок. Развертываемый артефакт, приложение, абсолютно одинаков во всех развертываниях и средах. Среда выполнения различается на разных этапах, но (как указано разными оттенками одного и того же серого цвета) основа одинакова. Только применяются разные конфигурации, такие как привязки базы данных. На стадии жизненного цикла все конфигурации одинаковы; у них один и тот же оттенок серого. Эти блоки для защиты от снежинок собраны из двух контролируемых объектов, о которых я говорила: стандартизированные среды выполнения и отдельные развертываемые артефакты, как показано на рис. 2.12.

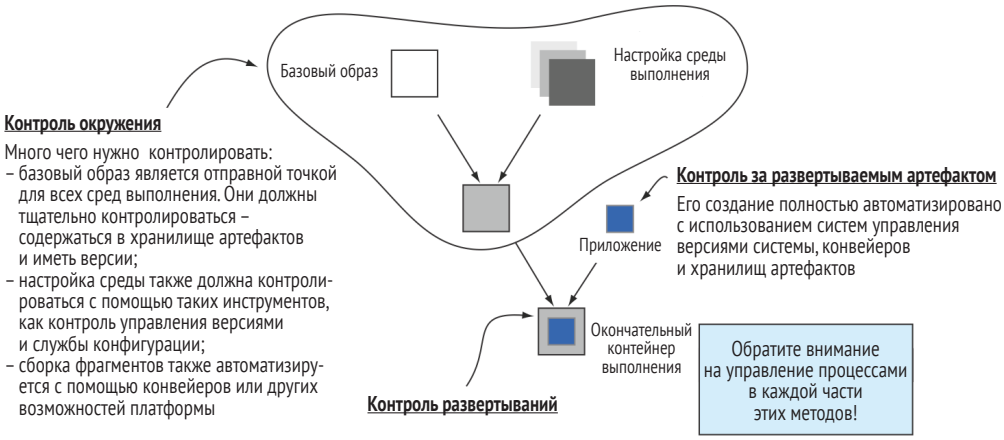


Рис. 2.12 ❖ Сборка стандартизированных базовых образов, управляемой настройки среды и отдельных развертываемых артефактов автоматизирована

Под поверхностью этого простого рисунка есть много чего. Из чего состоит хороший базовый образ и как сделать его доступным для разработчиков и операторов? Каков источник конфигурации среды и когда он вводится в контекст приложения? Когда именно приложение «инсталлируется» в контекст времени выполнения? Я буду давать ответы на эти и многие другие вопросы на протяжении книги, но на данном этапе моя главная мысль такова: единственный способ собрать все части воедино таким способом, который обеспечивал бы последовательность, – это автоматизация.

Хотя использование инструментов и методов непрерывной интеграции довольно широко распространено на этапе разработки программного обеспечения (например, конвейер сборки компилирует код, загруженный в репозиторий, и выполняет тесты), его использование для управления всем жизненным циклом разработки программного обеспечения не так широко распространено. Но автоматизация должна пройти весь путь от загрузки кода до развертывания, в тестовой среде и в рабочем окружении.

И когда я говорю, что все автоматизировано, я имею в виду *все*. Даже если вы не несете ответственности за создание различных фрагментов, сборка должна контролироваться таким образом. Например, пользователи популярной облачной платформы Pivotal Cloud Foundry используют API для скачивания новых «стволовых клеток»[1], базовых образов, в которые развертываются приложения, с сайта дистрибутивов, и используют конвейеры для завершения сборки среды выполнения и артефакта приложения. Еще один конвейер выполняет окончательное развертывание в рабочем окружении. На самом деле когда развертывания в рабочем окружении также осуществляются через конвейеры, серверы напрямую не касаются людей, что порадует вашего начальника службы безопасности (и другой персонал, связанный с обеспечением контроля).

Но если вы полностью автоматизировали процесс развертывания, как вы обеспечите безопасность этих развертываний? Это еще одна область, которая требует новой философии.

2.2.3. Безопасное развертывание

Ранее я говорила о рискованном развертывании и о том, что наиболее распространенный механизм, который используют компании в качестве попытки контролировать риск, заключается в создании обширных и дорогостоящих сред тестирования со сложными и медленными процессами для управления ими. Сначала вы можете подумать, что альтернативы нет, потому что единственный способ узнать, что что-то работает при развертывании в рабочем окружении, – сначала протестировать это. Но я полагаю, что это скорее симптом того, что, по словам Грейс Хоппер, было самой опасной фразой: «Мы всегда так делали».

Компании по разработке программного обеспечения, рожденные в эру облака, показали нам новый путь: они экспериментируют в рабочем окружении. О, боже! О чем это я?! Позвольте мне добавить одно слово: они *безопасно* экспериментируют в рабочем окружении.

---

[1] См. страницу с документацией по API платформы Pivotal по адресу **https://network.pivotal.io/docs/api** для получения дополнительной информации.

Для начала давайте посмотрим, что я подразумеваю под *безопасными экспериментами*, а затем посмотрим, какое влияние они оказывают на нашу цель, которая состоит в простом развертывании и стабильности.

Когда артисты на трапеции отпускают одно кольцо, крутятся в воздухе и хватают другое, они чаще всего достигают своей цели и развлекают зрителей. Вне всякого сомнения, их успех зависит от правильной подготовки и инструментария, а также для этого нужно много практиковаться. Но акробаты не дураки; они знают, что иногда что-то идет не так, поэтому используют страховочную сетку.

Когда вы экспериментируете в рабочем окружении, вы делаете это, используя нужные сетки безопасности. И эксплуатационные методы, и шаблоны проектирования программного обеспечения объединяются воедино, чтобы создать такую сетку. Добавьте надежные методики разработки программного обеспечения, такие как разработка через тестирование, и вы сможете минимизировать вероятность неудачи. Но цель состоит не в том, чтобы полностью избавиться от нее. Ожидание неудачи (а она случится) значительно снижает вероятность ее катастрофичности. Возможно, небольшая группа пользователей получит сообщение об ошибке, и им понадобится перезапуститься, но в целом система продолжит работать.

> **ПОДСКАЗКА** Вот ключевой момент: все, что связано с проектированием программного обеспечения и эксплуатационными методами, позволяет легко и быстро отложить эксперимент и вернуться в известное рабочее состояние (или перейти к следующему), когда это необходимо.

В этом состоит принципиальная разница между старым и новым мышлением. В первом случае вы много тестировали, прежде чем приступить к производственной эксплуатации, полагая, что вы справились со всеми проблемами. Когда это заблуждение оказалось неверным, вы не знали, что делать. Используя новый подход, вы планируете провал, намеренно создавая путь к отступлению, чтобы провалы не возникали. Это расширяет возможности! А влияние на вашу цель, которая состоит в более легком и быстром развертывании и стабильности, после запуска и эксплуатации очевидно и не заставит себя ждать.

Во-первых, если вы устраняете сложный и трудоемкий процесс тестирования, который я описывала в разделе 2.1.2, и вместо этого сразу переходите к производству после базового интеграционного тестирования, цикл значительно сокращается, и очевидно, что релизы могут выходить чаще. Процесс выпуска преднамеренно разработан, чтобы поощрить использовать его, и предполагает небольшую церемонию, перед тем как начать, а наличие правильных сеток безопасности позволяет вам не только предотвратить бедствие, но и быстро вернуться к полнофункциональной системе за считанные секунды.

Когда развертывание происходит без церемоний и с большей частотой, вы сможете лучше справляться с ошибками ПО, которое вы в настоящее время выполняете в рабочем окружении, что позволяет поддерживать более стабильную систему в целом.

Давайте еще немного поговорим о том, как выглядит эта страховочная сетка, и, в частности, о роли, которую играют разработчик, архитектор и операторы приложений в ее создании. Мы рассмотрим три неразрывно связанных шаблона:

- параллельное развертывание и службы, у которых есть версии;

- генерация необходимой телеметрии;
- гибкая маршрутизация.

В прошлом развертывание версии *n* некоего программного обеспечения почти всегда было заменой версии *n* – 1. Кроме того, мы разворачивали большие части программного обеспечения, охватывающие широкий спектр возможностей, поэтому, когда происходило непредвиденное, результаты могли быть катастрофическими. Например, все критически важные приложения могли испытывать значительные простои.

В основе вашей практики безопасного развертывания лежит параллельное развертывание. Вместо того чтобы полностью заменять одну версию запущенного программного обеспечения новой версией, вы продолжаете использовать известную рабочую версию, добавляя новую версию, чтобы запустить их вместе. Вы начинаете с небольшой части трафика, направляемой к новой реализации, и наблюдаете, что происходит. Вы можете контролировать, какой трафик направляется в новую реализацию, на основе множества доступных критериев, таких как: откуда поступают запросы (например, географически или на какую страницу ссылаются) или кто пользователь.

Чтобы оценить, дает ли эксперимент положительные результаты, вы смотрите на данные. Реализация работает без сбоев? Была ли введена новая задержка? Показатель кликабельности вырос или уменьшился?

Если все идет хорошо, можете продолжать увеличивать нагрузку, направленную на новую реализацию. Если вас что-то не устраивает, то можете перенести весь трафик обратно на предыдущую версию. Это путь отступления, который позволяет вам экспериментировать в рабочем окружении.

На рис. 2.13 показано, как это работает.

Ничего этого нельзя сделать, если игнорировать надлежащие дисциплины разработки программного обеспечения или если приложения не воплощают правильные архитектурные шаблоны. Вот некоторые из ключевых моментов, позволяющих активировать эту форму тестирования A/B:

- программные артефакты должны иметь версии, а механизм маршрутизации должен видеть версии, чтобы иметь возможность соответствующим образом направлять трафик. Кроме того, поскольку вы будете анализировать данные для определения стабильности нового развертывания и достижения желаемых результатов, все данные должны быть связаны с соответствующей версией программного обеспечения для правильного сравнения;
- данные, используемые для анализа работы новой версии, принимают различные формы. Некоторые метрики полностью независимы от каких-либо деталей реализации; например, задержка между запросом и ответом. Другие метрики начинают вглядываться в запущенные процессы, сообщая о таких вещах, как количество потоков или количество потребляемой памяти. И наконец, для принятия решений относительно развертывания также могут использоваться специфичные для домена метрики, такие как средняя общая сумма покупки при онлайн-транзакции. Хотя некоторые данные могут автоматически предоставляться средой, в которой выполняется реализация, вам не нужно писать код для их создания. Доступность метрик данных является первостепенной задачей. Я хочу, чтобы вы подумали о создании данных, которые поддерживают эксперименты в рабочем окружении;

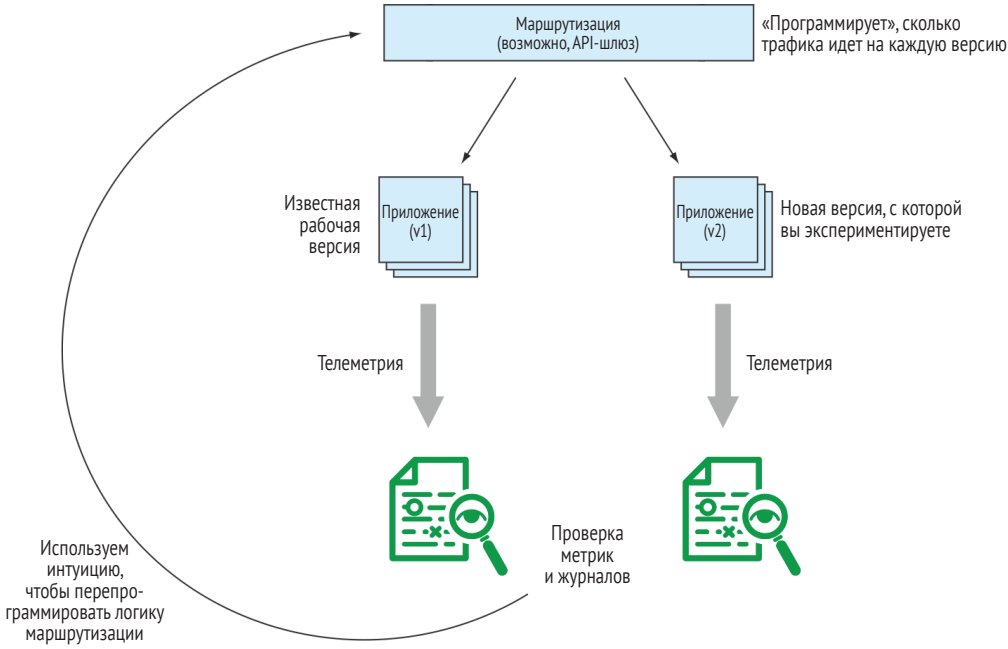


Рис. 2.13 ❖ Данные сообщают вам, как работают параллельные развертывания нескольких версий ваших приложений. Эти данные используются для программирования потоков управления этими приложениями, обеспечивая безопасное развертывание нового программного обеспечения в рабочем окружении

- ясно, что маршрутизация является ключевым фактором обеспечения параллельного развертывания, а алгоритмы маршрутизации являются частями программного обеспечения. Иногда алгоритм прост, например отправку процента от всего трафика в новую версию и «реализацию» программного обеспечения маршрутизации можно осуществить путем настройки некоторых компонентов вашей инфраструктуры. В других случаях вам может понадобиться более сложная логика маршрутизации, и вам потребуется написать код для ее реализации. Например, вы можете протестировать некоторые географически локализованные оптимизации и отправлять запросы только из той же географии в новую версию. Или, возможно, вам нужно представить новую функцию только своим премиум-клиентам. Независимо от того, ложится ли ответственность за реализацию логики маршрутизации на разработчика или достигается с помощью конфигурации среды выполнения, маршрутизация является первостепенной задачей для разработчика;
- и наконец, то, на что я уже намекала, – создание более мелких единиц развертывания. Вместо развертывания, охватывающего огромную часть вашей системы электронной коммерции – например, каталог, поисковую систему, службу изображений, механизм рекомендаций, корзину покупок и модуль обработки платежей – все в одном, – развертывания должны иметь гораздо меньшую область применения. Можно легко представить себе, что новая версия сервиса изображений представляет гораздо меньший риск для биз-

неса, чем та, что связана с обработкой платежей. Правильное компонентное представление ваших приложений – или, как многие называют это сегодня, «архитектура на основе микросервисов» – напрямую связано с работоспособностью цифровых решений[1].

Хотя платформа, на которой работают ваши приложения, обеспечивает некоторую необходимую поддержку для безопасного развертывания (я расскажу об этом подробнее в главе 3), все эти четыре фактора – версионирование, метрики, маршрутизация и компонентное представление – это то, что вы как разработчик должны учитывать, когда проектируете и создаете приложение для облачной среды. ПО для облачной среды – это нечто большее (например, проектирование переборок в вашей архитектуре, чтобы сбой потоком не обрушился на всю систему), но они являются одними из ключевых факторов безопасного развертывания.

2.2.4. Изменение – это правило

За последние несколько десятилетий мы получили достаточно свидетельств того, что операционная модель основана на убеждении, согласно которому наша среда изменяется только тогда, когда мы намеренно и сознательно инициируем такие изменения. Реагирование на неожиданные изменения доминирует во времени, потраченном ИТ-отделом, и даже традиционные процессы жизненного цикла разработки программного обеспечения, которые зависят от оценок и прогнозов, оказались проблематичными.

Как и в случае с новыми процессами цикла разработки ПО, которые я описывала в этой главе, наращивание мышечной массы, которая позволяет вам адаптироваться, когда вам навязывают изменения, дает гораздо большую устойчивость. Хитрость состоит в том, чтобы определить, что это за мышцы, когда дело доходит до стабильности и предсказуемости производственных систем. Эта концепция несколько коварна, слегка «промежуточна», если хотите; пожалуйста, потерпите немного.

Хитрость заключается не в том, чтобы лучше предсказывать неожиданные ситуации или выделять больше времени для устранения неполадок. Например, выделение половины времени команды разработчиков на реагирование на инциденты не помогает устранить первопричину экстренных мер. Вы реагируете на сбой, приводите все в рабочее состояние, и готово – и так до следующего инцидента.

«*Готово*»

В этом корень проблемы. Вы считаете, что после завершения развертывания, реагирования на инцидент или внесения изменений в брандмауэр вы каким-то образом завершили свою работу. Идея о том, что вы «закончили», по сути, рассматривает изменение как нечто, что заставляет вас превратиться в человека, который не довел дело до конца.

ПОДСКАЗКА Вы должны избавиться от понятия, что все уже сделано.

Давайте поговорим о *согласованности в конечном счете*. Вместо создания набора инструкций, которые переводят систему в состояние «готово», консистент-

[1] Google предоставляет более подробную информацию об этом в своем отчете «Accelerate: State of DevOps» на странице **http://mng.bz/vNap**.

ная в конечном счете система никогда не ожидает, что все сделано. Вместо этого система постоянно работает, чтобы достичь равновесия. Ключевые абстракции такой системы – желаемое состояние и фактическое состояние.

Желаемое состояние системы – это то, как вы хотите, чтобы она выглядела. Например, вы хотите, чтобы на одном сервере работала реляционная база данных, на трех серверах приложений работали веб-сервисы RESTful, а два веб-сервера предоставляли пользователям насыщенные веб-приложения. Эти шесть серверов правильно подключены к сети, и правила брандмауэра установлены соответствующим образом. Эта схема, как показано на рис. 2.14, является выражением желаемого состояния системы.

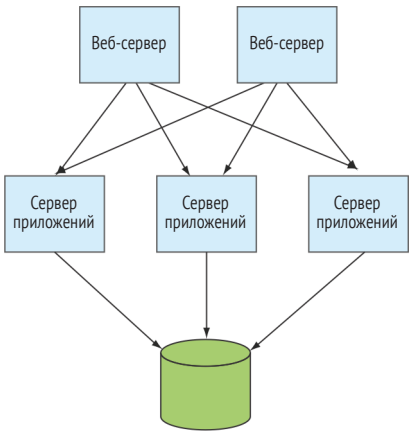


Рис. 2.14 ❖ Желаемое состояние вашего развернутого программного обеспечения

Вы надеетесь, что в какой-то момент, даже большую часть времени, у вас будет установлена эта система, и она будет хорошо работать, но вам и в голову не придет, что все будет так же, как вы оставили сразу после развертывания. Вместо этого вы рассматриваете *фактическое состояние*, модель того, что в настоящее время работает в вашей системе, как первостепенную сущность, создавая и поддерживая ее, используя метрики, которые вы уже рассмотрели в этой главе.

Консистентная в конечном счете система затем постоянно сравнивает *фактическое состояние* с *желаемым* и, если есть отклонение, выполняет действия, чтобы привести все в соответствие. Например, допустим, что вы потеряли сервер приложений из схемы, представленной на рис. 2.14. Это может произойти по ряду причин: аппаратный сбой, исключение out-of-memory, исходящее от самого приложения, или сетевой раздел, который отключает сервер приложений от других частей системы.

На рис. 2.15 изображены оба состояния: желаемое и фактическое. Фактическое состояние и желаемое состояние явно не совпадают. Чтобы привести их в соответствие, нужно запустить еще один сервер приложений и подключить его к сети на схеме, а приложение должно быть установлено и запущено на нем (вспомните более ранние дискуссии касательно повторяющегося развертывания).

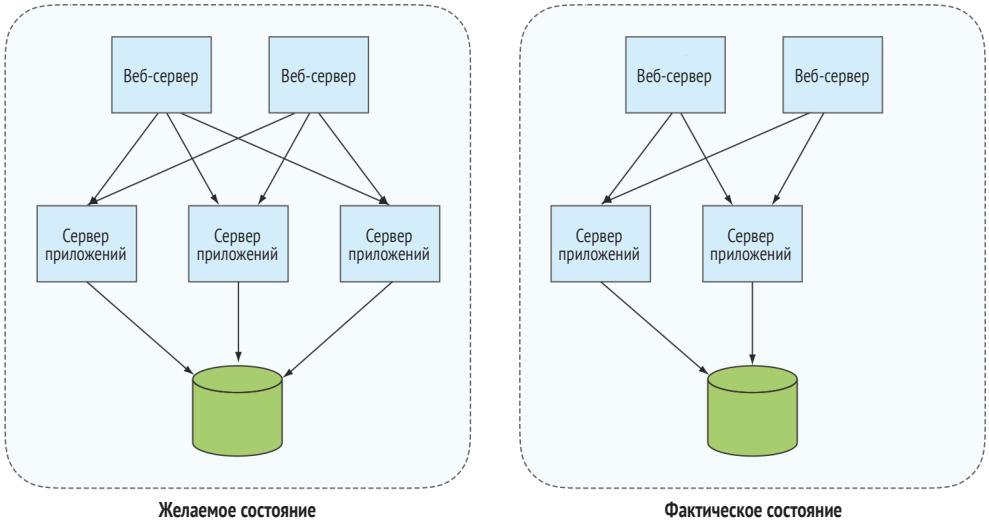


Рис. 2.15 ❖ Когда фактическое состояние не соответствует желаемому, консистентная в конечном счете система инициирует действия, чтобы привести их в соответствие

Для тех из вас, кто до этого, возможно, имел поверхностное представление о согласованности в конечном счете, это, скорее всего, немного напоминает ракетостроение. Опытный коллега избегает использования термина «*согласованность в конечном счете*», потому что опасается, что это вызовет страх у наших клиентов. Но системы, построенные на этой модели, становятся все более распространенными, и многие инструменты и образовательные материалы могут помочь претворить такие решения в жизнь.

И скажу вам следующее: это абсолютно, полностью, совершенно необходимо для запуска приложений в облаке. Я говорила это раньше: все постоянно меняется, поэтому лучше принять это изменение, чем реагировать на него. Не стоит бояться термина «*согласованность в конечном счете*». Нужно принять его.

Позвольте мне кое-что уточнить. Хотя система, о которой я здесь говорю, не обязательно полностью автоматизирована, необходимо наличие платформы, которая реализует основные части этой парадигмы (я подробнее расскажу о роли платформы в следующей главе). Я хочу, чтобы вы разрабатывали и создавали свое программное обеспечение таким образом, чтобы самовосстанавливающаяся система могла адаптироваться к постоянным изменениям, которые на нее воздействуют. Научить вас, как это сделать, – цель этой книги.

Программное обеспечение, разработанное, чтобы оставаться работоспособным перед лицом постоянных изменений, – своего рода святой Грааль, и влияние на стабильность и надежность системы очевидно. Самовосстанавливающаяся система поддерживает бóльшую продолжительность безотказной работы, чем та, что требует вмешательства человека каждый раз, когда что-то идет не так. А трактовка развертывания как выражения нового желаемого состояния значительно упрощает его и снижает риск. Принятие того факта, что изменение является правилом, фундаментально меняет природу управления программным обеспечением в рабочем окружении.

Резюме

- Для того чтобы достичь правильного эффекта с помощью кода, который вы пишете, вы должны уметь делать две вещи: легко и часто развертывать его и следить за тем, чтобы в рабочем окружении все работало как надо.
- Не следует обвинять разработчиков или операторов в том, что они пропустили какую-то из этих задач. Напротив, «вина» лежит на неисправной системе.
- Система дает сбой, потому что она разрешает индивидуальные решения, которые сложно поддерживать; создает среду, которая делает процесс развертывания программного обеспечения по своей сути рискованным; и рассматривает изменения в программном обеспечении и среде как исключение.
- Когда развертывание является рискованным, оно выполняется реже, что только делает его еще более рискованным.
- Вы можете инвертировать каждый из этих недостатков – сосредоточиться на повторяемости, сделать развертывание безопасным и принять изменения – и создать систему, которая поддерживает нужные вам методы, а не мешает им.
- В основе оптимизированных ИТ-операций лежит повторяемость, а автоматизация применяется не только по отношению к процессу сборки программного обеспечения, но и при создании сред выполнения и развертывании приложений.
- Шаблоны разработки программного обеспечения, а также методы эксплуатации ожидают постоянного изменения в облачных средах.
- Новая система зависит от высоко итеративного жизненного цикла разработки программного обеспечения, который поддерживает методы непрерывной доставки.
- Непрерывная доставка – это то, что нужно быстро реагирующему бизнесу, чтобы конкурировать на современных рынках.
- Более тонкая детализация всей системы – ключевой момент. Более короткие циклы разработки и более мелкие компоненты приложения (микросервисы) обеспечивают значительный рост гибкости и устойчивости.
- Согласованность в конечном счете правит бал в системе, где изменение является правилом.

Глава 3

Платформа для облачного ПО

О чем идет речь в этой главе:

- краткая история эволюции облачной платформы;
- основополагающие элементы платформы для облачной среды;
- основы контейнеров;
- использование платформы на протяжении всего жизненного цикла разработки программного обеспечения;
- безопасность, соответствие и контроль изменений.

Я работаю с большим количеством клиентов, чтобы помочь им понять и использовать шаблоны и методы, необходимые для работы в облачной среде, а также платформу, оптимизированную для запуска производимого ими программного обеспечения. В частности, я работаю с платформой Cloud Foundry и на ней. Я хочу поделиться опытом одного из моих клиентов, который воспользовался Cloud Foundry и развернул на ней существующее приложение.

Хотя это развернутое программное обеспечение придерживалось только нескольких облачных шаблонов, описанных в этой книге (приложения не имели состояния и были связаны с бэк-сервисами, поддерживающими необходимое состояние), мой клиент сразу осознал преимущества перехода на современную платформу. После развертывания на платформе Cloud Foundry они обнаружили, что программное обеспечение более стабильно, чем когда-либо. Первоначально они связывали это с непреднамеренным улучшением качества во время небольшого рефакторинга, выполненного для развертывания на Cloud Foundry.

Но, просматривая журналы приложений, они обнаружили нечто удивительное: приложение зависало так же часто, как и раньше. Они просто не заметили этого. Платформа приложений для облачной среды контролировала работоспособность приложения и в случае сбоя автоматически запускала замену. Скрытые проблемы остались, но опыт оператора и, что более важно, пользователя намного улучшился.

> **ПРИМЕЧАНИЕ** Мораль этой истории такова: хотя программное обеспечение для облачной среды предписывает множество новых шаблонов и методов, ни разработчик, ни оператор не несут ответственности за предоставление всей функциональности. Облачные платформы, разработанные для поддержки облачного программного обеспечения, предоставляют множество возможностей, которые поддерживают разработку и эксплуатацию этих современных цифровых решений.

Теперь позвольте мне прояснить ситуацию: я не предполагаю, что такая платформа должна ухудшать качество приложений. Если ошибка вызывает сбой, ее следует найти и исправить. Но такой сбой не обязательно разбудит оператора среди ночи или оставит пользователя в состоянии недовольства, пока проблема не будет устранена. Новая платформа обеспечивает набор сервисов, предназначенных для удовлетворения требований, которые я описала в предыдущих главах, требований к программному обеспечению, которое постоянно развертывается, чрезвычайно распределено и работает в постоянно меняющейся среде.

В этой главе я расскажу о ключевых элементах облачных платформ, чтобы объяснить, какие возможности вы можете использовать для них. Твердое понимание этих возможностей не только поможет вам сосредоточиться на потребностях своего бизнеса, не только на том, как поддерживать его на плаву, но также позволит вам оптимизировать свою реализацию для развертывания в облачной среде.

3.1. Эволюция облачных платформ

Использование платформ для поддержки разработки и эксплуатации программного обеспечения не новшество. Массово используемая платформа Java 2, Enterprise Edition (J2EE) появилась почти 20 лет назад, и с тех пор вышло семь ее основных релизов. JBoss, WebSphere и Web-Logic – коммерческие продукты этой технологии с открытым исходным кодом, которые принесли миллиардные доходы компаниям RedHat, IBM и Oracle соответственно. Многие другие проприетарные платформы, такие как TIBCO Software или Microsoft, были одинаково успешны и принесли пользу своим пользователям.

Но так же, как новые архитектуры необходимы для удовлетворения современных требований к программному обеспечению, новые платформы необходимы для поддержки новых реализаций и методов эксплуатации. Давайте кратко рассмотрим, как мы попали туда, где находимся сегодня.

3.1.1. Все началось с облака

Возможно, серьезное развитие облачных платформ началось с Amazon Web Services (AWS). Его первые продукты, обнародованные в релизах в течение 2006 года, включали в себя службы вычислений (Elastic Compute Cloud или EC2), хранилище (Simple Storage Service, или S3) и службы обмена сообщениями (Simple Queue Service, или SQS). Это определенно изменило ситуацию, поскольку разработчикам и членам команды эксплуатации больше не нужно было приобретать собственное оборудование и управлять им. Вместо этого можно было получать ресурсы, которые им требовались, в более короткие сроки, используя интерфейсы самообслуживания.

Первоначально эта новая платформа представляла собой перенос существующих моделей клиент–сервер в центры обработки данных, доступные через интернет. Архитектуры программного обеспечения не претерпели существенных изменений, равно как и методы разработки и эксплуатации, связанные с ними. На этапе становления понятие *облако* больше относилось к тому, *где* происходили вычисления.

Практически сразу характеристики облака начали оказывать давление на программное обеспечение, созданное для предоблачных инфраструктур. Вместо того чтобы использовать серверы «корпоративного уровня», сетевые устройства и хра-

нилище, AWS использовал в своих центрах обработки данных стандартные аппаратные средства. Использование менее дорогого оборудования стало ключом к предложению облачных сервисов по приемлемой цене, но вместе с этим вырос процент сбоев. AWS компенсировал снижение уровня отказоустойчивости оборудования в своем программном обеспечении и продуктах и представила своим пользователям абстракции, такие как *зоны доступности*, которые позволили бы программному обеспечению, работающему на AWS, оставаться стабильным, даже если инфраструктура не работала.

Здесь важно то, что когда пользователю сервиса предоставляют эти новые базовые элементы, такие как зоны доступности или *регионы*, пользователь берет на себя ответственность за правильное использование этих элементов. Возможно, в свое время мы этого не понимали, но предоставление этих новых абстракций в интерфейсе прикладного программирования (API) платформы стало оказывать влияние на новую архитектуру программного обеспечения. Люди начали писать программное обеспечение, которое было спроектировано таким образом, чтобы хорошо работать на такой платформе.

AWS эффективно создал новый рынок, и у таких конкурентов, как Google и Microsoft, ушло два года, чтобы получить какой-либо ответ. Когда это произошло, каждый из них представил свои уникальные продукты.

Google впервые вышел на рынок с Google App Engine (GAE), платформой, специально разработанной для запуска веб-приложений. Абстракции, которые он представил, первоклассные сущности в API, заметно отличались от тех, что были в AWS, который предоставлял преимущественно возможности для вычисления, хранения и работы с сетью; зоны доступности, например, обычно отображаются в наборы серверов, что позволяет абстракции предоставлять пользователю контроль над правилами привязки (affinity) и развязки (anti-affinity) для пула серверов. В отличие от этого, интерфейс GAE не предоставлял и по-прежнему не предоставляет никакого доступа к необработанным вычислительным ресурсам, на которых работают эти веб-приложения; он не дает доступа к активам инфраструктуры напрямую.

Компания Microsoft представила свой собственный вид облачной платформы, включая, например, возможность запуска *кода со средним уровнем доверия*. Подобно Google, Medium Trust предоставлял мало прямого доступа к вычислительным, сетевым ресурсам и хранилищам, а вместо этого взял на себя ответственность за создание инфраструктуры, в которой будет запускаться код пользователя. Это позволило платформе ограничить возможности пользовательского кода в инфраструктуре, тем самым предлагая определенные гарантии безопасности и устойчивости. Оглядываясь назад, я рассматриваю эти продукты от Google и Microsoft как самые ранние попытки перехода из облака в cloud-native.

Google и Microsoft в конечном итоге предоставили сервисы, которые давали доступ к абстракциям инфраструктуры, как показано на рис. 3.1, а AWS, напротив, стал предлагать облачные сервисы с абстракциями более высокого уровня.

Разные курсы, которые эти три поставщика прошли во второй половине 2000-х годов, намекали на значительные изменения, которые происходили в архитектурах программного обеспечения. Как отрасль мы экспериментировали, пытаясь понять, есть ли способы потребления и взаимодействия с ресурсами центра обработки данных, которые дадут нам преимущества в плане производительности, маневренности и устойчивости. Эти эксперименты в конечном итоге приве-

ли к формированию нового класса платформ – платформы для облачной среды, для которой характерны такие высокоуровневые абстракции, привязанные к ним сервисы и предоставляемые ими возможности.

	AWS	GCP	Azure
Вычисления	Elastic Compute Cloud (EC2)	Google Compute Engine	Виртуальные машины Azure
Хранение	Simple Storage Service (S3)	Google Cloud Storage	Хранилище blob-объектов Azure
Сеть	Виртуальное частное облако (VPC)	Виртуальное частное облако (VPC)	Виртуальная частная сеть (VPN)

Рис. 3.1 ❖ Предложения от основных IaaS-провайдеров

Платформа для облачной среды – это то, о чем вы узнаете в этой главе. Давайте начнем с того, что подробнее поговорим об абстракциях более высокого уровня, которые предоставляет такая платформа.

3.1.2. Тональный вызов

Разработчики и операторы приложений заботятся о том, правильно ли работает программное обеспечение, которое они используют для своих пользователей. В прошлые десятилетия, чтобы обеспечить надлежащий уровень обслуживания, они должны были правильно настроить не только развертывание приложений, но и инфраструктуру, в которой эти приложения работали. Это связано с тем, что доступные им примитивы были теми же компонентами вычислений, хранения и сети, с которыми они всегда работали.

Как мы уже упоминали в разделе, посвященном эволюции облачной платформы, который вы только недавно читали, ситуация меняется. Чтобы четко понять разницу, давайте рассмотрим конкретный пример. Скажем, у вас развернуто приложение. Чтобы убедиться, что оно работает нормально, или провести диагностику, когда что-то идет не так, у вас должен быть доступ к данным журнала и метрики.

Как я уже установила, развернуто несколько копий приложений для облачной среды для обеспечения устойчивости и масштабирования. Если вы используете эти современные приложения на инфраструктурно-ориентированной платформе, которая предоставляет традиционные объекты инфраструктуры, такие как хосты, тома хранения и сети, вы должны перемещаться по абстракциям традиционного центра обработки данных для получения доступа к этим журналам.

На рис. 3.2 показаны эти шаги.

1. Определите, на каких хостах работают экземпляры вашего приложения; обычно оно хранится в базе данных управления конфигурацией (CMDB).
2. Определите, на каком из этих хостов запущен экземпляр приложения, поведение которого вы пытаетесь диагностировать. Иногда это сводится к проверке одного хоста за раз, пока не будет найден нужный.
3. После того как вы нашли нужный хост, вы должны перейти к определенному каталогу, чтобы найти журналы, которые вы ищете.

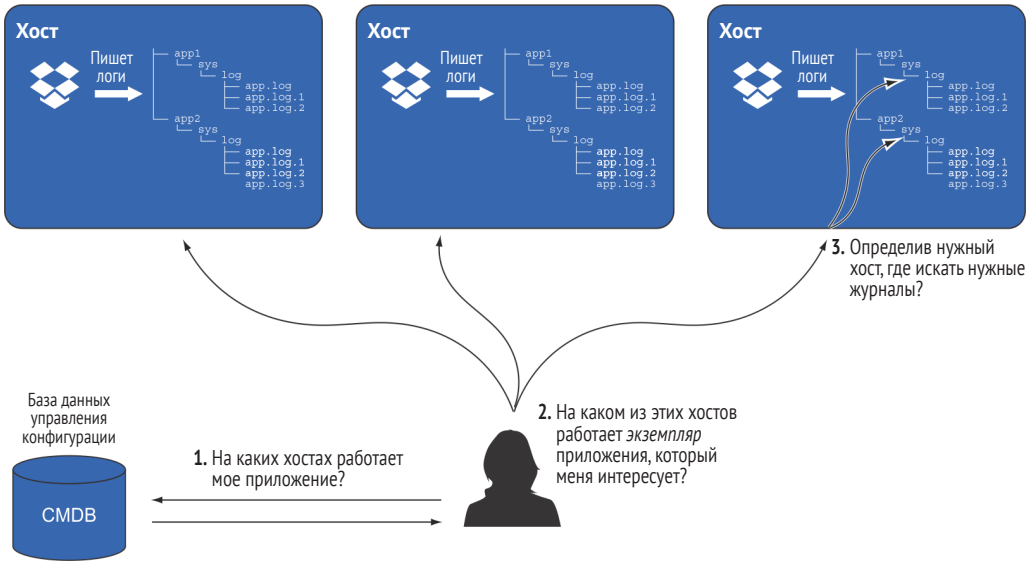


Рис. 3.2 ❖ Доступ к журналам приложений в инфраструктурно-ориентированной среде – утомительный процесс

Субъекты, с которыми взаимодействует оператор для выполнения работы, – это базы данных управления конфигурацией, хосты и каталоги файловой системы.

В отличие от этого, на рис. 3.3 показано восприятие и ответные действия оператора, когда приложения работают на облачной платформе. Все очень просто: вы запрашиваете журналы для вашего приложения и делаете запросы, ориентированные на приложение.

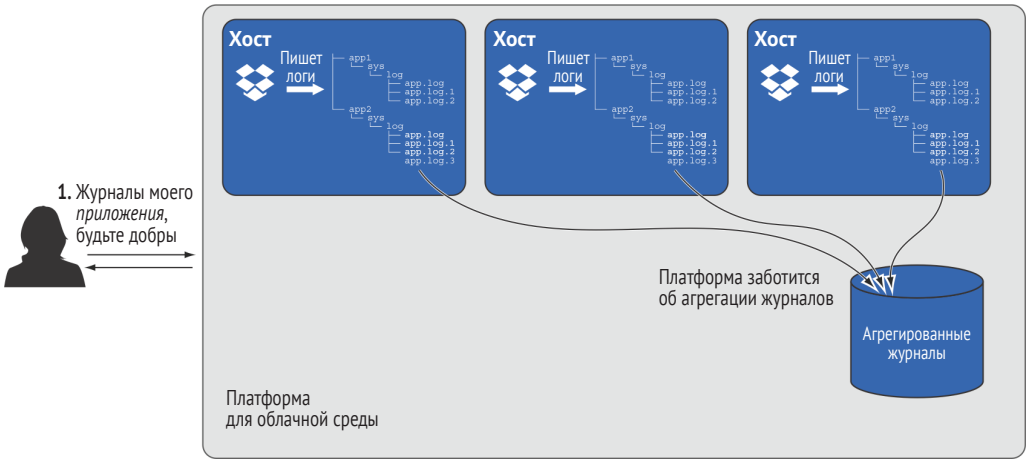


Рис. 3.3 ❖ Доступ к журналам приложений в среде, ориентированной на приложения, прост

Платформа для облачной среды берет на себя бремя, которое ранее возлагалось на оператора. Данная платформа нативно поддерживает понимание топологии приложения (до этого сохраненного в базе данных управления конфигурацией), использует ее для агрегирования журналов всех экземпляров приложения и предоставляет оператору данные, необходимые для интересующей его сущности.

Ключевым моментом является следующее: сущность, которая интересует оператора, – это *приложение*, а не хосты, на которых оно работает, или каталоги, в которых хранятся журналы. Оператору нужны журналы приложения, которое он диагностирует.

Контраст, который вы видите в этом примере,– это контраст между ориентированностью на инфраструктуру и ориентированностью на приложение. Разница в опыте взаимодействия оператора приложения обусловлена различием в абстракциях, с которыми он работает. Мне нравится называть это разницей в *тональном вызове* (*dial tone*).

> **ОПРЕДЕЛЕНИЕ** Платформы IaaS представляют *тональный вызов инфраструктуры*: интерфейс, обеспечивающий доступ к хостам, хранилищу и сетям – инфраструктурным примитивам.

> **ОПРЕДЕЛЕНИЕ** Платформа для облачной среды представляет *тональный вызов приложения*: интерфейс, который делает приложение первостепенным объектом, с которым взаимодействует разработчик или оператор.

Вы наверняка видели блоки на рис. 3.4, четко обозначающие три слоя, которые в конечном итоге объединяются, чтобы предоставить потребителям цифровое решение. Виртуализированная инфраструктура упрощает использование вычислительных ресурсов, хранилищ и сетевых абстракций, оставляя управление базовым оборудованием поставщику IaaS. Платформа для облачной среды повышает уровень абстракции еще больше, позволяя потребителю с большей легкостью использовать ресурсы ОС и промежуточного программного обеспечения, оставляя управление базовыми вычислениями, хранилищами и сетью поставщику инфраструктуры.

Аннотации с обеих сторон стека на рис. 3.4 предполагают различия в операциях, выполняемых с этими абстракциями. Вместо развертывания приложения на одном или нескольких хостах через интерфейсы IaaS оператор развертывает приложение на платформе для облачной среды, которая занимается распределением запрошенных экземпляров по доступным ресурсам. Вместо настройки правил брандмауэра для защиты границы хостов, на которых выполняется конкретное приложение, оператор применяет к приложению политику, а платформа заботится о защите контейнера приложения. Вместо того чтобы обращаться к хостам для доступа к журналам приложения, оператор обращается к журналам приложения. Экспериментальные различия между платформой для облачной среды и платформой IaaS, значительны.

То, о чем я расскажу в этой главе, и то, почему я призываю вас создавать собственное облачное программное обеспечение, – это платформа для облачной среды, которая генерирует тональный вызов приложения. На сегодняшний момент доступно несколько таких платформ. Крупные облачные провайдеры предлагают нам Google App Engine, AWS Elastic Beanstalk и Azure App Service (ни одна из кото-

рых не является широко распространенной). Cloud Foundry – это платформа для облачной среды с открытым исходным кодом, которая проникла в крупные компании по всему миру. Некоторые поставщики предлагают коммерческие продукты (Pivotal, IBM и SAP среди прочих)[1]. Хотя эти платформы могут отличаться друг от друга, все они имеют общую философскую основу и предоставляют тональный вызов приложения.

Рис. 3.4 ❖ Платформа для облачной среды абстрагирует проблемы инфраструктуры, позволяя командам сосредоточиваться на своих приложениях, а не на задачах более низкого уровня

3.2. Основные принципы платформы для облачной среды

Прежде чем углубляться в возможности и вытекающие из этого преимущества внедрения платформы для облачной среды, важно, чтобы вы поняли основы философии и фундаментальные шаблоны, на которых строится все остальное. Вас не должно удивлять, что этот базис на самом деле касается предоставления поддержки для сильно распределенных приложений, которые находятся в постоянно меняющейся среде. Но, прежде чем я представлю эти два элемента более подробно, давайте поговорим о технологиях, которые необходимы для таких платформ.

3.2.1. Вначале поговорим о контейнерах

Как оказалось, контейнеры являются отличным стимулирующим фактором программного обеспечения для облачной среды. Ну хорошо, это не совсем совпадение, о котором говорит мое несколько легкомысленное замечание, но это ситуация, относящаяся к разряду «курица и яйцо»: популярность контейнеров, без

[1] Раскрою секрет: на момент публикации этой книги в компании Pivotal я работаю на Cloud Foundry, Kubernetes и других новых платформах.

сомнения, была обусловлена необходимостью поддержки приложений для облачной среды, а доступность контейнеров в равной степени способствовала развитию облачного программного обеспечения.

Если, когда я использую термин «контейнер», вы сразу же думаете о Docker, это круто – почти угадали. Но я хочу охватить ключевые элементы контейнеров абстрактно, чтобы вам было проще подключить эти возможности к элементам программного обеспечения для облачной среды.

Начиная с самого базового уровня, *контейнер* – это вычислительный контекст, который использует функциональные возможности хоста, на котором он работает; например, базовая операционная система. Как правило, на одном хосте работает несколько контейнеров, последний из которых является сервером, физическим или виртуальным. Эти контейнеры изолированы друг от друга. На высшем уровне они похожи на виртуальные машины (ВМ), изолированную вычислительную среду, работающую на общем ресурсе. Контейнеры однако имеют меньший вес, по сравнению с виртуальными машинами, что позволяет создавать их на порядок быстрее и потребляющими меньше ресурсов.

Я уже упоминала, что несколько контейнеров, работающих на одном хосте, совместно используют операционную систему хоста, но не более того. Остальная среда выполнения, необходимая вашему приложению (и да, ваше приложение будет работать в контейнере), запускается внутри контейнера.

На рис. 3.5 показаны части приложения и среды выполнения, которые работают как на хосте, так и внутри ваших контейнеров. Хост предоставляет только *ядро* ОС. Внутри контейнера у вас сначала идет корневая файловая система ОС, включая функции операционной системы, такие как `openssh` или `apt get`. Среда выполнения, необходимая вашему приложению, также находится внутри контейнера, например Java Runtime Environment (JRE) или .NET Framework. И наконец, ваше приложение тоже находится в контейнере, и надеюсь, что оно работает там.

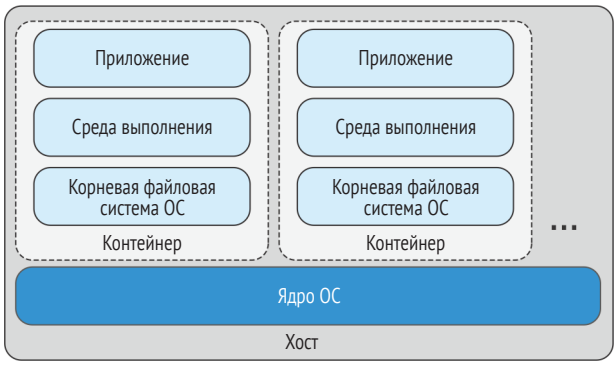


Рис. 3.5 ❖ На хосте обычно работает несколько контейнеров. Эти контейнеры используют ядро ОС из хоста. Но у каждого контейнера есть собственная корневая файловая система, среда выполнения и код приложения

Когда экземпляр приложения должен быть запущен, на хосте создается контейнер. Все составляющие, необходимые для запуска вашего приложения, – файловая

система ОС, среда выполнения приложения и само приложение – будут установлены в этот контейнер и будут запущены соответствующие процессы. Платформа для облачной среды, использующая в основе своей контейнеры, предоставляет множество функциональных возможностей для вашего программного обеспечения, и создание экземпляра приложения – всего лишь одно из них. В числе других:

- мониторинг работоспособности приложения;
- надлежащее распределение экземпляров приложений по всей инфраструктуре;
- назначение IP-адресов контейнерам;
- динамическая маршрутизация к экземплярам приложения;
- внедрение конфигурации
- и многое другое.

Ключевые моменты, которые я хочу, чтобы вы помнили о контейнере, когда начнете изучать, что привносит платформа для облачной среды: (1) в вашей инфраструктуре будет несколько хостов, (2) на хосте работает несколько контейнеров и (3) ваше приложение использует для своей функциональности ОС и среду выполнения, установленные в контейнере. На многих диаграммах, приведенных ниже, я изображаю контейнер с помощью значка, показанного на рис. 3.6.

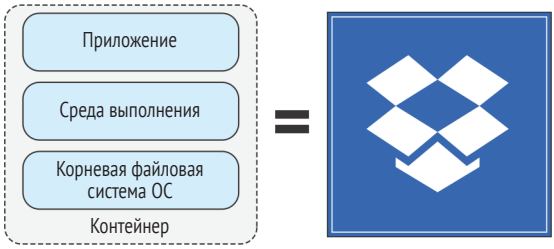


Рис. 3.6 ❖ При изучении возможностей платформы для облачной среды порой мы будем рассматривать контейнер как «черный ящик», внутри которого работает ваше приложение. Чуть позже мы подробно рассмотрим то, что работает в контейнере

Теперь, когда у вас есть базовое представление о контейнере, давайте посмотрим на ключевые принципы платформы для облачной среды.

3.2.2. Поддержка «постоянно меняющихся»

Я начала эту книгу с рассказа о сбое в Amazon, который продемонстрировал, что приложение может оставаться стабильным, даже если платформа, на которой оно работает, испытывает проблемы. Хотя разработчики играют решающую роль в достижении этой устойчивости посредством разработки своего программного обеспечения, они не должны нести ответственность за непосредственную реализацию каждой функции, отвечающей за стабильность. Платформа для облачной среды обеспечивает значительную часть этого сервиса.

Взять, к примеру, зоны доступности. Для обеспечения надежности Amazon предоставляет пользователям EC2 доступ к нескольким зонам доступности, давая им возможность развертывать свои приложения в разных зонах, чтобы приложе-

ния могли пережить сбой. Но когда в AWS происходят неполадки с зоной доступности, некоторые пользователи по-прежнему испытывают проблемы в онлайне.

Точная причина, безусловно, может быть разной, но в целом проблемы при развертывании приложений в зонах доступности происходят из-за того, что это весьма непростой процесс. Нужно отслеживать зоны доступности, которые вы используете, запускать экземпляры машин в каждой зоне, настраивать в них сети и решать, как развертывать экземпляры приложений (контейнеры) на виртуальных машинах, которые есть в каждой зоне доступности. Когда вы выполняете какое-либо обслуживание (например, обновление ОС), вы должны решить, будете ли вы делать это по одной зоне за раз или использовать другой шаблон. Нужно ли переносить рабочие нагрузки, потому что AWS выводит из эксплуатации хост, на котором вы работаете? Вы должны продумать всю схему целиком, чтобы увидеть, куда, в том числе и в какую зону доступности, следует перенести эту рабочую нагрузку. Это определенно сложно.

Несмотря на то что зона доступности – это абстракция, которую AWS предоставляет пользователю EC2, доступ к ней не должен быть предоставлен пользователю платформы для облачной среды. Вместо этого платформу можно настроить на использование нескольких зон доступности, и управлением экземплярами приложений в этих зонах затем займется платформа. Команда, занимающаяся разработкой приложения, просто запрашивает несколько экземпляров приложения, которое должно быть развернуто (скажем, четыре, как показано на рис. 3.7), а платформа автоматически распределяет их равномерно по всем доступным зонам. Платформа реализует всю оркестровку и управление. В противном случае это бремя легло бы на людей, если бы они не использовали платформу для облачной среды. Затем, когда происходит какое-то изменение (например, зона доступности отключается), приложение продолжает работать.

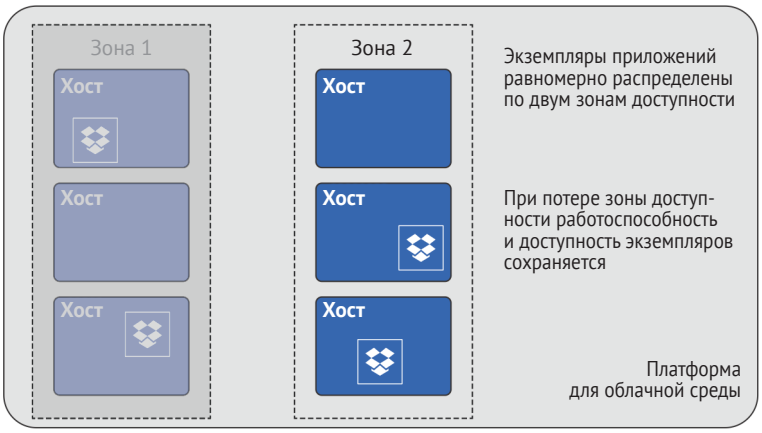


Рис. 3.7 ❖ Управление рабочими нагрузками в зонах доступности осуществляется платформой для облачной среды

Еще одна концепция, о которой я упоминала ранее, – это *согласованность в конечном счете*, ключевой шаблон в облаке, где происходят постоянные изменения. Развертывания и задачи управления, которые, как мы знаем, должны быть ав-

томатизированы, разработаны с таким расчетом, что они никогда не будут сделаны. Вместо этого управление системой осуществляется посредством постоянного мониторинга фактического (постоянно меняющегося) состояния системы в сравнении его с желаемым состоянием и исправления при необходимости. Эту технику легко описать, но сложно реализовать, и реализация такой возможности с помощью платформы для облачной среды имеет важное значение.

Ряд таких платформ реализует этот базовый шаблон, включая Kubernetes и Cloud Foundry. Хотя детали этой реализации немного отличаются, основные подходы одинаковы. На рис. 3.8 изображены основные действующие лица и базовая последовательность операций.

1. Пользователь выражает требуемое состояние, взаимодействуя с API платформы. Например, пользователь может попросить запустить четыре экземпляра конкретного приложения.
2. API платформы постоянно транслируют изменения для требуемого состояния в отказоустойчивое распределенное хранилище данных или инфраструктуру обмена сообщениями.
3. Каждый хост, на котором выполняются рабочие нагрузки, отвечает за передачу информации о состоянии того, что в них работает, в отказоустойчивые распределенные хранилища данных или инфраструктуру обмена сообщениями.
4. Действующее лицо, котороея здесь называю *компаратором*, получает информацию из хранилища состояний, поддерживает модель требуемого и фактического состояний и сравнивает их.
5. Если требуемое и фактическое состояния не совпадают, компаратор информирует о разнице другой компонент в системе.

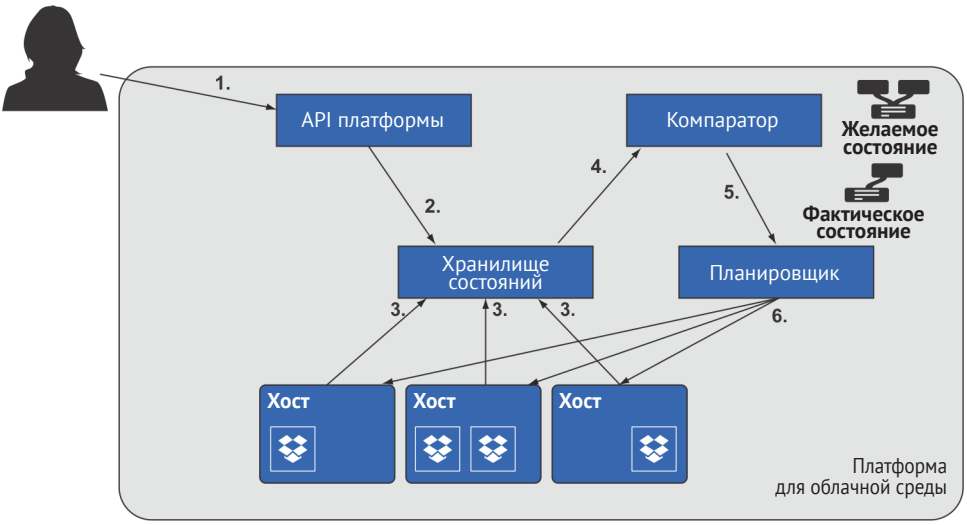


Рис. 3.8 ❖ Состояние приложений, работающих на платформе, управляется путем постоянного сравнения желаемого состояния с фактическим и последующим выполнением корректирующих действий при необходимости

6. Этот компонент, который я называю *планировщиком*, определяет, где следует создавать новые рабочие нагрузки или какие нагрузки должны быть отключены, и обменивается данными с хостами, чтобы это произошло.

Сложность тут заключается в распределенной природе системы. Честно говоря, распределенные системы сложны. Алгоритмы, реализованные в платформе, должны учитывать потерянные сообщения от API или хостов, сетевые разделы, которые могут быть короткими, но тем не менее нарушать поток, и изменения в чередовании маршрутов, которые иногда возникают из-за таких нестабильных сетей.

Такие компоненты, как хранилище состояний, должны иметь способы поддерживать состояние, когда входные данные конфликтуют (протоколы Paxos и Raft – одни из наиболее широко используемых в настоящее время). Точно так же, как командам разработки приложений не нужно заботиться о сложности управления рабочими нагрузками в зонах доступности, они также не должны обременять себя внедрением консистентных в конечном счете систем; этим занимается платформа.

Платформа представляет собой сложную распределенную систему, и она должна быть такой же устойчивой, как и распределенные приложения. Если компаратор выйдет из строя из-за сбоя или по какой-то запланированной причине, например обновления, платформа должна быть самовосстанавливающейся. Шаблоны, которые я описала здесь для приложений, работающих на платформе, также используются для управления платформой. Требуемое состояние может включать в себя 100 хостов, на которых выполняются рабочие нагрузки приложений, и распределенное хранилище состояний из пяти узлов. Если схема системы отличается от этой, корректирующие действия вернут ее в желаемое состояние. То, что я описала в этом разделе, является сложным и выходит далеко за рамки простой автоматизации шагов, которые ранее можно было выполнять вручную. Это возможности платформы для облачной среды, которые поддерживают постоянные изменения.

3.2.3. Поддержка «сильно распределенных»

После всех этих разговоров об автономности – групповой автономности, которая дает команде возможность разрабатывать и развертывать свои приложения без особых церемоний и жестко скоординированных усилий, а также самой автономности приложений, в которой отдельные микросервисы работают в собственной среде для поддержки независимой разработки и сокращения риска возникновения сбоев, которые обрушиваются потоком, – кажется, что многие проблемы решены. Так и есть, но (да, есть одно «но») из этого подхода вытекает система, состоящая из распределенных компонентов, которые в предыдущих архитектурах могли быть одноэлементными компонентами или размещаться внутри процесса, что вызывает сложность там, где раньше ее не было (или, по крайней мере, она была, но в меньшей степени).

Хорошая новость заключается в том, что мы, будучи отраслью, работаем над решением этих новых проблем в течение некоторого времени, и уже существуют довольно хорошо известные шаблоны. Когда один компонент должен обмениваться данными с другим, он должен знать, где найти этот другой компонент. Когда приложение масштабируется по горизонтали до сотен экземпляров, вам нужно

найти способ изменения конфигурации для всех экземпляров без необходимости массовой коллективной перезагрузки. Когда поток выполнения проходит через дюжину микросервисов, чтобы выполнить запрос пользователя, и он неэффективен, вам нужно найти, где в сложной сети приложений находится проблема. Вы должны продолжать повторную отправку запросов – это основополагающий шаблон в архитектурах программного обеспечения для облачной среды, в котором клиентская служба повторяет запросы, когда ответы не поступают, – все это напоминает DDoS-атаки[1].

Но помните, что разработчик не несет ответственности за реализацию всех шаблонов, необходимых для программного обеспечения, работающего в облачной среде; вместо этого помощь может оказать платформа. Давайте кратко рассмотрим некоторые возможности, предлагаемые такими облачными платформами в этом отношении.

Я хочу привести конкретный пример, чтобы проиллюстрировать несколько шаблонов, и буду использовать сайт обмена рецептами. Одной из предоставляемых услуг является список рекомендуемых рецептов, и для этого *служба рекомендаций* вызывает *службу избранного*, чтобы получить список рецептов, которые пользователь ранее отметил звездочкой. То, что попало в избранное, затем используется для расчета рекомендаций. У вас есть несколько приложений, каждое из которых имеет несколько развернутых экземпляров, и функциональность этих приложений и взаимодействие между ними определяет поведение вашего программного обеспечения. У вас есть распределенная система. Что может предоставить платформа для поддержки этой распределенной системы?

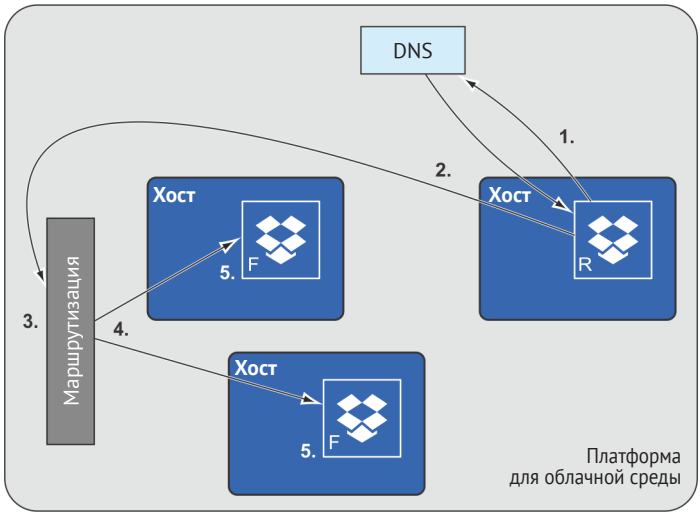
Обнаружение сервисов

Отдельные сервисы работают в отдельных контейнерах и на разных хостах; чтобы одна служба могла вызывать другую, она должна сначала найти другую службу. Один из способов как это может произойти, – хорошо известные шаблоны Всемирной паутины: DNS и маршрутизация. Служба рекомендаций вызывает службу избранного через свой URL-адрес, URL-адрес преобразуется в IP-адрес через просмотр DNS-записей, и этот IP-адрес указывает на маршрутизатор, который затем отправляет запрос в один из экземпляров службы избранного (рис. 3.9).

Еще один способ – предоставить службе рекомендаций прямой доступ к экземплярам службы избранного через IP-адрес, но, поскольку экземпляров этой службы большое количество, запросы должны быть сбалансированы по нагрузке, как и прежде. На рис. 3.10 показано, что в результате этого функция маршрутизации помещается в вызывающую службу, тем самым распределяя саму функцию маршрутизации.

---

[1] Распределенная атака «отказ в обслуживании» (DDoS) (**http://mng.bz/4OGR**) не всегда является преднамеренной или имеет злонамеренные цели.

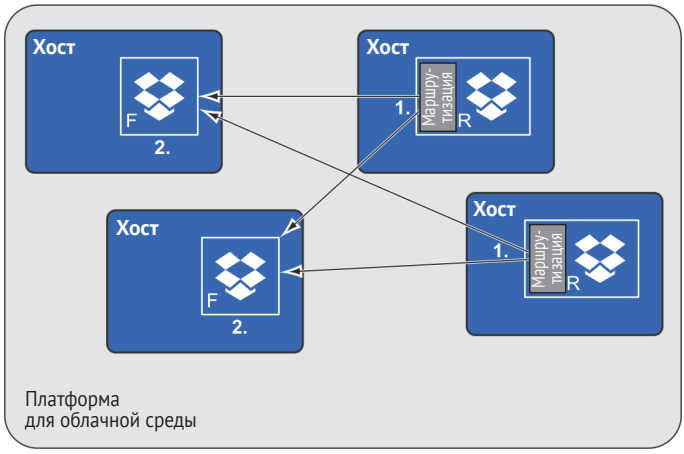


Приложение (R) рекомендаций должно получить доступ к службе (F) избранного и настроить URL-адрес доменного имени. Оно будет **1)** использовать DNS, чтобы найти IP-адрес службы избранного, и **2)** отправлять запрос на этот IP-адрес.

Этот IP-адрес разрешается в маршрутизатор **3)**, который затем **4)** направляет запрос одному из экземпляров службы избранного.

5. Поддержание в актуальном состоянии таблиц IP-адресов маршрутизатора с изменениями IP-адресов экземпляров приложения является функцией платформы для облачного приложения

Рис. 3.9 ❖ Служба рекомендаций находит службу избранного через просмотр DNS-записей и маршрутизацию



Приложению рекомендаций нужно получить доступ к службе избранного. У нее есть встраиваемый балансировщик нагрузки на стороне клиента, который будет **1)** направлять запросы на IP-адреса службы избранного.

2. Поддержание таблиц IP-адресов этих распределенных маршрутизаторов в актуальном состоянии с изменениями IP-адресов экземпляров приложения – функция платформы для облачной среды

Рис. 3.10 ❖ Служба рекомендаций напрямую обращается к службе избранного через IP-адрес; функция маршрутизации распределена

Независимо от того, является ли функция маршрутизации логически централизованной (рис. 3.9) или сильно распределенной (рис. 3.10), поддержание актуальности таблиц маршрутизации является важным процессом. Чтобы полностью автоматизировать этот процесс, платформа реализует такие шаблоны, как сбор информации об IP-адресах из только что запущенных или восстановленных экземпляров микросервиса и распространяет эти данные на компоненты маршрутизации, где бы они ни находились.

Конфигурация сервисов

Наши специалисты по обработке и анализу данных провели дополнительный анализ и в результате хотели бы изменить некоторые параметры для алгоритма рекомендации. У службы рекомендаций есть сотни развернутых экземпляров, каждый из которых должен получить новые значения. Когда механизм рекомендаций был развернут как отдельный процесс, вы могли перейти к этому экземпляру, предоставить новый файл конфигурации и перезапустить приложение. Но теперь, при наличии сильно распределенной программной архитектуры, никто (ни один человек) не знает, где работают все экземпляры в какой бы то ни было момент времени. А вот платформа для облачной среды знает.

Чтобы обеспечить такую возможность в облачной среде, требуется служба конфигурации. Этот сервис работает совместно с другими частями платформы, чтобы реализовать то, что показано на рис. 3.11. Изображенный там процесс выглядит следующим образом:

- оператор передает новые значения конфигурации службе конфигурации (вероятно, посредством фиксации в системе контроля и управления версиями);
- экземпляры службы знают, как получить доступ к службе конфигурации, от которой они получают значения конфигурации при необходимости. Конечно, экземпляры службы будут делать это во время запуска, но они также должны делать это при изменении значений конфигурации или при возникновении определенных событий жизненного цикла;
- когда значения конфигурации изменяются, хитрость заключается в том, чтобы каждый экземпляр службы обновлялся сам; платформа знает обо всех экземплярах службы. Фактическое состояние существует в хранилище состояний;
- платформа уведомляет каждый экземпляр службы о том, что доступны новые значения, и экземпляры принимают эти новые значения.

Опять же, ни разработчик, ни оператор приложения не несут ответственности за реализацию этого протокола; скорее, он автоматически предоставляется приложениям, развернутым в облачной платформе.

Обнаружение и настройка сервисов – это лишь две из множества возможностей, предлагаемых платформой для облачной среды, но они являются примерами поддержки среды выполнения, необходимой для модульной и сильно распределенной природы облачного приложения. Среди других сервисов можно упомянуть:

- механизм распределенной трассировки, позволяющий диагностировать проблемные запросы, которые проходят через множество микросервисов, автоматически встраивая трассировщики в эти запросы;
- предохранители, которые предотвращают непреднамеренные внутренние DDoS-атаки, когда нечто вроде сбоя в сети вызывает шквал повторных попыток.

Эти и многие другие сервисы – своего рода начальные ставки для облачной платформы, значительно снижающие нагрузку, которая в противном случае легла бы на плечи разработчика и оператора современного программного обеспечения, которое мы сейчас создаем. Использование такой платформы необходимо для высокофункциональной ИТ-компании.

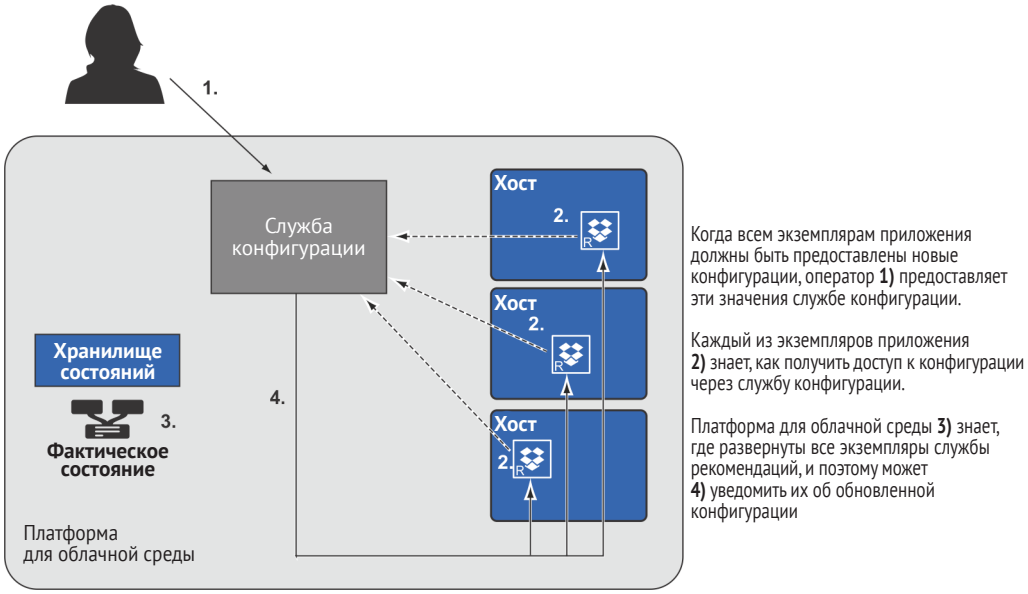


Рис. 3.11 ❖ Служба конфигурации платформы для облачной среды предоставляет важные возможности для развертывания приложений на базе микросервисов

3.3. Кто что делает?

Платформа для облачной среды может помочь в решении многих других задач – безопасности и соответствии требованиям, контроля изменений, многопользовательского режима и управления процессом развертывания, о котором я говорила в предыдущей главе. Но для того, чтобы в полной мере оценить ее ценность, сначала нужно поговорить о людях. В частности, я хочу сопоставить обязанности со структурой облачной платформы и центра обработки данных.

На рис. 3.12, это вариант рис. 3.4, показан тот же стек, но теперь я хочу сосредоточиться на границе, которая проходит между платформой для облачной среды и вашим программным обеспечением. На этой границе находится контракт, который определяет, как должно быть предоставлено программное обеспечение (API платформы), и набор уровней сервиса, которые обеспечивают гарантии того, насколько хорошо программное обеспечение будет работать на платформе.

Например, для запуска простого веб-приложения вы можете предоставить, используя API платформы, файлы JAR и HTML-файлы веб-приложения и ряда серверных служб, а также схему развертывания. Возможно, вам понадобится два экземпляра вашего веб-приложения и три экземпляра серверной службы, которые подключаются к базе данных клиентов. С точки зрения уровней сервиса контракт может предоставить гарантию того, что процент доступности приложения будет составлять 99,999 %, и у него будут все журналы, сохраненные в вашем экземпляре в системе Splunk.

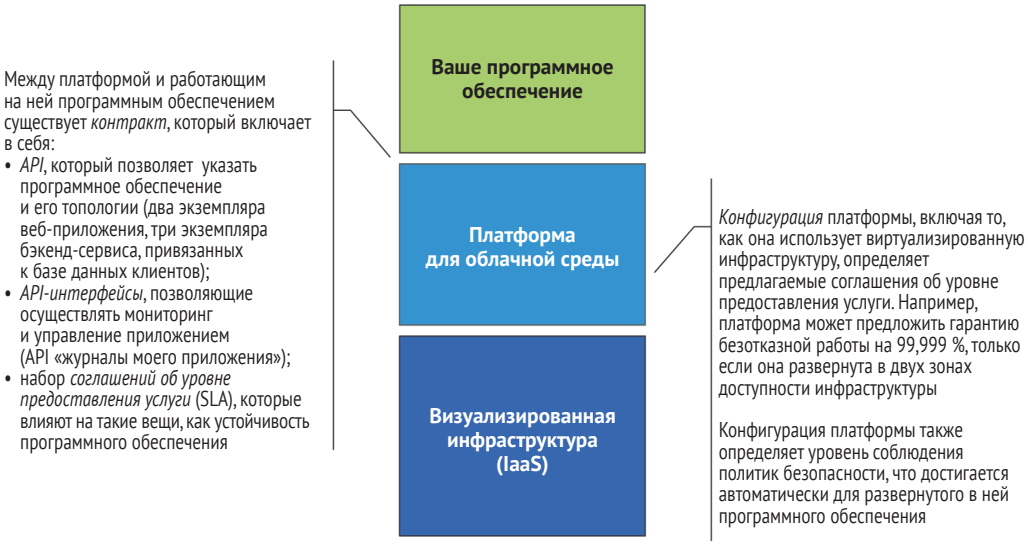


Рис. 3.12 ❖ Платформа для облачной среды предоставляет контракт, который позволяет потребителям развертывать и управлять своим программным обеспечением, не подвергаясь воздействию низкоуровневой инфраструктуры. Нефункциональные требования, такие как гарантии производительности, реализуются с помощью соглашений об уровне предоставления услуги, которые достигаются с помощью конкретных конфигураций платформы

Установление этих границ и контрактов дает мощную возможность: оно позволяет вам формировать отдельные группы. Одна группа отвечает за настройку платформы для облачной среды таким образом, чтобы обеспечить уровни сервиса, необходимые компании. Члены этой команды имеют определенный профиль навыков; они знают, как работать с ресурсами инфраструктуры, и понимают внутреннюю работу платформы для облачной среды и примитивов, что позволяет им точно настроить поведение платформы (что журналы приложений отправляются в Splunk).

Другая группа, или, скажем так, *группы*, – это команды, члены которых создают и эксплуатируют программное обеспечение для конечных потребителей. Они знают, как использовать API платформы для развертывания и управления приложениями, которые там работают. Члены этих групп имеют профили навыков, которые позволяют им понимать архитектуры программного обеспечения для облачной среды, и они знают, как контролировать и настраивать их для оптимальной производительности.

На рис. 3.13 показана часть полного стека, за которую отвечает каждая команда. Я хочу обратить ваше внимание на два элемента этой диаграммы:

- части стека, за которые отвечает каждая команда, не перекрываются. Это чрезвычайно расширяет возможности и является одной из основных причин, по которой развертывание приложений может происходить гораздо чаще, если использовать такую платформу. Однако этого можно достичь только в том случае, если контракт на границе между слоями спроектирован правильно;

- каждая команда «владеет» продуктом, за который она отвечает, и ей принадлежит весь жизненный цикл. Команда разработки приложения отвечает за сборку и эксплуатацию программного обеспечения; платформа дает членам этой команды контракт, который им необходим для этого. А команда, занимающая платформой, отвечает за создание (или настройку) и эксплуатацию продукта – платформы. Клиенты этого продукта – члены команды разработки приложения.

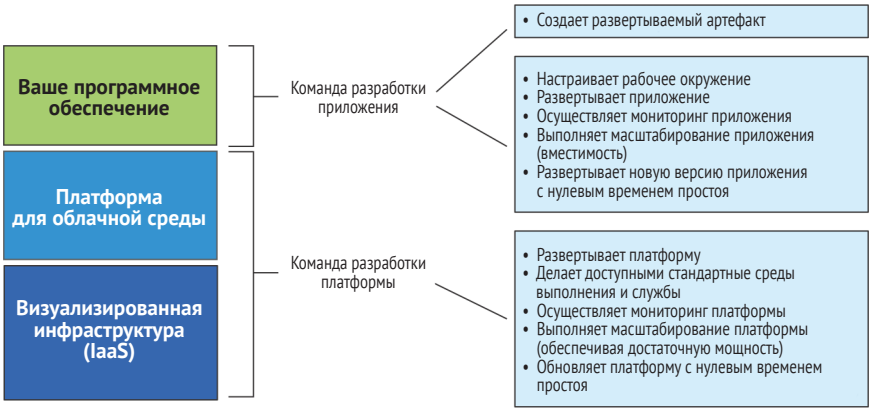


Рис. 3.13 ❖ Правильные абстракции поддерживают формирование автономных команд разработки платформ и приложений. Каждая из них отвечает за развертывание, мониторинг, масштабирование и обновление своих соответствующих продуктов

При наличии правильных контрактов команда разработки приложения и команда, занимающая платформой, автономны. Каждый может выполнять свои обязанности, не координируя в масштабном объеме свои действия с другими. Опять же, интересно отметить, насколько похожи их обязанности. Каждая группа отвечает за развертывание, настройку, мониторинг, масштабирование и обновление своих соответствующих продуктов. В чем состоит отличие, так это в продуктах, за которые они отвечают, и инструментах, которые они используют для выполнения этих обязанностей.

Но достижение подобной автономности, которая является таким важным компонентом при разработке программного обеспечения в нашу эпоху, зависит не только от определения контрактов, но и от внутренней работы самой облачной платформы. Платформа должна поддерживать методы непрерывной доставки, которые необходимы для достижения требуемой гибкости. Она должна обеспечивать превосходное качество работы, не допуская появления снежинок, и гарантировать автономность команде разработки приложений, одновременно с этим обеспечивая безопасность, нормативные и другие элементы управления. И она должна предоставлять сервисы, уменьшающие нагрузку, которая появляется при создании программного обеспечения, состоящего из множества сильно распределенных компонентов приложения (микросервисов), работающих в мультиарендной среде.

Я уже касалась некоторых из этих тем, когда говорила об основных принципах облачных платформ. Давайте теперь копнем поглубже.

3.4. Дополнительные возможности платформы для облачной среды

Теперь, когда вы представляете себе, какую основную поддержку платформы предоставляют для сильно распределенного программного обеспечения, работающего в среде постоянных изменений, а также как работают команды разработки приложений и платформы, давайте рассмотрим дополнительные факторы, в которых нужно разбираться, когда вы имеете дело с платформой для облачной среды.

3.4.1. Платформа поддерживает весь жизненный цикл разработки программного обеспечения

Нельзя достигнуть непрерывной доставки только за счет автоматизации развертываний в рабочем окружении. Успех начинается на раннем этапе жизненного цикла разработки программного обеспечения. Я говорила, что один развертываемый артефакт, который завершает жизненный цикл разработки программного обеспечения, крайне важен. Теперь вам нужны среды, в которых этот артефакт будет развернут, и способ заставить этот артефакт принять соответствующие конфигурации этих сред.

После того как вы, будучи разработчиком, убедились, что код работает на вашей собственной рабочей станции, вы загружаете его в репозиторий. После этого будет запущен конвейер, который создает развертываемый артефакт, устанавливает его в официальную среду разработки и запускает набор тестов. Если тесты прошли успешно, можно перейти к реализации следующей функции, и цикл продолжится. На рис. 3.14 показаны эти развертывания в среде разработки. Среда разработки содержит легкие версии различных сервисов, от которых зависит приложение, – базы данных, очереди сообщений и т. д. На диаграмме они представлены символами справа.

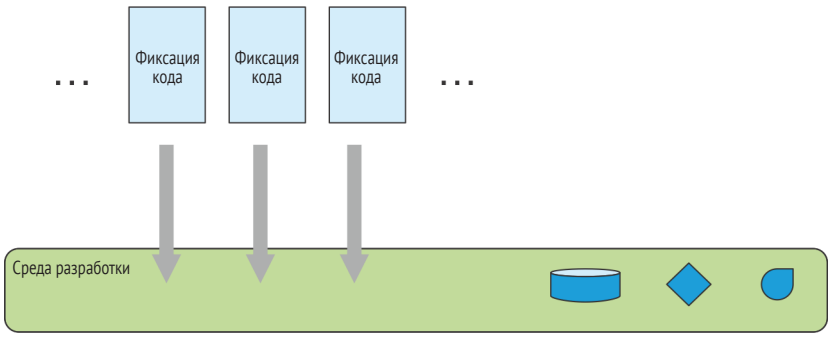


Рис. 3.14 ❖ При фиксации кода создаются развертываемые артефакты, которые развертываются в среде разработки, напоминающей рабочее окружение, но обладающей версиями служб, такими как базы данных и очереди сообщений (изображены символами справа)

Еще один, менее частый триггер, возможно, с привязкой ко времени, который запускается ежедневно, развернет артефакт для тестирования, где более полный

(и, вероятно, более продолжительный) набор тестов выполняется в окружении, которое чуть ближе к рабочему. Вы заметите, что на рис. 3.15 общая форма тестовой среды такая же, как у среды разработки, но они окрашены по-разному, что указывает на отклонения. Например, топология сети в среде разработки может быть плоской, а все приложения развертываются в одной и той же подсети, тогда как в тестовой среде сеть может быть разделена для обеспечения границ безопасности.

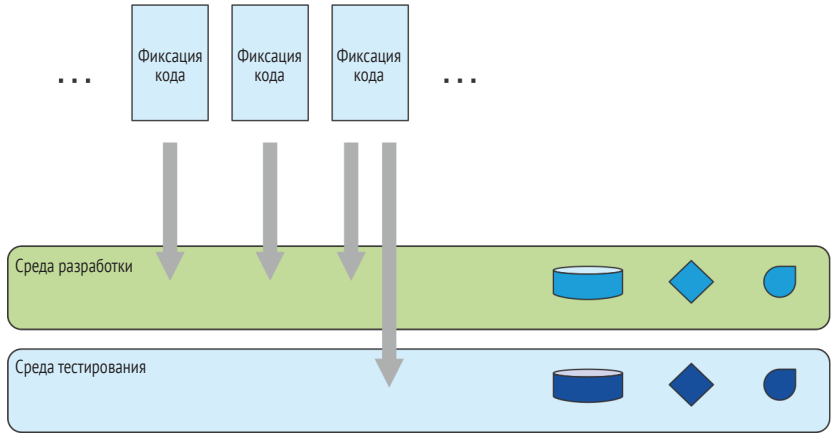


Рис. 3.15 ❖ Тот же развертываемый артефакт развертывается в окружении для обкатки, где он связан со службами (изображены символами справа), которые более точно соответствуют тем, что существуют в рабочем окружении

Экземпляры служб, доступных в каждой среде, также различаются. Их общая форма одинакова (если это реляционная база данных в среде разработки, то она будет таковой и в тестовой среде), но различие в окрашивании снова означает, что они отличаются. Например, в тестовой среде база данных клиентов, к которой привязано приложение, может быть версией всей производственной базы данных клиентов, очищенной от информации, позволяющей установить личность, тогда как в среде разработки это небольшой экземпляр с примерами данных.

Наконец, когда компания решает выпустить программное обеспечение, артефакт помечается версией релиза и развертывается в рабочей среде; см. рис. 3.16. Рабочее окружение, включая экземпляры сервисов, опять-таки отличается от среды тестирования. Например, здесь приложение привязано к действующей базе данных клиентов.

Хотя в средах разработки, тестирования и рабочем окружении и есть различия, я намекала на то, и хочу подчеркнуть это, что существуют и важные сходства. Например, API, используемый для развертывания в любой из сред, одинаков; управление основами автоматизации для оптимизированного процесса жизненного цикла разработки программного обеспечения с различными API-интерфейсами было бы ненужным бременем и препятствовало бы эффективности. Базовая среда, включающая в себя такие элементы, как операционная система, языковая среда выполнения, специальные библиотеки ввода-вывода и т. д., должна быть одинаковой во всех средах (я вернусь к этому, когда мы будем говорить об управлении

процессом в следующем разделе). Контракты, регулирующие обмен данными между приложением и любыми связанными сервисами, также являются едиными во всех средах. Говоря кратко, наличие паритета среды абсолютно необходимо для процесса непрерывной доставки, который начинается в среде разработки.

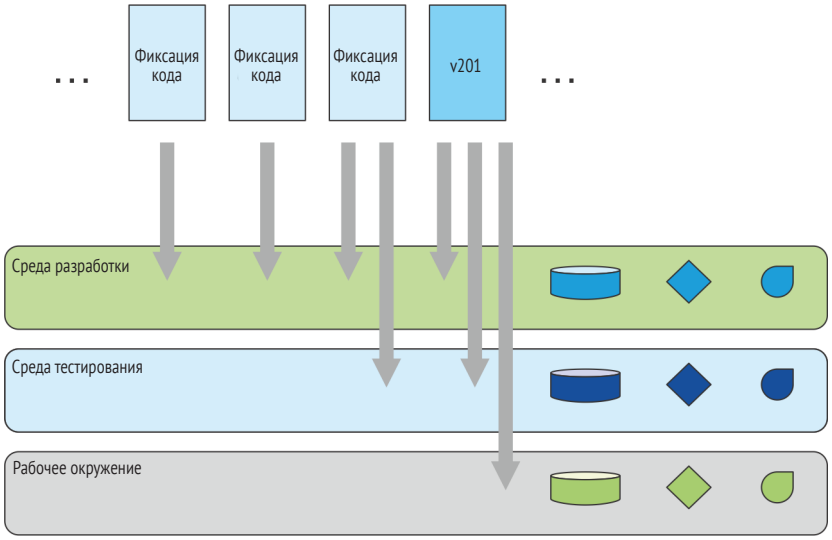


Рис. 3.16 ❖ Один и тот же артефакт разворачивается в похожих средах по всему жизненному циклу разработки программного обеспечения, и он должен поглощать неизбежные различия между ними

Управление этими средами является первостепенной задачей ИТ-компании, а платформа для облачной среды – это место для их определения и управления ими. Когда версия ОС в среде разработки обновляется, это идет в такт со всеми остальными средами.

Точно так же, когда какой-либо из сервисов набирает обороты (например, появляется новая версия RabbitMQ или Postgres), это происходит одновременно во всех средах.

Но, помимо обеспечения соответствия сред выполнения, платформа также должна предоставлять контракты, которые позволяют развернутым приложениям учитывать различия, существующие на разных этапах. Например, переменные среды, которые являются широко распространенным способом предоставления значений, необходимых приложению, должны быть переданы приложению так же, как и в жизненном цикле разработки программного обеспечения. И способ, которым сервисы связаны с приложениями, тем самым предоставляя аргументы соединения, также должен быть единым.

На рис. 3.17 дано визуальное представление этой концепции. Артефакт, развернутый в каждом из пространств, абсолютно одинаков. Контракты между приложением и конфигурацией среды, а также приложением и службами (в данном случае базой данных участников программы лояльности), также одинаковы. Обратите внимание, что стрелки, указывающие на каждый из развертываемых артефактов, абсолютно одинаковы во всех средах – отличаются подробности, лежащие в осно-

ве абстракций *env config* и *loyalty members*. Абстракции, подобные этим, являются крайне важной частью платформы, предназначенной для поддержки всего жизненного цикла разработки программного обеспечения.

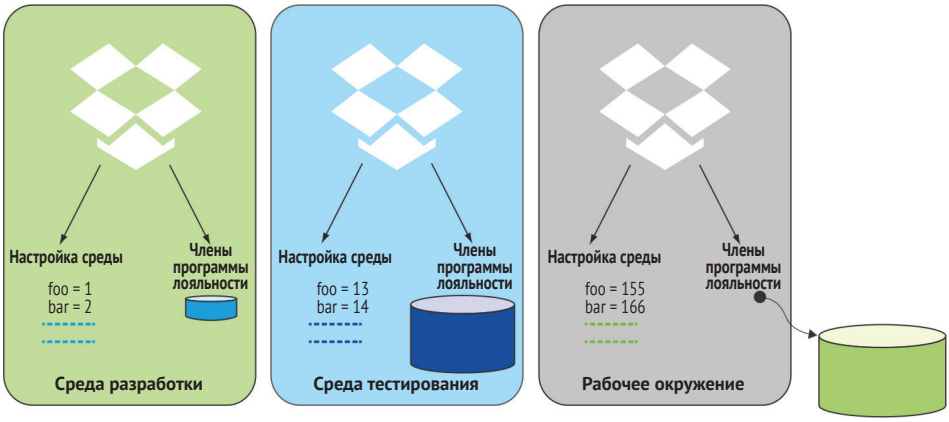


Рис. 3.17 ❖ Платформа должна включать в себя механизм, разрешающий контракт между приложением и средой выполнения, и привязанные службы, чтобы удовлетворять потребности жизненного цикла разработки программного обеспечения

Иногда у меня были клиенты, которые реализовывали платформу только для среды предварительной производственной эксплуатации или только для рабочего окружения. Нет сомнений в том, что платформа с такими возможностями, как автоматическое управление работоспособностью или средства контроля за стандартизированными машинными образами, – ценная вещь, даже если она доступна только в рабочем окружении. Но, учитывая необходимость непрерывной доставки ПО, платформа должна быть применена на протяжении всего жизненного цикла разработки программного обеспечения. Когда платформа предлагает паритет среды с правильными абстракциями и API, который можно использовать для автоматизации всех взаимодействий с ней, процесс разработки программного обеспечения и доведения его до производства можно превратить в предсказуемую, эффективную машину.

3.4.2. Безопасность, контроль над изменениями, соответствие требованиям (функции управления)

Я обнаружила, что многие, если не большинство, разработчики, не очень-то любят службу безопасности, соответствие или контроль изменений. С одной стороны, за что их винить? Разработчики хотят, чтобы их приложения работали в рабочем окружении, а эти функции управления требуют бесконечной подачи заявок и в конечном итоге могут остановить развертывание. С другой стороны, если разработчики на мгновение забудут о своем разочаровании, даже они должны признать ценность, которую приносят эти организации. Мы должны защищать личные данные наших клиентов и не должны допускать, чтобы изменения уничтожали критически важные бизнес-приложения. Мы должны ценить принятые

меры предосторожности, чтобы надзор не превратился в полномасштабный производственный инцидент.

Проблема с текущим процессом кроется не в людях и даже не в компаниях, из которых они пришли. Проблемы возникают потому, что существует слишком много способов ошибиться. Разработчик может, например, указать зависимость от конкретной версии JRE, которая, как известно, вызывала снижение производительности для определенных типов рабочих нагрузок и, следовательно, была запрещена в производственных системах, – поэтому контроль над изменениями необходим, чтобы он не попал в рабочее окружение. Есть ли у вас элемент контроля, который говорит, что каждый доступ пользователя к конкретной базе данных должен регистрироваться? Отдел, занимающийся вопросами соответствия, проверит правильность развертывания и настройки логгинг-агентов. Явная и часто ручная проверка соблюдения правил иногда является единственной точкой контроля.

Эти точки контроля реализуются в разных местах на протяжении всего жизненного цикла приложения и слишком часто довольно поздно помещаются в цикл. Когда дефект обнаруживается только за день до запланированного развертывания, график подвергается большому риску, сроки пропускаются, и все недовольны. Самый отрезвляющий момент тут, как показано на рис. 3.18, состоит в том, что эти элементы управления применяются к каждому развертыванию: к каждой версии, к каждому приложению. Время от точки «*готово к отправке*» до точки «*развернуто*» в лучшем случае исчисляется днями, а при наличии нескольких развертываний – неделями.

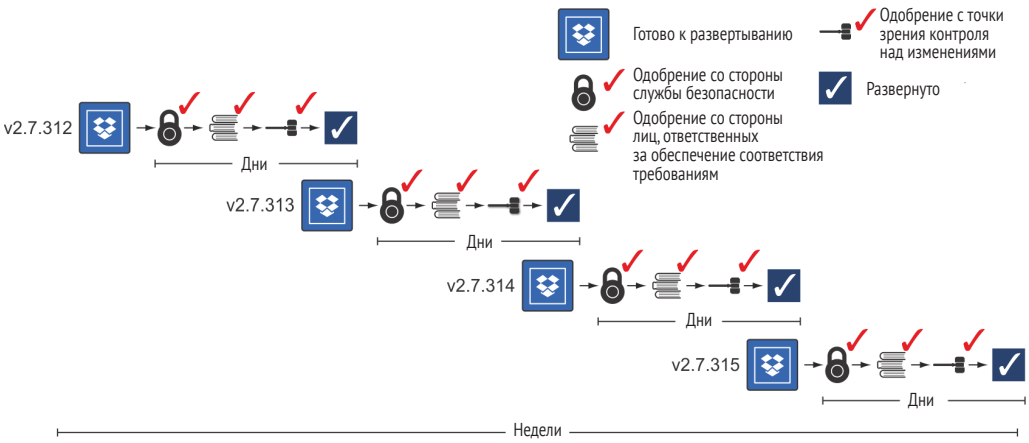


Рис. 3.18 ❖ Функции управления, которые находятся на критическом пути каждого выпуска каждого приложения, уменьшают количество развертываний, которые можно выполнить

Помните, я говорила о том, что Amazon выполняет десятки тысяч развертываний в день? Он делает нечто принципиально другое. Это не значит, что Amazon освобождается от нормативных требований или искажает личные данные

клиентов. Наоборот, Amazon соответствует требованиям, для которых элементы управления разработаны по-другому. Он встраивает элементы управления непосредственно в свою платформу, так что все, что там развернуто, гарантированно отвечает требованиям безопасности и нормативным требованиям.

Совсем скоро я расскажу о том, как работает встраивание этих элементов управления в платформу, но сначала давайте посмотрим на результат. Если развертывание гарантированно соответствует элементам управления, вам больше не нужно просматривать контрольный список, прежде чем это произойдет. А если у вас больше нет длинного контрольного списка, то время, когда у вас есть готовый артефакт, и время его развертывания резко сокращаются. Развертывание, которое раньше занимало дни, теперь занимает минуты. И хотя последовательность развертываний заняла недели, теперь у вас есть возможность выполнять несколько циклов в течение одного дня (рис. 3.19). Вы можете попробовать и получать отзывы гораздо чаще, чем раньше. Вы уже изучили множество преимуществ такого метода.

Рис. 3.19 ❖ Развертывание на платформе, которая реализует элементы управления, позволяет сократить до нескольких минут время между подготовкой приложения к развертыванию и его выполнением

Далее давайте подробно рассмотрим, как такие элементы управления встраиваются в платформу для облачной среды. Как получить гарантии безопасности, соответствия и контроля над изменениями, заявленные на рис. 3.19?

3.4.3. Контроль за тем, что идет в контейнер

В главе 2 я говорила о необходимости повторяемости и о том, что она достигается путем контроля над средой выполнения, развертываемым артефактом и самим процессом развертывания. Используя технологию контейнеров, можно внедрить этот уровень контроля непосредственно в платформу и, таким образом, получить гарантии безопасности, соответствия и контроля над изменениями, которые вам нужны.

Рисунок 3.20 повторяет рис. 2.12 из главы 2, в которой рассматривался способ объединения различных частей работающего приложения. Теперь, когда мы поговорили о контейнерах, мы можем сопоставить каждую часть этой диаграммы с той самой сущностью, которая будет работающим приложением.

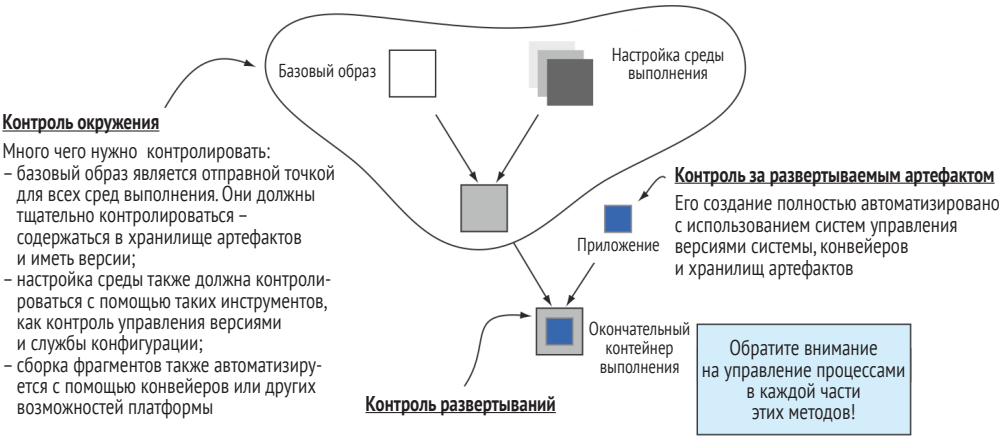


Рис. 3.20 ❖ Сборка стандартизированных базовых образов, управляемой настройки среды и отдельных развертываемых артефактов автоматизирована

На рис. 3.21 показан контейнер, работающий на хосте. То, что я называю «базовым образом» на рис. 3.20, теперь ясно показано как корневая файловая система ОС контейнера. Среда выполнения представляет дополнительные компоненты, установленные в корневой файловой системе, такие как среда выполнения JRE или .NET. И наконец, развертываемый артефакт также входит в контейнер. Итак, как контролировать каждый из этих фрагментов?

Во-первых, давайте поговорим о базовом образе. Напомню, что ядро операционной системы происходит из того, что работает на хосте (я вернусь к этому через минуту). В корневой файловой системе внутри контейнера есть дополнительные элементы, которые добавляются в это ядро. Можно рассматривать их как программные пакеты, установленные в ОС. Поскольку любое программное обеспечение, развернутое в операционной системе, может привести к уязвимости, лучше всего сохранять этот базовый образ минимальным. Например, если вы не хотите разрешать доступ в контейнер по протоколу SSH (ограничение доступа по SSH – *действительно* хорошая идея), вы бы не включили OpenSSH в базовый образ. Если вы затем будете управлять набором базовых образов, у вас будет значительный контроль над множеством характеристик безопасности ваших рабочих нагрузок.

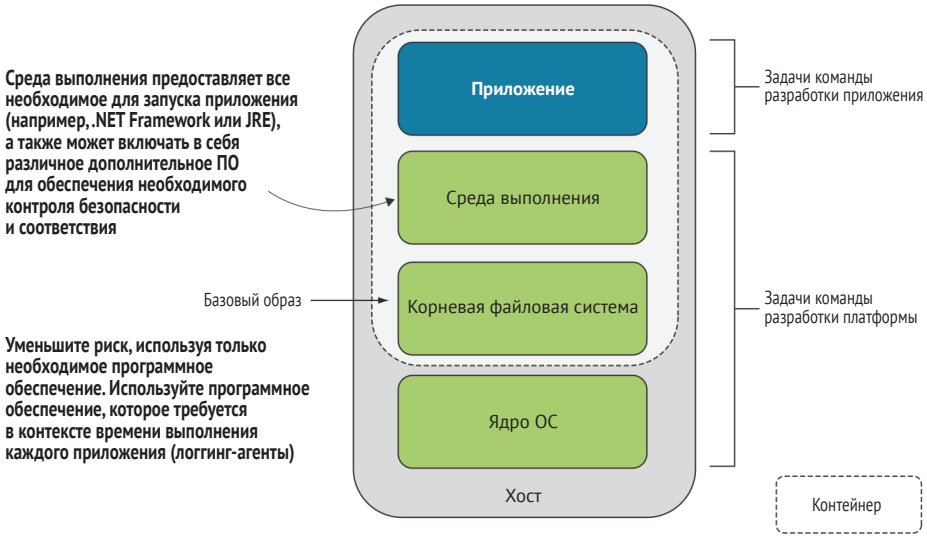


Рис. 3.21 ❖ Структура образа контейнера четко отделяет задачи команды разработки приложения от задач команды платформы

Лучше всего сделать базовый образ как можно меньше, а уменьшение площади атаки делает систему более безопасной. Но безопасность и соответствие также обеспечиваются гарантией того, что определенные процессы будут работать (например, логгинг-агенты). Пакеты программного обеспечения, необходимые для запуска в каждом контейнере, должны быть включены в базовый образ.

ПУНКТ 1 Платформа должна позволять использование только утвержденных базовых образов.

Этот базовый образ можно использовать в качестве основы для различных специализированных рабочих нагрузок. Например, некоторые из ваших приложений могут быть написаны на языке Java, и, следовательно, им нужна JRE, а другие приложения могут быть написаны на Python, и, следовательно, для них необходимо установить интерпретатор Python. Это роль того, что показано на рис. 3.21 в качестве среды выполнения. Конечно, части этой среды выполнения, такие как JRE или интерпретатор Python, могут сами по себе быть уязвимыми, поэтому у отдела безопасности будут определенные версии, одобренные для использования.

ПУНКТ 2 Платформа должна контролировать все среды выполнения, которые могут быть включены в контейнер.

Наконец, последнее, что есть в контейнере, – это само приложение, и практика тщательного создания этого развертываемого артефакта вполне ясна.

ПУНКТ 3 Создавайте конвейеры в сочетании со сканированием кода для обеспечения автоматизации, чтобы в повторяемой манере и безопасно создавать артефакт.

Давайте теперь перейдем к контрольным точкам. Я говорила о наличии архитектуры, которая отделяет проблемы команды разработки приложения от проблем, которые есть у команды, работающей с платформой. Команда разработки

приложения отвечает за доставку программного обеспечения, поддерживающего бизнес, а команда платформы отвечает за обеспечение требований безопасности и соответствия для предприятия. Команда приложения предоставляет только свое приложение, а команда платформы – все остальное.

Снова вернувшись к рис. 3.21, вы видите, что команда платформы предоставляет утвержденные базовые образы и утвержденные среды выполнения. Также видно, что эта группа отвечает за ядро ОС, работающее на хосте. Говоря кратко, команда платформы может применять элементы контроля над безопасностью и соответствием на каждом из уровней, которые вы только что рассмотрели.

3.4.4. Обновление и исправление уязвимостей

Когда какую-либо часть контейнера, изображенного на рис. 3.21, нужно обновить, работающие экземпляры не изменяются. Вместо этого вы развертываете *новые* контейнеры с новым набором компонентов. Поскольку приложения для облачной среды всегда имеют несколько развернутых экземпляров, вы можете перейти со старой версии на новую с нулевым временем простоя. Основной шаблон такого обновления состоит в том, что (1) подмножество экземпляров приложения закрывается и удаляется, (2) запускается равное количество экземпляров нового контейнера, и (3) после их запуска вы переходите к замене следующей партии старых экземпляров. Платформа для облачной среды выполняет этот процесс за вас. Вам нужно только предоставить новую версию приложения.

Посмотрите на первые слова первого абзаца этого раздела: «Когда *какую-либо часть* контейнера» нужно обновить – иногда меняется приложение, а иногда – все компоненты, предоставляемые платформой. Верно – последовательное обновление выполняется, когда у вас есть новая версия вашего приложения или когда у вас есть новые версии операционной системы (ядро или корневая файловая система) или что-либо еще в среде выполнения.

И даже еще лучше. Платформа для облачной среды, разработанная для удовлетворения потребностей всех команд, позволяет им работать независимо друг от друга. Это просто невероятно!

На рис. 3.22 показаны более ранние диаграммы, на которых команда приложения выполняла развертывания в средах разработки, тестирования и рабочем окружении. Теперь вы понимаете, что при каждом развертывании, предоставляемом командой приложения, контейнер собирается из частей, которые даются командой платформы, и частей, полученных командой приложения (см. рис. 3.21).

Из этого следует, что когда появляются новые версии частей контейнера, поставляемые командой, занимающейся платформой, новый контейнер также можно собрать. Если у команды приложения есть что-то новое, новый контейнер собирается и развертывается, и если у команды платформы есть что-то новое, новый контейнер собирается и развертывается. Это показано на рис. 3.22: наверху команда приложения создает новые контейнеры; и со стороны, по собственному графику, команда, которая занимается платформой, ревитирует элементы платформы. Команда платформы обновляет предоставленные платформой фрагменты контейнера.

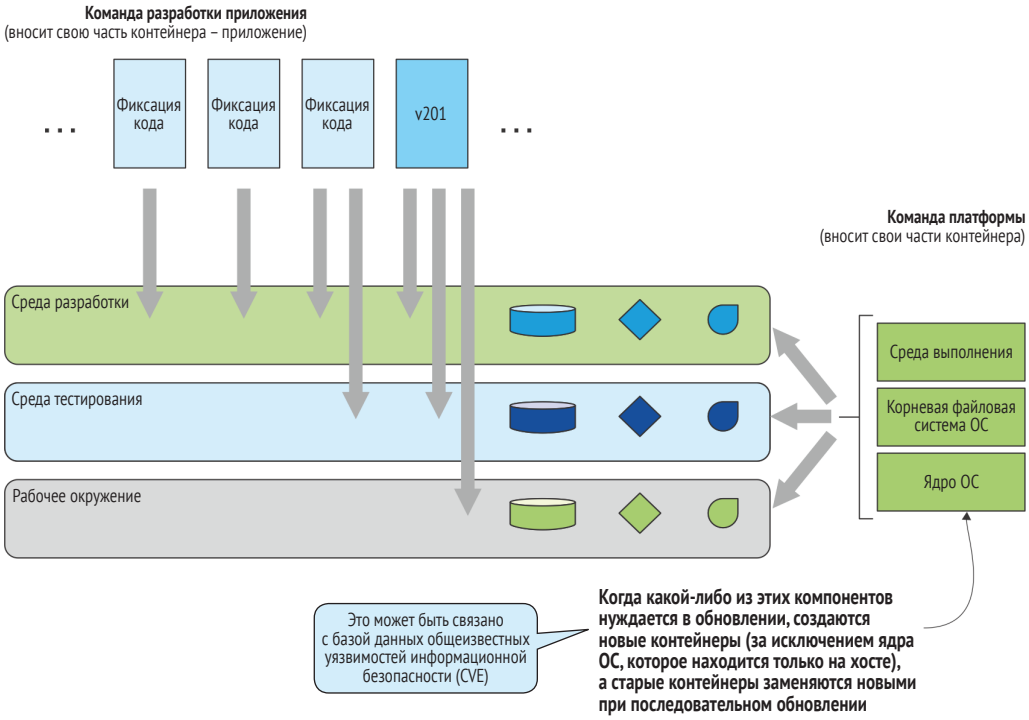


Рис. 3.22 ❖ Правильная платформа позволяет командам разработки приложений работать независимо от команд платформы

Такая автономность необходима для управления исправлениями в центре обработки данных. При обнаружении новой уязвимости (CVE)[1] необходимо быстро применить исправления, не прибегая к сложной координации всех приложений, работающих в центре обработки данных. Такой тип координации был одной из причин того, что исправления часто применялись не так быстро, как это должно было быть в предыдущих конфигурациях центра обработки данных.

Теперь при наличии платформы для облачной среды – как вы уже догадались – когда команда платформы выпускает исправление для последней уязвимости, платформа автоматически создает новый образ контейнера и затем заменяет работающие экземпляры партиями, всегда оставляя подмножество экземпляров приложения в работающем состоянии, когда другие участвуют в цикле. Это последовательное обновление. Конечно, вы не будете действовать безрассудно; сначала вы развернете патч в окружении для обкатки и проведете тесты там, и только после этого перейдете к развертыванию в среде эксплуатации.

[1] CVE (Common Vulnerabilities and Exposures) – база данных общеизвестных уязвимостей информационной безопасности. Более подробную информацию можно найти в Википедии на странице **https://en.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures**.

Если вы подумаете об этом на мгновение с точки зрения Google Cloud Platform, Amazon Web Services, Azure или любого другого поставщика облачных платформ, то этот тип автономности между командой платформы и пользователями платформы крайне важен. При наличии свыше миллиона активных пользователей[1] AWS не смог бы управлять своей платформой, если бы ей требовалась координировать действия с отдельными членами этой пользовательской базы. Вы, конечно же, можете применять те же методы в своем центре обработки данных с помощью платформы приложений для облачной среды.

3.4.5. Контроль над изменениями

Функция контроля над изменениями является последней линией обороны от изменений (например, обновления или развертывания нового приложения), вызывающих что-то плохое в среде эксплуатации. Это довольно сложная задача, которую обычно решают, внимательно изучая все детали запланированного развертывания и оценивая, как это может повлиять на другие системы, работающие в той же среде. Последствия могут включать в себя конфликты за вычислительные ресурсы, расширение или ограничение доступа к различным компонентам системы или резкое увеличение сетевого трафика. Что делает эту работу трудной, так это то, что многие вещи используются и развертываются в одной и той же ИТ-среде, поэтому изменение в одной области может иметь негативные последствия во многих других.

Платформа для облачной среды предоставляет принципиально иной способ решения проблем, связанных с контролем над изменениями. Она обеспечивает средства изоляции компонентов друг от друга, поэтому проблемы в одной части центра обработки данных не будут влиять на другие.

Полезно иметь имя, которое вы можете использовать для обозначения сущностей, которые нужно изолировать друг от друга; термин, который использую я, – *арендатор*. Когда я использую этот термин в данном контексте, то не имею в виду общеизвестные компании Coke и Pepsi, которые могут использовать одну и ту же среду, но должны быть настолько изолированы, что даже не знают друг о друге. Меня больше беспокоят арендаторы, уровень изоляции которых не позволяет им непреднамеренно влиять друг на друга. Наш разговор теперь касается темы мультиарендности: у вас есть множество арендаторов, и все они используют общую ИТ-среду.

Компания VMware стала пионером в области общей вычислительной инфраструктуры на рубеже веков. Она создала абстракцию виртуальной машины, ту же сущность, с которой вы взаимодействуете как физический ресурс – компьютер, – и программно управляемое распределение долей физических ресурсов на несколько виртуальных машин. Можно утверждать, что основная проблема, решаемая с помощью таких технологий виртуализации, – это совместное использование ресурсов, и многие, если не большинство, цифровые продукты, работающие

[1] См. статью Ингрид Лунден «Amazon's AWS Is Now a $7.3B Business as It Passes 1M Active Customers» (**http://mng.bz/Xgm9**) для получения более подробной информации.

на больших и малых предприятиях, теперь работают на виртуальных машинах. Независимые развертывания программного обеспечения являются арендаторами в общей вычислительной инфраструктуре, и это отлично подошло для программного обеспечения, созданного для запуска на машинах.

Но, как вы знаете, архитектуры изменились, и небольшие отдельные части, объединяющиеся в программное обеспечение для облачной среды в сочетании с гораздо более динамичной средой, в которой эти приложения работают, оказали воздействие на платформы на базе виртуальных машин. Хотя другие попытки предпринимались и раньше, подход с использованием контейнеров оказался выдающимся решением. На основе основополагающих концепций контрольных групп (cgroups), которые контролируют использование общих ресурсов, и пространств имен, которые контролируют их видимость, контейнеры Linux стали средой выполнения для сообщества микросервисов, которое формирует программное обеспечение для облачной среды[1].

Контейнеры обеспечивают часть изоляции, отвечающей интересам управления контроля над изменениями; приложение, которое поглощает всю память или ЦП, доступные в одном контейнере, не повлияет на другие контейнеры, работающие на том же хосте. Но, как я уже говорила, остаются другие проблемы. Кому разрешено развертывать контейнеры? Как можно быть уверенным в наличии данных мониторинга, достаточных для того, чтобы проверить, работает ли приложение в бешеном темпе? Как разрешить изменения в маршрутизации для одного приложения, не позволяя непреднамеренно вносить изменения в другое?

Ответ заключается в том, что сама платформа, которая обеспечивает управление доступом, мониторинг, функции маршрутизации и многое другое, должна быть осведомлена об арендаторе. На рис. 3.23 внизу показан набор хостов, где контрольные группы и пространства имен Linux обеспечивают необходимую вам изоляцию вычислений. В верхней части диаграммы представлен целый ряд других компонентов платформы, которые определяют ее использование. API платформы – это место, где применяется контроль доступа. Система метрик и ведения журналов должна группировать собранные данные в группы для отдельных арендаторов. Планировщик, который определяет, где будут запускаться контейнеры, должен знать об отношениях внутри арендатора и между арендаторами. Говоря кратко, платформа приложений для облачной среды является мультиарендной.

А мультиарендность – это то, что снимает напряжение с функции контроля над изменениями. Поскольку изменения в развертывании, обновлении и конфигурации, примененные к одному приложению/арендатору, изолированы от других приложений/арендаторов, команды приложений могут самостоятельно управлять программным обеспечением.

---

[1] Технология контейнеров первоначально была внедрена и использовалась в Linux, и большинство контейнероориентированных систем по-прежнему работает в этой операционной системе. Совсем недавно Windows добавила поддержку контейнеров, хотя в этом отношении тут она пока еще уступает Linux.

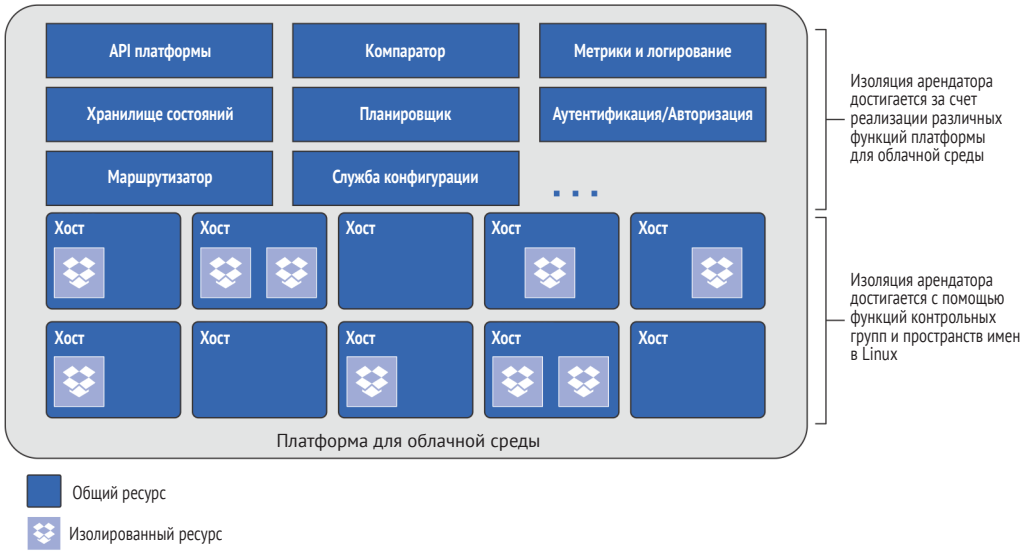


Рис. 3.23 ❖ Истинная мультиарендность на уровне вычислений совместно использует ресурсы в плоскости управления, а также в слое вычислений (хост и ядро Linux), применяя контейнеры для достижения изоляции ресурсов

Резюме

- Платформа для облачной среды берет на себя большую часть бремени для удовлетворения требований к современному программному обеспечению.
- Эта платформа используется на протяжении всего жизненного цикла разработки программного обеспечения.
- Эта платформа проектирует более высокоуровневую абстракцию, по сравнению с той, что была у инфраструктурно-ориентированных платформ последнего десятилетия.
- Благодаря внедрению функций управления в платформу развертывания могут выполняться гораздо чаще и безопаснее, по сравнению с ситуацией, когда для каждой версии каждого приложения требуется утверждение.
- Команды приложений и команды, занимающиеся платформами, могут работать независимо друг от друга, каждая из которой управляет созданием, развертыванием и обслуживанием соответствующих продуктов.
- В основе платформы лежит согласованность в конечном счете, поскольку она постоянно отслеживает фактическое состояние системы, сравнивает его с желаемым состоянием и исправляет при необходимости. Это относится как к программному обеспечению, работающему на платформе, так и к развертыванию самой платформы.
- По мере того как программное обеспечение становится более модульным и распределенным, службы, объединяющие компоненты в одно целое, также становятся таковыми. Платформа должна поддерживать такие распределенные системы.
- Платформа для облачной среды абсолютно необходима для компаний, создающих и эксплуатирующих облачное программное обеспечение.

Часть II

ШАБЛОНЫ ДЛЯ ОБЛАЧНОЙ СРЕДЫ

Здесь пойдет речь о шаблонах. Если вы ожидаете увидеть шаблоны в стиле «*Банды четырех*», я боюсь, что вы будете разочарованы, хотя надеюсь, что это не так. Книга Эриха Гаммы, Джона Влиссидеса, Ральфа Джонсона и Ричарда Хелма «*Шаблоны проектирования*» просто фантастическая, и, пожалуй, она более всех ответственна за рост осведомленности о шаблонах многократного использования в эпоху разработчиков программного обеспечения. Но, вместо того чтобы использовать подход в стиле справочника, все мое освещение шаблонов идет в контексте проблем, для решения которых они предназначены.

Практически каждая глава начинается с обсуждения определенных проблем, иногда говоря о подходах к проектированию, которые предшествовали эпохе облака, а затем предлагает решения – и эти решения являются шаблонами. Не случайно, что решения, которые я представляю, являются одними из тех, о которых вы, несомненно, слышали, – например sidecar-контейнеры и событийно-ориентированные архитектуры, но опять же, я надеюсь, что представление их таким образом поможет вам лучше понять и узнать, когда и как их применять.

Я начну эту часть со знакомства с событийно-ориентированного проектирования в главе 4. Большинство шаблонов, обсуждаемых в контексте облачных архитектур, неявно в основе своей предполагают подход типа «запрос/ответ». По правде говоря, именно так большинство из нас, естественно, рассматривает свое программное обеспечение. Я хочу посеять семена событийно-ориентированного мышления прямо с самого начала, чтобы вы, по крайней мере, держали его в глубине сознания, когда будете читать остальные главы. А потом я закрою эту часть книги в главе 12, снова рассказывая о событийно-ориентированных системах, на этот раз сосредоточив внимание на важной роли, которую они играют в предоставлении облачных данных. По общему признанию, это слишком краткий рассказ об облачных данных, но я надеюсь, что этого будет достаточно, чтобы завершить облачную картину для вас, по крайней мере не вдаваясь в подробности.

В промежутках между главами 4 и 12 я рассматриваю целый ряд шаблонов. В главах 5, 6 и 7 основное внимание уделяется облачным приложениям. В них рассказывается об отсутствии фиксации состояния, конфигурации и жизненном цикле приложения. Начиная с главы 8 я больше обращаю внимание на взаимодействия в облачной среде, вначале рассказывая об обнаружении сервисов и динамической маршрутизации. Затем в главах 9 и 10 я остановлюсь на шаблонах,

которые вы будете применять к каждой стороне взаимодействия, после того как оно будет установлено, – клиент и сервис. Я отмечу, что глава 4 о событийно-ориентированном проектировании также в основном посвящена взаимодействиям в облачной среде. Сильно распределенная архитектура, которая постоянно меняется, характеризует программное обеспечение для облачной среды и создает новые проблемы для устранения неполадок, и об этом пойдет речь в главе 11. Также интересно, что решения в этой главе используют многие шаблоны, описанные в предыдущих главах. И как я уже говорила, книга заканчивается знакомством с основными шаблонами данных для облачной среды.

Глава 4

Событийно-ориентированные микросервисы: не только запрос/ответ

О чем пойдет речь в этой главе:

- использование модели программирования «запрос/ответ»;
- применение модели программирования на базе событий;
- изучение обеих моделей для использования их с ПО для облачной среды;
- сходство и различие моделей;
- использование принципа Command Query Responsibility Segregation.

Одним из основных столпов, на котором стоит программное обеспечение для облачной среды, являются микросервисы. Разбиение того, что когда-то было большим, монолитным приложением, на набор независимых компонентов дало много преимуществ, в том числе повышение производительности труда разработчиков и более устойчивые системы при условии применения правильных шаблонов для развертываний на базе микросервисов. Но общее программное решение почти никогда не состоит из одного компонента; вместо этого микросервисы объединяются в коллекцию для формирования мощного ПО.

Однако здесь есть риск: если вы не будете осторожны, то можете склеить компоненты таким образом, что созданные вами микросервисы дадут вам лишь иллюзию слабой связанности. Вы должны быть осторожны, чтобы не воссоздать монолит, соединяя отдельные фрагменты слишком плотно или слишком рано. Шаблоны, о которых рассказывается в оставшейся части этой книги, предназначены для того, чтобы избежать этой ловушки и создать надежное и гибкое программное обеспечение в виде набора независимых компонентов, объединенных таким образом, чтобы максимизировать гибкость и устойчивость.

Но, прежде чем мы подробно рассмотрим эти темы, мне нужно затронуть еще одну сквозную тему – сквозную функциональность, в отношении которой будут применяться все другие облачные шаблоны: *базовый стиль вызова, используемый в вашей программной архитектуре*. Будет ли взаимодействие между микросервисами происходить в стиле «запрос/ответ» или в событийно-ориентированном стиле? В первом случае клиент делает запрос и ожидает ответа. Хотя запрашива-

ющая сторона может допустить, что этот ответ поступит асинхронно, само ожидание того, что ответ придет, устанавливает прямую зависимость одного от другого. Во втором случае стороны, потребляющие события, могут быть полностью независимы от тех, кто их производит. Такая автономность становится основой различий между этими двумя стилями взаимодействия.

В крупных и сложных развертываниях программного обеспечения используются оба подхода – иногда первый, иногда второй. Но я полагаю, что факторы, влияющие на наш выбор, гораздо более нюансированы, чем мы, вероятно, полагали в прошлом. И опять же, сильно распределенный и постоянно меняющийся облачный контекст, в котором работает наше программное обеспечение, приносит дополнительное измерение, которое требует от нас пересмотра и оспаривания наших предыдущих толкований и предположений.

В этой главе я начну со стиля, который кажется наиболее естественным для большинства разработчиков и архитекторов: запрос/ответ. Это настолько естественно, что вы, возможно, даже не заметили, что базовая модель, которую я представила в главе 1, едва ощутимо благоволит ей. Затем я оспариваю эту предвзятость и представляю событийно-ориентированную модель в нашем облачном контексте. Событийно-ориентированное мышление принципиально отличается от подхода «запрос/ответ», поэтому возможные последствия значительны. Вы будете изучать две эти модели с помощью примеров кода и в результате расширите свое представление о нашей ментальной модели программного обеспечения для облачной среды в части вопроса касательно взаимодействия.

4.1. (Обычно) нас учат императивному программированию

Подавляющее большинство студентов, независимо от того, учатся ли они программировать в аудитории или с помощью какого-то из множества источников, доступных онлайн, изучают императивное программирование. Они изучают такие языки, как Python, Node.js, Java, C # и Golang, большинство из которых предназначено для того, чтобы программист мог предоставить серию инструкций, выполняемых от начала до конца. Конечно, управляющие конструкции допускают ветвление и зацикливание, а некоторые инструкции будут вызовами процедур или функций, но даже логика внутри цикла или функции, например, будет выполняться сверху вниз.

Вы наверняка понимаете, к чему я веду. Эта последовательная модель программирования заставляет вас думать в форме «запрос/ответ». По мере выполнения набора инструкций вы делаете запрос функции, ожидая ответа. И в контексте программы, которая выполняется в одном процессе, это работает хорошо. Фактически процедурное программирование доминировало в этой отрасли в течение почти полувека[1]. Пока процесс программирования запущен и работает, кто-то, выполняющий запрос функции, может разумно ожидать ответа от функции, также выполняющейся в том же процессе.

Но наше программное обеспечение как единое целое больше не выполняется в одном процессе. В большинстве случаев разные части нашего программного

[1] Для получения дополнительной информации о процедурном программировании см. статью на странице **http://mng.bz/lp56**.

обеспечения даже не работают на одном компьютере. В сильно распределенной, постоянно меняющейся облачной среде инициатор запроса больше не может зависеть от немедленного ответа на запрос. Несмотря на это, модель «запрос/ответ» по-прежнему доминирует в качестве основной парадигмы программирования для интернета. Да, с появлением React.js и других подобных фреймворков реактивное программирование становится все более распространенным явлением для кода, выполняемого в браузере, но в программировании на стороне сервера по-прежнему преобладает подход «запрос/ответ».

Например, на рис. 4.1 показано значительное разветвление запросов к десяткам микросервисов, которое происходит, когда пользователь получает доступ к своей домашней странице Netflix. Этот слайд был взят из презентации, которую старший инженер-программист Скотт Мэнсфилд представляет на многочисленных конференциях, в которых он рассказывает о шаблонах, использующихся Netflix для компенсации случаев, когда ответ на нисходящий запрос не поступает немедленно.

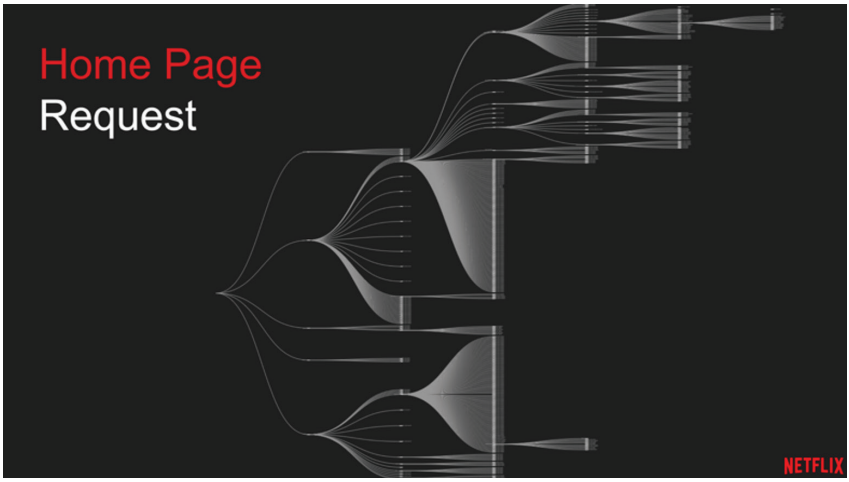


Рис. 4.1 ❖ Диаграмма, представленная в презентации Скотта Мэнсфилда из компании Netflix, показывает, что один запрос на получение домашней страницы пользователя приводит к значительному разветвлению запросов на нисходящие микросервисы

Я привожу эту конкретную диаграмму, потому что она отлично иллюстрирует масштаб проблемы. Если бы запрос домашней страницы был успешным только тогда, когда все каскадные запросы, изображенные в этом дереве, также были успешными, у Netflix появилось бы очень много разочарованных клиентов. Даже если каждый микросервис может похвастаться почти стопроцентной доступностью (99,999 %) *и* сеть всегда работает[1], один запрос с менее чем 100 нисходящими зависимостями «запрос/ответ» понижает процент доступности, который теперь будет составлять приблизительно 99,9 % в корне. Я не видела оценки

[1] См. № 1 в списке ошибок распределенных вычислений на сайте Википедии (**http://mng.bz/BD90**).

потери доходов Netflix, когда их сайт находится в автономном режиме, но если мы вернемся к оценкам *Forbes* об экономическом пакте до итогового показателя Amazon, то 500 минут простоя обойдутся Amazon в 80 млн долл. в год.

Конечно, Netflix и многие другие очень успешные сайты работают намного лучше, внедряя такие шаблоны, как автоматические повторные запросы, когда ответы не принимаются, и у них есть несколько работающих экземпляров микросервисов, которые могут выполнять запросы. И как опытно представил Скотт Мэнсфилд, кеширование также может помочь обеспечить перемычку между клиентами и сервисами[1]. Эти и многие другие шаблоны построены вокруг стиля вызова «запрос/ответ», и хотя вы можете и будете продолжать укреплять эту основу с помощью дополнительных методов, вам также следует рассмотреть другую основу для создания этих шаблонов.

4.2. Повторное знакомство с событийно-ориентированными вычислениями

Если углубиться в детали, можно встретить различные мнения относительно того, из чего состоит событийно-ориентированная система[2]. Но даже благодаря этой изменчивости одна вещь является общей. Сущность, запускающая выполнение кода в событийно-ориентированной системе, не ожидает какого-либо ответа – она работает по принципу «*выстрелил и забыл*». Выполнение кода имеет эффект (иначе зачем вообще его запускать), а результат может привести к тому, что с программным обеспечением произойдет еще что-то, но сущность, которая запустила выполнение, не ожидает ответа.

Эту концепцию легко понять с помощью простой диаграммы, особенно если рассматривать ее в контрасте с шаблоном «запрос/ответ». В левой части рис. 4.2 изображен простой запрос и ответ: код, который выполняется при получении запроса, находится на крючке, чтобы предоставить какой-то ответ запрашивающей стороне. В противоположность этому в правой части данного рисунка показан событийно-ориентированный сервис: результат выполнения кода не имеет прямого отношения к событию, которое его вызвало.

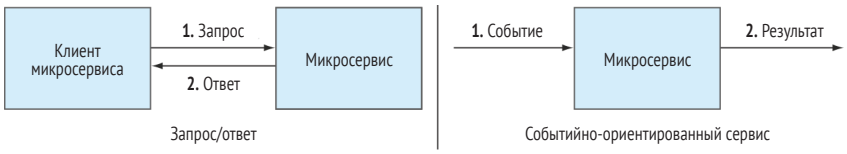


Рис. 4.2 ❖ Сравните базовый примитив обоих стилей

В этих двух диаграммах есть пара интересных моментов. Во-первых, слева в танце участвуют две стороны: клиент микросервиса и сам микросервис. Партнеры в танце зависят друг от друга, чтобы все шло гладко. Справа изображена только одна сторона, и это важно. Микросервис выполняется в результате события, но то,

[1] См. это видео на сайте YouTube (**http://mng.bz/dPvN**).

[2] В статье Мартина Фаулера данная тема исследуется более подробно (**http://mng.bz/YPla**).

что вызвало это событие, вообще не касается микросервиса. В результате сервис имеет меньше зависимостей.

Во-вторых, обратите внимание, что событие и результат полностью отключены. Отсутствие связи между ними даже позволяет мне рисовать стрелки на разных сторонах микросервиса. При использовании стиля «запрос/ответ» слева я бы не смогла этого сделать.

Последствия данных различий довольно глубокие. Лучший способ начать разбираться в них – это конкретный пример. Давайте перейдем к первому фрагменту кода в этой книге.

4.3. Моя глобальная поваренная книга

Я люблю готовить, и я трачу гораздо больше времени на просмотр блогов, связанных с едой, чем хотелось бы признать. У меня есть мои любимые блогеры (**https://food52.com** и **https://smittenkitchen.com**, я смотрю на тебя), а также мои любимые «официальные» публикации (**www.bonappetit.com**). Сейчас я хочу создать сайт, который собирает контент со всех моих любимых сайтов и позволяет мне организовать его. Да, в основном мне нужен агрегатор блогов, но, возможно, это нечто специально для моей `зависимости`, ээ, моего хобби.

Одно из представлений контента, которые меня интересуют, – это список последних сообщений, которые приходят с моих любимых сайтов: есть сеть людей или сайтов, на которые я подписана, какие у них последние сообщения? Я назову этот набор контента *Connections' Posts*, и он будет создан службой, которую я собираюсь написать. Две части собираются воедино, чтобы сформировать этот контент: список людей или сайтов, на которые я подписана, и список контента, предоставленного этими людьми. Контент для каждой из этих двух частей предоставляется двумя дополнительными сервисами.

На рис. 4.3 изображена связь между этими компонентами. Данная диаграмма не отображает какой-либо конкретный протокол между различными микросервисами, а только отношения между ними. Для вас это прекрасный пример, чтобы мы могли глубже взглянуть на оба протокола: «запрос/ответ» и управление с помощью событий.

4.3.1. Запрос/ответ

Как я уже говорила ранее в этой главе, использование протокола «запрос/ответ», чтобы конкретным образом свести воедино компоненты, показанное на рис. 4.3, для большинства людей является наиболее естественным. Когда мы думаем о создании набора постов, написанных моими любимыми блогерами, легко сказать: давайте сначала получим список людей, которые мне нравятся, а затем поищем посты, которые оставил каждый из этих людей. Конкретно поток развивается следующим образом (см. рис. 4.4):

1) JavaScript в браузере отправляет запрос в службу Connections' Posts, предоставляя идентификатор для человека, который запросил веб-страницу (скажем, это вы), и ожидает ответа. Обратите внимание, что ответ может быть возвращен асинхронно, но протокол вызова по-прежнему является протоколом «запрос/ответ», поскольку ожидается ответ;

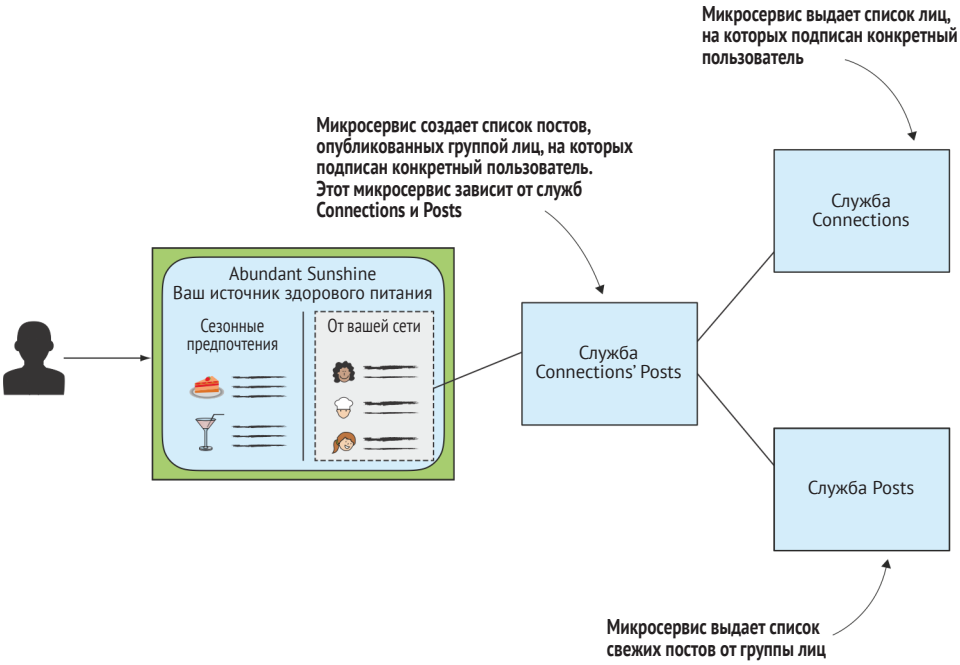


Рис. 4.3 ❖ На сайте Abundant Sunshine будет отображаться список постов, опубликованных моими любимыми блогерами о еде. Агрегация – это сочетание сети лиц, на которых я подписана, и опубликованных ими постов

2) служба Connections' Posts отправляет запрос в службу Connections с этим идентификатором и ожидает ответа;
3) служба Connections выдает в ответ список блогеров, на которых вы подписаны;
4) служба Connections' Posts отправляет запрос в службу Posts с этим списком блогеров, только что полученным от службы Connections, и ожидает ответа;
5) служба Posts выдает в ответ список постов для этого набора блогеров;
6) служба Connections' Posts создает композицию данных, полученных в ответах, и сама отвечает на веб-страницу.

Давайте посмотрим на код, который реализует шаги, которые я изобразила на рис. 4.4.

Настройка

Этот и большинство примеров, приведенных в книге, требуют, чтобы у вас были установлены следующие инструменты:

- Maven;
- Git;
- Java 1.8.

Я не буду требовать от вас написания кода. Вам нужно только скачать его из GitHub и выполнить несколько команд для сборки и запуска приложений. Хотя это не книга по программированию, я использую код, чтобы продемонстрировать архитектурные принципы, которые рассматриваю.

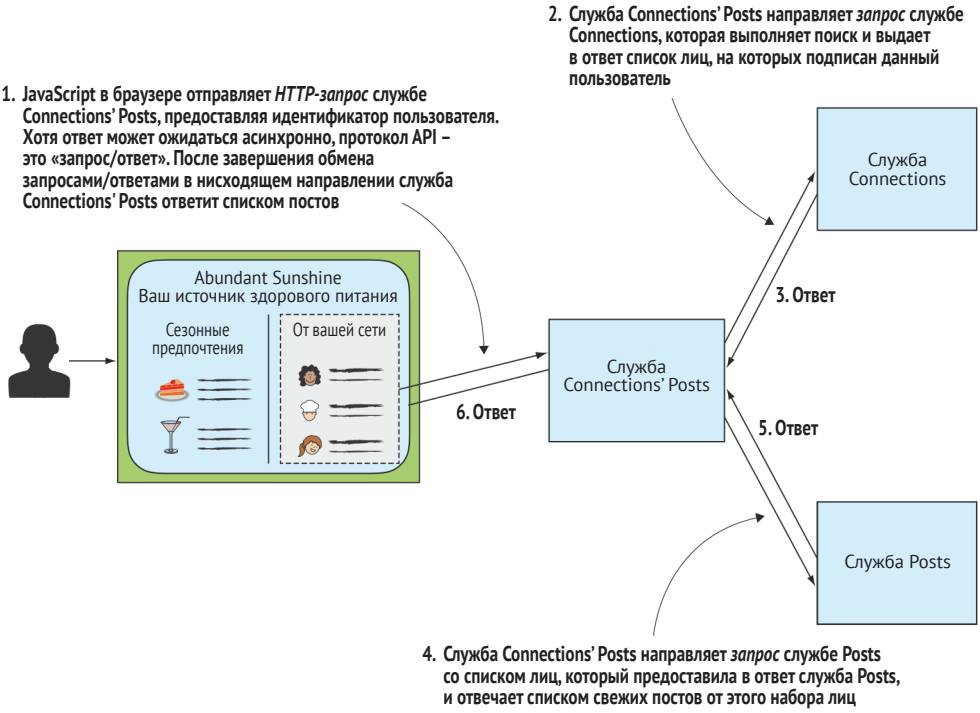


Рис. 4.4 ❖ Рендеринг фрагмента веб-страницы зависит от последовательности скоординированных запросов и ответов

Получение и создание микросервисов

Мы начнем с клонирования репозитория cloudnative-abundantsunshine с помощью приведенной ниже команды, а затем перейдем в этот каталог:

```
git clone https://github.com/cdavisafc/cloudnative-abundantsunshine.git
cd cloudnative-abundantsunshine
```

Здесь вы увидите подкаталоги, содержащие примеры кода, которые появляются в разных главах этого текста. Код для этого первого примера находится в каталоге cloudnative-requestresponse, поэтому я позволю вам продвинуться на один уровень глубже в проекте:

```
cd cloudnative-requestresponse
```

Вскоре вы ознакомитесь с исходным кодом примера, но сначала давайте приступим к работе. Приведенная ниже команда создает код:

```
mvn clean install
```

Запуск микросервисов

Теперь вы увидите, что новый файл cloudnative-requestresponse-0.0.1-SNAPSHOT.jar создан в подкаталоге target. Это то, что мы называем *fat jar* – приложение Spring Boot полностью автономно, включая контейнер Tomcat. Следовательно, чтобы запустить приложение, вам нужно только запустить Java, указав путь к JAR-файлу:

```
java -jar target/cloudnative-requestresponse-0.0.1-SNAPSHOT.jar
```

Микросервисы теперь готовы и работают. В отдельном окне командной строки вы можете подключиться к службе Connections' Posts, используя команду `curl`:

```
curl localhost:8080/connectionsposts/cdavisafc
```

Чтобы получить ответ:

```
[
  {
    "date": "2019-01-22T01:06:19.895+0000",
    "title": "Chicken Pho",
    "usersName": "Max"
  },
  {
    "date": "2019-01-22T01:06:19.985+0000",
    "title": "French Press Lattes",
    «usersName»: «Glen»
  }
]
```

В рамках запуска этого приложения я предварительно заполнила несколько баз данных образцом контента. Этот ответ представляет собой список постов отдельных лиц, на которых я, `cdavisafc`, подписана. В данном случае один пост носит название «Суп фо с курятиной», написан человеком по имени Макс, а второй пост под названием «Кофе латте во френч-прессе» написан человеком по имени Глен. На рис. 4.5 показано, что подключено три пользователя, а также посты, которые каждый пользователь недавно опубликовал.

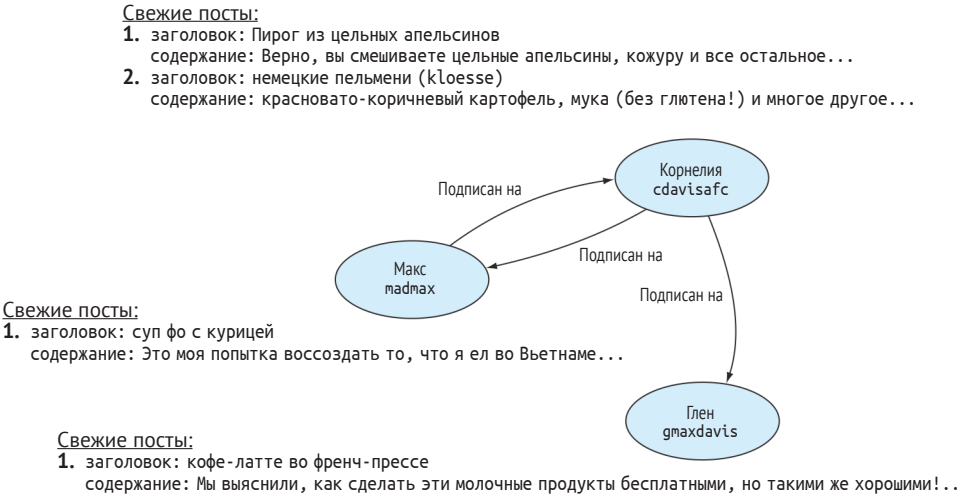


Рис. 4.5 ❖ Пользователи, связи между ними и посты, которые каждый из них недавно опубликовал

Действительно, вы можете увидеть эти данные, вызвав сервис Connections' Posts для каждого из пользователей:

```
curl localhost:8080/connectionsposts/madmax
curl localhost:8080/connectionsposts/gmaxdavis
```

Примечание относительно структуры проекта

Я объединила реализации каждого из трех сервисов в один и тот же JAR-файл, но позвольте мне прояснить кое-что: в любой реальной ситуации это настоятельно не рекомендуется. Одним из преимуществ архитектуры на базе микросервисов является наличие перегородок между сервисами, поэтому сбои в одном из них не переходят на другие. В приведенной здесь реализации таких переборок нет. Если служба Connections даст сбой, с двумя другими службами произойдет то же самое. Я начинаю реализацию с этого антишаблона по двум причинам. Во-первых, он позволяет запустить пример кода с минимальным количеством шагов. Чтобы было проще, я пошла коротким путем. Этот подход также позволит вам ясно увидеть преимущества при рефакторинге реализации путем применения шаблонов, которые тоже обслуживают архитектуры для облачной среды.

Изучаем код

Java-программа, которую вы используете, реализует все три микросервиса, изображенных на рис. 4.4. Я организовала код для реализации в четыре пакета, каждый из которых входит в пакет com.corneliadavis.cloudnative:

- пакет *config* содержит приложение и конфигурацию Spring Boot, а также небольшое количество кода, который заполняет базы данных образцами данных;
- пакет *connections* содержит код микросервиса Connections, в том числе объекты домена, хранилища данных и контроллеры;
- пакет *posts* содержит код микросервиса Posts, включая объекты домена, хранилища данных и контроллеры;
- пакет *connectionposts* содержит код микросервиса Connections' Posts, включая объекты домена и контроллеры (обратите внимание, что хранилища данных нет).

Один пакет объединяет все части в одно приложение Spring Boot и содержит несколько реализаций утилит. Далее у вас идет один пакет для каждого из трех микросервисов, которые образуют ПО.

Микросервисы Connections и Posts имеют сходную структуру. Каждый содержит классы, которые определяют объекты домена службы, а также интерфейсы, используемые реализацией Java Persistence API (JPA) для Spring, чтобы создавать базы данных, в которых хранится контент для объектов каждого типа. В каждом пакете также содержится контроллер, который реализует сервис и основной функционал микросервиса. Эти два микросервиса являются базовыми службами CRUD: они позволяют создавать, считывать, обновлять и удалять объекты, а данные сохраняются в базе данных.

Микросервис, представляющий наибольший интерес в этой реализации, – Connections' Posts, поскольку он не только сохраняет данные в базе данных и извлекает их оттуда, но и рассчитывает составной результат. Глядя на содержимое пакета, можно увидеть, что тут есть только два класса: объект домена с именем `PostSummary` и контроллер.

Класс `PostSummary` определяет объект, содержащий поля для данных, которые будет возвращать служба Connections' Posts: для каждого поста он возвращает заголовок и дату, а также имя того, кто разместил пост. Для этого доменного объекта

нет репозитория JPA, поскольку микросервис использует его только в памяти для хранения результатов вычислений.

Контроллер `ConnectionsPosts` реализует единственный открытый метод – тот, что выполняется, когда к службе делается HTTP-запрос с помощью метода GET. Принимая имя пользователя, реализация запрашивает у службы Connections список пользователей, на которых подписан этот человек, и когда он получает ответ, то делает еще один HTTP-запрос к службе Posts, используя этот набор идентификаторов пользователей. Когда ответ на запрос к Posts получен, приводится составной результат. На рис. 4.6 представлен код этого микросервиса, помеченный шагами, подробно описанными на рис. 4.4.

```
1 @RequestMapping(method = RequestMethod.GET, value="/connectionsposts/{username}")
  public Iterable<PostSummary> getByUsername(
                                  @PathVariable("username") String username,
                                  HttpServletResponse response) {

    ArrayList<PostSummary> postSummaries = new ArrayList<~>();
    logger.info("getting posts for user network " + username);

    String ids = "";
    RestTemplate restTemplate = new RestTemplate();

    // get connections
3   ResponseEntity<Connection[]> respConns
            = restTemplate.getForEntity( url: connectionsUrl+username,  2
                                         Connection[].class);
    Connection[] connections = respConns.getBody();
    for (int i=0; i<connections.length; i++) {
        if (i > 0) ids += ",";
        ids += connections[i].getFollowed().toString();
    }
    logger.info("connections = " + ids);

    // get posts for those connections
5   ResponseEntity<Post[]> respPosts
            = restTemplate.getForEntity( url: postsUrl+ids, Post[].class);  4
    Post[] posts = respPosts.getBody();

    for (int i=0; i<posts.length; i++)
        postSummaries.add(new PostSummary(getUsersname(posts[i].getUserId()),
                                          posts[i].getTitle(),
                                          posts[i].getDate()));
6   return postSummaries;
}
```

Рис. 4.6 ❖ Комплексный результат, сгенерированный микросервисом Connections 'Posts, создается путем вызова микросервисов Connections и Posts и агрегирования результатов

На этапах 2 и 3, а также на этапах 4 и 5 микросервис Connections' Posts действует в качестве клиента для микросервисов Connections и Posts соответственно. Ясно, что у вас есть экземпляры протокола, изображенного на левой стороне рис. 4.2, – ответ/запрос. Однако если приглядеться, то можно увидеть, что есть еще один экземпляр этого шаблона. Составной результат включает в себя имя пользователя, который разместил пост, но эти данные не возвращаются ни из запроса к Connections, ни из запроса к Posts; каждый ответ включает в себя только идентификаторы пользователей. Конечно, это выглядит простовато, но на данный момент эта реализация извлекает имя каждого пользователя для каждого поста, выполняя

набор дополнительных HTTP-запросов к службе Connections. Через мгновение, когда вы перейдете к событийно-ориентированному подходу, вы увидите, что эти экстравызовы исчезнут сами по себе.

Итак, эта базовая реализация работает достаточно хорошо, даже если она может использовать некоторую оптимизацию для повышения эффективности. Но она довольно хрупкая. Чтобы получить результат, микросервис Connections должен быть запущен и работать, как и микросервис Posts. И сеть также должна быть достаточно стабильной для выполнения всех запросов и ответов без всяких проблем. Правильная работа микросервиса Connections 'Posts в значительной степени зависит от многих других вещей, функционирующих правильно; он не контролирует свою судьбу в полной мере.

Событийно-ориентированные архитектуры в значительной степени предназначены для решения проблемы систем, которые слишком тесно связаны. Давайте теперь посмотрим на реализацию, которая удовлетворяет тем же требованиям в микросервисе Connections' Posts, но с другой архитектурой и уровнем устойчивости.

4.2.2. Событийно-ориентированный подход

Вместо того чтобы выполнять код только тогда, когда кто-то или что-то выполняет запрос, в событийно-ориентированной системе код выполняется, когда что-то происходит. То, чем является это «что-то», может сильно различаться и даже может быть запросом пользователя, но основная идея событийно-ориентированных систем заключается в том, что события приводят к выполнению кода, который, в свою очередь, может генерировать события, которые в дальнейшем проходят через систему. Лучший способ понять основы – это конкретный пример, поэтому давайте возьмем ту же проблему, которую мы только что решили, в стиле «запрос/ответ» и реорганизуем ее таким образом, чтобы она была событийно-ориентированной.

Нашей конечной целью по-прежнему является создание списка постов, размещенных людьми, на которых я подписана. Какие события могут повлиять на этот результат в этом контексте? Конечно, если кто-то из моих подписчиков публикует новый пост, его необходимо будет включить в мой список. Но изменения в моей сети также повлияют на результат. Если я добавлю или удалю кого-то из списка лиц, на которых я подписана, или если один из этих людей изменит свое имя пользователя, это также может привести к изменениям в данных, создаваемых службой Connections' Posts.

Конечно, у нас есть микросервисы, которые отвечают за посты и пользовательские подключения; они отслеживают состояние этих объектов. При подходе «запрос/ответ», который вы только что рассмотрели, эти микросервисы управляют состоянием данных объектов и, когда это необходимо, обслуживают это состояние. В нашей новой модели эти микросервисы по-прежнему управляют состоянием этих объектов, но они более активны и генерируют события тогда, когда какое-либо из этих состояний изменяется. Затем, исходя из топологии программного обеспечения, эти события влияют на микросервис Connections' Posts. Эта связь показана на рис. 4.7.

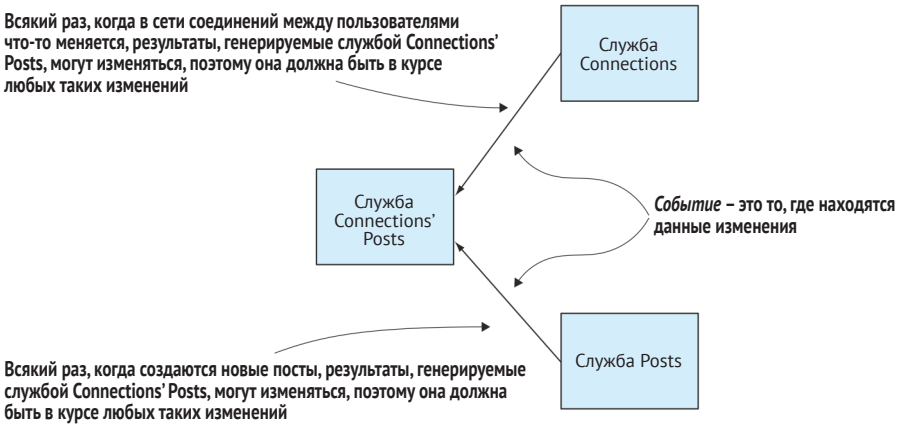


Рис. 4.7 ❖ События – это средство, с помощью которого микросервисы связаны между собой

Конечно, вы уже видели связь между Connections' Posts, Connections и Posts на рис. 4.3 и 4.4, но здесь важно направление стрелок. Как я уже сказала, микросервисы Connections и Posts *заранее* рассылают уведомления об изменениях, не дожидаясь, когда их об этом попросят. Когда служба Connections' Posts запрашивает данные, она *уже знает ответ*. Никаких последующих запросов не требуется.

Я хочу более подробно рассмотреть эти реализации, но сначала давайте посмотрим на код, который реализует этот шаблон. Это поможет вам перейти к принципиально другому образу мышления.

Настройка

Как и в случае со всеми примерами, приведенными в этой книге, вам понадобятся следующие инструменты, установленные на вашей рабочей станции:

- Maven;
- Git;
- Java 1.8.

Получение и создание микросервисов

Клонируйте репозиторий cloudnative-abundantsunshine с помощью приведенной ниже команды, если вы этого еще не сделали:

```
git clone https://github.com/cdavisafc/cloudnative-abundantsunshine.git
cd cloudnative-abundantsunshine
```

Код для этого примера находится в подкаталоге cloudnative-eventdriven, поэтому вам нужно будет перейти туда:

```
cd cloudnative-eventdriven
```

Вскоре вы ознакомитесь с исходным кодом примера, но сначала давайте выполним необходимые приготовления. Приведенная ниже команда создает код:

```
mvn clean install
```

Запуск микросервисов

Как и прежде, вы увидите, что в подкаталоге target был создан новый файл cloudnative-eventdriven-0.0.1-SNAPSHOT.jar. Приложение Spring Boot полностью автономно, включая контейнер Tomcat. Следовательно, чтобы запустить приложение, нужно лишь запустить Java, указав путь к файлу JAR:

```
java -jar target/cloudnative-eventdriven-0.0.1-SNAPSHOT.jar
```

Микросервисы теперь запущены и работают. Как и прежде, я загрузила примеры данных во время запуска, чтобы вы могли в отдельном окне командной строки подключиться к службе Connections' Posts с помощью команды `curl`:

```
curl localhost:8080/connectionsposts/cdavisafc
```

Вы должны увидеть точно такой же вывод, что и при запуске версии приложения «запрос/ответ»:

```
[
  {
    “date”: “2019-01-22T01:06:19.895+0000”,
    “title”: “Chicken Pho”,
    “usersName”: “Max”
  },
  {
    “date”: “2019-01-22T01:06:19.985+0000”,
    “title”: “French Press Lattes”,
    «usersName»: «Glen»
  }
]
```

Если вы ранее не делали это упражнение, ознакомьтесь с разделом 4.3.1, где вы найдете описание примера данных и других конечных точек API, к которым можете отправлять запросы при изучении образца данных. Каждый из трех микросервисов реализует тот же интерфейс, что и прежде; они различаются только в реализации.

Теперь я хочу продемонстрировать, как события, которые мы только что определили (создание новых постов и создание новых подключений), изменят то, что создает микросервис Connections' Posts. Чтобы добавить новый пост, выполните приведенную ниже команду:

```
curl -X POST localhost:8080/posts \
-d ‘{“userId”:2,
     “title”:”Tuna Onigiri”,
     “body”:”Sushi rice, seaweed and tuna. Yum...”}’ \
--header “Content-Type: application/json”
```

Снова выполните команду:

```
curl localhost:8080/connectionsposts/cdavisafc
```

В результате чего вы увидите это:

```
[
  {
    “date”: “2019-01-22T05:36:44.546+0000”,
    “usersName”: “Max”,
    “title”: “Chicken Pho”
  },
  {
    “date”: “2019-01-22T05:41:01.766+0000”,
    “usersName”: “Max”,
    “title”: “Tuna Onigiri”
  },
  {
    “date”: “2019-01-22T05:36:44.648+0000”,
    “usersName”: “Glen”,
    “title”: “French Press Lattes”
  }
]
```

Изучаем код

Это именно то, чего вы ожидали, и те же самые шаги, выполненные для реализации типа «запрос/ответ», дадут только такой результат. В предыдущей реализации, приведенной на рис. 4.6, отчетливо видно, что новый результат генерируется с помощью вызовов, чтобы получить список лиц, на которых я подписана, а затем получить их посты.

Чтобы увидеть, как это работает в событийно-ориентированной реализации, нужно заглянуть в несколько мест. Давайте сначала посмотрим на реализацию агрегатора – Connections’ Posts:

Листинг 4.1 ❖ Метод из ConnectionsPostsController.java

```
@RequestMapping(method = RequestMethod.GET,
                value=”/connectionsposts/{username}”)
public Iterable<PostSummary> getByUsername(
                            @PathVariable(“username”) String username,
                            HttpServletResponse response) {
         Iterable<PostSummary> postSummaries;
         logger.info(“getting posts for user network “ + username);

         postSummaries = mPostRepository.findForUsersConnections(username);

         return postSummaries;
     }
```

Да, вот оно. Это и есть вся реализация. Чтобы сгенерировать результат для микросервиса Connections’ Posts, единственное, что делает метод `getByUsername`, – это выполняет запрос к базе данных. Это возможно благодаря тому, что изменения, влияющие на вывод службы Connections’ Posts, известны ей до того, как будет сделан запрос. Вместо того чтобы ждать запрос к Connections’ Posts, дабы узнать о новых постах, наше программное обеспечение разработано для того, чтобы заранее сообщать о таких изменениях службе Connections’ Posts посредством доставки события.

Я не хочу прямо сейчас вдаваться в детали базы данных, к которой отправляет запрос метод `getByUsername`. Я подробно рассказываю об этом в главе 12, когда говорю о данных для облачной среды. Пока просто знайте, что служба Connections' Posts имеет базу данных, которая хранит состояние, являющееся результатом распространяемых событий. Подведем итог. Метод `get-ByUsername` *возвращает* результат, когда поступает запрос от одностраничного приложения Abundant Sunshine, как показано на рис. 4.8; вы узнаете в нем шаблон, который видите в левой части рис. 4.2. Важно отметить, что служба Connections' Posts может генерировать свой ответ, не завися от других служб.

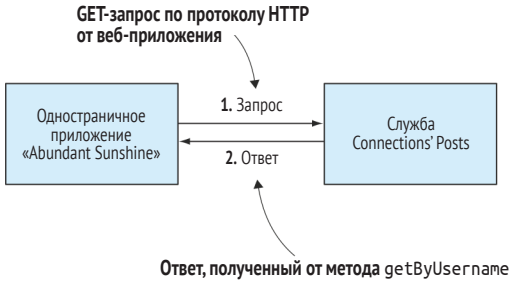


Рис. 4.8 ❖ При получении запроса служба Connections' Posts может генерировать результат, не завися от других служб в системе

Теперь давайте обратимся к событию, которым воспользовалась служба Connections' Posts, начиная с конечной точки службы Posts, которая позволяет создавать новые посты. Метод `newPost` можно найти в пакете com.corneliadavis.cloudnative.posts.write:

Листинг 4.2 ❖ Метод из PostsWriteController.java

```
@RequestMapping(method = RequestMethod.POST, value="/posts")
public void newPost(@RequestBody Post newPost,
                    HttpServletResponse response) {

    logger.info("Have a new post with title " + newPost.getTitle());

    if (newPost.getDate() == null)
        newPost.setDate(new Date());
    postRepository.save(newPost);

    //событие
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<String> resp =
restTemplate.postForEntity("http://localhost:8080/connectionsposts/posts",
                           newPost, String.class);
    logger.info("[Post] resp " + resp.getStatusCode());

}
```

Сначала служба Posts берет событие HTTP POST и сохраняет данные этого поста в репозитории Posts; основная задача службы Posts – реализации операций по созданию и чтению постов в блоге. Но поскольку она является частью событийно-ориентированной системы, она также генерирует событие, чтобы его могли ис-

пользовать любые другие службы, для которых оно актуально. В этом конкретном примере данное событие представлено в виде HTTP POST стороне, которая заинтересована в этом событии, а именно в службе Connections' Posts. Этот фрагмент кода реализует шаблон, показанный на рис. 4.9, в котором вы узнаете экземпляр из правой части рис. 4.2.

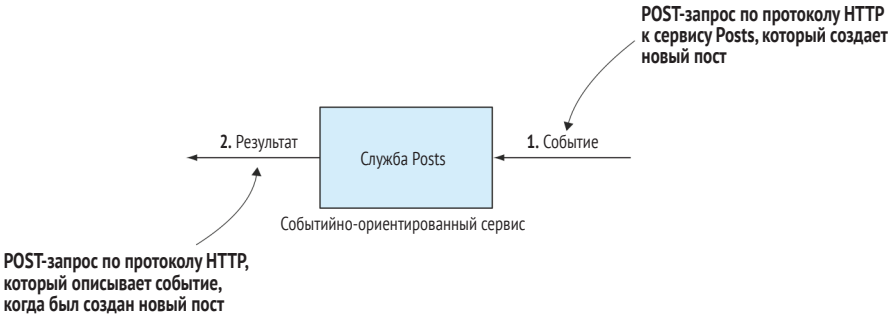


Рис. 4.9 ❖ Служба Posts не только сохраняет созданный новый пост, но также доставляет событие, сообщающее, что этот пост был создан

Обращаясь теперь к получателю этого события, давайте посмотрим на метод `newPost` в службе Connections' Posts; вы найдете его в пакете com.corneliadavis.cloudnative.newpostsfromconnections.eventhandlers:

Листинг 4.3 ❖ Метод из EventsController.java службы Connections' Posts

```
@RequestMapping(method = RequestMethod.POST, value="/posts")
public void newPost(@RequestBody Post newPost, HttpServletResponse
response) {

    logger.info("[NewFromConnections] Have a new post with title "
                + newPost.getTitle());
    MPost mPost = new MPost(newPost.getId(),
                            newPost.getDate(),
                            newPost.getUserId(),
                            newPost.getTitle());
    MUser mUser;
    mUser = mUserRepository.findOne(newPost.getUserId());
    mPost.setmUser(mUser);
    mPostRepository.save(mPost);

}
```

Как видно, этот метод просто принимает событие как тело HTTP POST и сохраняет результат этого события для последующего использования. Помните, что при запросе службе Connections' Posts требуется только выполнить запрос к базе данных, чтобы получить результат. Это код, благодаря которому подобное становится возможным. Вы заметите, что службу Connections' Posts интересует только

несколько полей из получаемого сообщения, идентификатор, дата, идентификатор пользователя и заголовок, которые она сохраняет в локально определенном объекте post. Она также устанавливает правильную связь по внешнему ключу между этим постом и данным пользователем.

Ого, тут много всего. Большинство элементов данного решения подробно рассматривается далее. Например, тот факт, что у каждого микросервиса в этом решении есть свои собственные хранилища данных, а также факт, что Connections' Posts требуется только подмножество контента в событии post, – это сами по себе уже отдельные темы. Но не беспокойтесь сейчас об этих деталях.

На данном этапе я хочу обратить ваше внимание на независимость всех трех микросервисов. Когда вызывается служба Connections' Posts, она не обращается к службе Connections или Posts. Вместо этого она действует самостоятельно; она автономна и будет работать, даже если сетевой раздел отключит Connections и Posts в момент запроса к Connections' Posts.

Я также хочу отметить, что служба Connections' Posts обрабатывает как запросы, так и события. Когда вы использовали команду `curl`, чтобы получить список постов, размещенных отдельными лицами, на которых подписан данный пользователь, служба сгенерировала ответ. Но когда было сгенерировано событие new post, служба обработала это событие без ответа. Вместо этого она сгенерировала только конкретный результат сохранения нового поста в своем локальном репозитории. Никакого дальнейшего события не было. На рис. 4.10 составлены шаблоны, изображенные на рис. 4.2, чтобы показать то, что я только что изложила здесь.

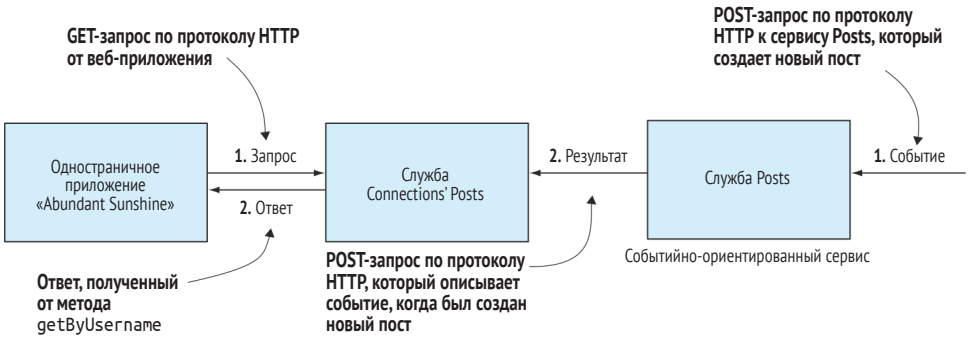


Рис. 4.10 ❖ Микросервисы могут реализовывать шаблон «запрос/ответ» и событийно-ориентированный подход. События в основном используются для слабой связанности микросервисов

На этой диаграмме вы заметите, что две службы слабо связаны. Каждая выполняется автономно. Микросервис Posts делает свое дело каждый раз, когда получает новый пост, генерируя последующее событие. Микросервис Connections' Posts обрабатывает запросы, просто запрашивая свои локальные хранилища данных.

Я попалась!

Вы, наверное, думаете, что мои слова о слабой связи немного преувеличены, и в отношении текущей реализации вы абсолютно правы. Слишком жесткая привязка существует при реализации стрелки с надписью «Результат» на рис. 4.10. В текущей реализации я реализовала это «событие» из микросервиса Posts в виде HTTP POST, который выполняет вызов непосредственно к микросервису Connections' Posts. Это не надежно; если Connections' Posts недоступен, когда Posts выполняет запрос с помощью метода POST, наше событие будет потеряно, а система выйдет из строя.

Если нужно гарантировать, чтобы этот событийно-ориентированный шаблон работал в распределенной системе, для этого требуется больше изощренности, и это именно те методы, которые рассматриваются в оставшейся части данной книги. Пока я реализовала этот пример таким образом только для простоты.

Собрав все части воедино, на рис. 4.11 показана событийно-ориентированная архитектура нашего приложения. Видно, что каждый микросервис работает независимо от других. Обратите внимание на большое количество аннотаций с цифрой «1»; тут мало упорядочения. Как вы видели в примере с Posts, когда происходят события, влияющие на пользователей и подключения, служба Connections делает свою работу, сохраняя данные и генерируя событие для службы Connections' Posts. Когда происходят события, влияющие на результат службы Connections' Posts, обработчик событий выполняет работу по перемещению этих изменений в локальные хранилища данных.

Что особенно интересно, так это то, что работа по сбору данных из обоих источников изменилась. В случае с моделью «запрос/ответ» она реализуется в классе `NewFromConnectionsController`. В случае с событийно-ориентированным подходом она реализуется через генерацию событий и реализацию обработчиков событий.

Код, соответствующий аннотациям для микросервисов на рис. 4.11, появляется на трех приведенных ниже рисунках. На рис. 4.12 для службы Connections изображен `ConnectionsWriteController`. На рис. 4.13 для службы Posts показан `PostsWriteController`. На рис. 4.14 для службы Connections' Posts приведен `EventsController` (обработчик событий). И наконец, на рис. 4.15 служба Connections' Posts отвечает на запросы в `ConnectionsPostsController`.

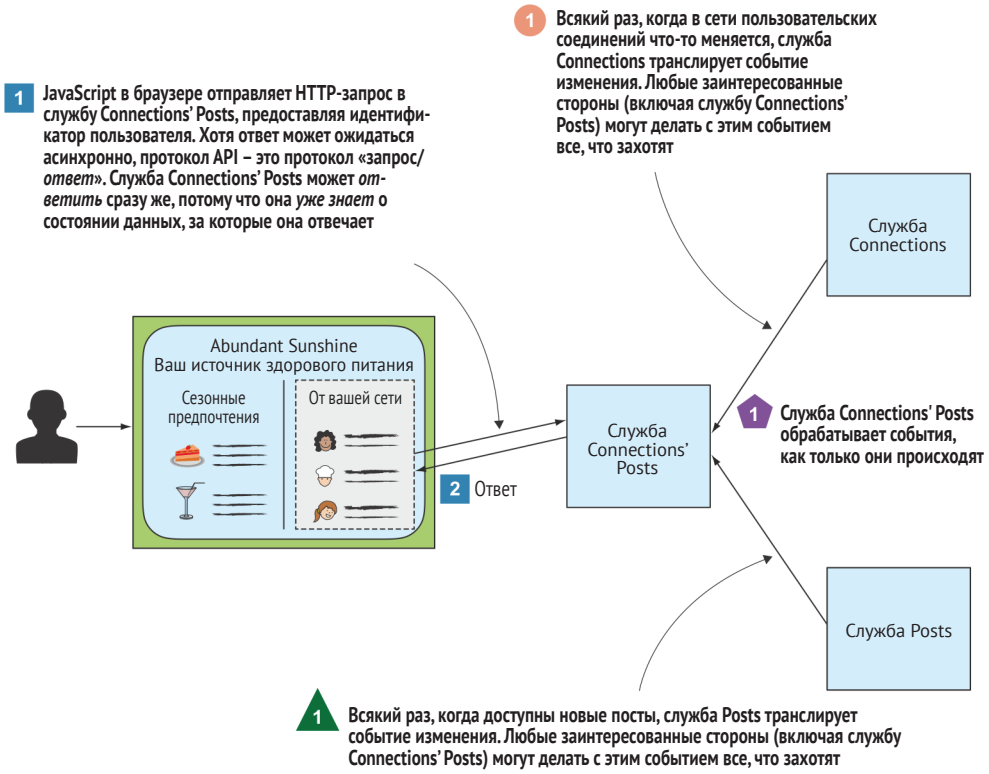


Рис. 4.11 ❖ Рендеринг веб-страницы теперь требует только выполнения микросервиса Connections 'Posts. Благодаря обработке событий у него уже есть все данные, необходимые для удовлетворения запроса, поэтому никакие последующие запросы и ответы не требуются

```
@RequestMapping(method = RequestMethod.POST, value="/connections")
public void newConnection(@RequestBody Connection newConnection, HttpServletResponse response) {

    logger.info("Have a new connection: " + newConnection.getFollower() +
                " is following " + newConnection.getFollowed());
    connectionRepository.save(newConnection);

    //event
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<String> resp = restTemplate.postForEntity( 1
            url: "http://localhost:8080/connectionsposts/connections", newConnection, String.class);
    logger.info("resp " + resp.getStatusCode());
}
```

Рис. 4.12 ❖ Служба Connections генерирует событие, когда было записано новое соединение

```
@RequestMapping(method = RequestMethod.POST, value="/posts")
public void newPost(@RequestBody Post newPost, HttpServletResponse response) {

    logger.info("Have a new post with title " + newPost.getTitle());

    if (newPost.getDate() == null)
        newPost.setDate(new Date());
    postRepository.save(newPost);

    //event
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<String> resp = restTemplate.postForEntity(
            url: "http://localhost:8080/connectionsposts/posts", newPost, String.class);
    logger.info("[Post] resp " + resp.getStatusCode());
}
```

1

Рис. 4.13 ❖ Служба Posts генерирует событие, когда был записан новый пост

```
1 @RequestMapping(method = RequestMethod.POST, value="/users")
public void newUser(@RequestBody User newUser, HttpServletResponse response) {

    logger.info("[NewPosts] Creating new user with username " + newUser.getUsername());
    mUserRepository.save(new MUser(newUser.getId(), newUser.getName(), newUser.getUsername()));
}

1 @RequestMapping(method = RequestMethod.PUT, value="/users/{id}")
public void updateUser(@PathVariable("id") Long userId,
                       @RequestBody User newUser, HttpServletResponse response) {

    logger.info("Updating user with id " + userId);
    MUser mUser = mUserRepository.findById(userId).get();
    mUserRepository.save(mUser);

}

1 @RequestMapping(method = RequestMethod.POST, value="/connections")
public void newConnection(@RequestBody Connection newConnection, HttpServletResponse response) {

    logger.info("Have a new connection: " + newConnection.getFollower() +
            " is following " + newConnection.getFollowed());
    MConnection mConnection = new MConnection(newConnection.getId(), newConnection.getFollower(),
                                              newConnection.getFollowed());
    // add connection to the users
    MUser mUser;
    mUser = mUserRepository.findById(newConnection.getFollower()).get();
    mConnection.setFollowerUser(mUser);
    mUser = mUserRepository.findById(newConnection.getFollowed()).get();
    mConnection.setFollowedUser(mUser);
    mConnectionRepository.save(mConnection);
}

1 @RequestMapping(method = RequestMethod.DELETE, value="/connections/{id}")
public void deleteConnection(@PathVariable("id") Long connectionId, HttpServletResponse response) {

    MConnection mConnection = mConnectionRepository.findById(connectionId).get();

    logger.info("deleting connection: " + mConnection.getFollower() +
            " is no longer following " + mConnection.getFollowed());
    mConnectionRepository.delete(mConnection);

}

1 @RequestMapping(method = RequestMethod.POST, value="/posts")
public void newPost(@RequestBody Post newPost, HttpServletResponse response) {

    logger.info("Have a new post with title " + newPost.getTitle());
    MPost mPost = new MPost(newPost.getId(), newPost.getDate(), newPost.getUserId(), newPost.getTitle());
    MUser mUser;
    mUser = mUserRepository.findById(newPost.getUserId()).get();
    mPost.setmUser(mUser);
    mPostRepository.save(mPost);

}
```

Рис. 4.14 ❖ Обработчик событий службы Connections' Posts обрабатывает события по мере их возникновения

```
@RequestMapping(method = RequestMethod.GET, value="/connectionsposts/{username}")
public Iterable<PostSummary> getByUsername(@PathVariable("username") String username,
                                           HttpServletResponse response) {

    Iterable<PostSummary> postSummaries;
    logger.info("getting posts for user network " + username);

    postSummaries = mPostRepository.findForUsersConnections(username);

    return postSummaries;
}
```

Рис. 4.15 ❖ Служба Connections' Posts генерирует и доставляет ответ при получении запроса. Она полностью независима от операций других микросервисов в нашем ПО

Хотя интересно посмотреть, как обработка, в ходе которой в конечном итоге генерируется результат Connections' Posts, распределяется по микросервисам, еще более важными являются временные аспекты. При использовании стиля «запрос/ответ» сбор данных происходит, когда пользователь делает запрос. В случае с событийно-ориентированным подходом это происходит независимо от того, когда меняются данные в системе; это асинхронный процесс. Как вы убедитесь, асинхронность важна для распределенных систем.

4.4 Знакомство с шаблоном Command Query Responsibility Segregation

Я хочу рассмотреть еще один аспект кода для этого примера, начиная со служб Posts и Connections. По сути, это CRUD-сервисы; они позволяют *создавать, обновлять* и *удалять* сообщения, пользователей и соединения и, конечно же, *считывать* значения. База данных хранит состояние службы, а реализация службы RESTful поддерживает HTTP-запросы с использованием методов GET, PUT, POST и DELETE, которые, по сути, взаимодействуют с этим хранилищем данных.

При реализации типа «запрос/ответ» весь этот функционал находится в одном контроллере; например, в случае со службой Posts это `PostsController`. Но в событийно-ориентированной реализации видно, что в пакете com .corneliadavis. cloudnative.posts теперь есть контроллер чтения, а в пакете com.corneliadavis. cloudnative.posts.write есть контроллер записи. По большей части, у этих двух контроллеров есть методы для реализации того, что ранее находилось в единственном контроллере Posts, за исключением того, что я добавила доставку событий для любых операций с изменением состояния. Но вам, наверное, интересно, почему я занялась этим, разбив тело кода на две части.

По правде говоря, в этом простом примере – службы Posts и Connections – такое разделение мало что значит. Но даже для чуть более сложных сервисов отделение чтения от записи позволяет лучше контролировать дизайн сервиса. Если вы знакомы с шаблоном Model, View, Controller (MVC), который широко использовался на протяжении последних нескольких десятилетий, то знаете, что контроллер, который является бизнес-логикой сервиса (вот почему вы встречаете классы … `Controller` во всех моих реализациях), работает не в пользу модели. При наличии одного контроллера модель операций чтения и записи одна и та же, но при раз-

делении бизнес-логики на два отдельных контроллера каждый может иметь свою собственную модель, что может дать мощные результаты.

Нашим сервисам даже не нужно быть сложными, чтобы вы могли увидеть эту мощь. Рассмотрим сценарий с автомобилем, имеющим выход в интернет, в котором датчики собирают данные ежесекундно. Эти данные будут включать в себя временную метку и координаты GPS – широту и долготу, и сервис позволяет сохранять эти значения. Модель этого сервиса включает в себя поля для этих данных и многое другое: временную метку, широту и долготу. Одна из вещей, которую вам нужно поддерживать, – это возможность доступа к данным о скорости во время поездок. Например, возможно, вам нужно проанализировать сегменты поездки, где скорость превышала 50 миль в час. Эти данные не передаются напрямую через контроллер записи, но очевидно, что их можно вычислить на основе предоставленных данных; часть бизнес-логики этого сервиса заключается в создании этих производных данных.

Понятно, что модели для чтения и записи разные. Хотя и там, и там есть определенные поля, существуют поля, которые имеют смысл только для одной или другой стороны (рис. 4.16). Это одно из первых преимуществ наличия двух отдельных моделей; это позволяет вам писать код, который легче понять и поддерживать, и значительно уменьшает область поверхности, в которую могут закрасться ошибки.

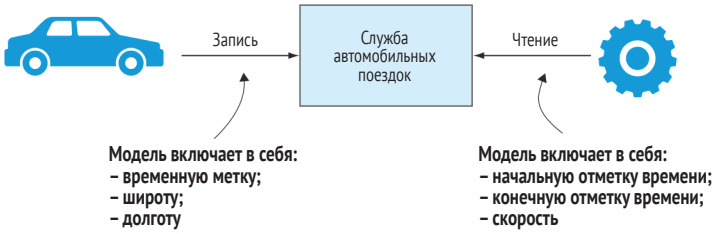


Рис. 4.16 ❖ Отделение логики записи от логики чтения позволяет поддерживать разные модели этих разных задач, что приводит к более изящному коду, удобному в сопровождении

Итак, что вы делаете: вы отделяете логику записи (команды) от логики чтения (запросы). В этом состоит основа шаблона *Command Query Responsibility Segregation* (CQRS). По сути, CQRS как раз разделяет две эти проблемы. Этот шаблон может дать большое количество преимуществ. Предыдущий пример более элегантного кода – лишь одно из них. Еще одно преимущество начинает проявлять себя при событийно-ориентированной реализации нашего примера.

Например, в службе Posts код, который был распределен между контроллерами чтения и записи в событийно-ориентированной реализации, по большей части находился в одном контроллере, относящемся к варианту типа «запрос/ответ». Но если вы обратитесь к реализации службы Connections' Posts, то там будут существенные различия.

При событийно-ориентированном подходе реализация запроса находится в пакете com.corneliadavis.cloudnative.connectionsposts, а код для реализации команды – в пакете com.corneliadavis.cloudnative.connectionsposts.eventhandlers.

Сравнивая это с реализацией типа «запрос/ответ», первое, на что следует обратить внимание, – это то, что в случае с типом «запрос/ ответ» отсутствует код команды.

Поскольку служба Connections' Posts абсолютно не сохраняла текущее состояние, в результате чего было получено агрегирование данных из двух других служб, реализация команд не требовалась. Как вы убедились, при событийно-ориентированном подходе реализация запроса значительно упрощена; вся логика агрегирования исчезла. Агрегирование теперь появляется, хотя и в совершенно другой форме, в контроллере команд. Если в случае «запрос/ответ» вы видели вызовы нисходящих ресурсов, пользователей, соединений и постов, в этом контроллере команд вы теперь видите обработчики событий, которые имеют дело с изменениями тех же объектов.

Эти команды меняют состояние, и хотя я не буду здесь подробно рассказывать об этом хранилище данных (это будет в главе 12), интересно отметить и другие реализации шаблона CQRS. Пусть моя текущая реализация процессоров событий, составляющих командную часть службы Connections' Posts, реализована в виде конечных точек HTTP, как и сторона запроса, отделение запроса от обработки команд в конечном итоге позволит нам использовать разные протоколы для обеих сторон. На стороне запроса в этом случае реализация REST через протокол HTTP идеально подходит для веб-приложений, которые будут получать доступ к этой службе. Однако на стороне команд, возможно, лучше подойдет асинхронная реализация или даже реализация шаблона FaaS (функция как услуга)[1]. Отделение команд от запросов обеспечивает такую гибкость. (Те из вас, кто склонен забегать вперёд, могут посмотреть на реализацию обработчика событий для службы Connections' Posts в модуле cloudnative-eventlog нашего репозитория. Вы увидите, что поход на базе HTTP, который вы видите здесь, был заменен другим подходом.)

Я хотела бы поделиться одним заключительным замечанием по данной теме. Я обнаружила, что CQRS часто сопоставляют с событийно-ориентированными системами. Если в программном обеспечении используется событийно-ориентированный подход, то можно использовать шаблоны CQRS, но в противном случае CQRS не стоит рассматривать. Я рекомендую вам рассматривать CQRS независимо от событийно-ориентированных систем. Да, они дополняют друг друга, и их совместное использование дает большие преимущества, но отделение логики команд от логики запросов также применимо в проектах, которые не являются событийно-ориентированными.

4.5. Разные стили, схожие проблемы

Эти две реализации дают абсолютно одинаковый результат – «счастливый путь». Выбор использования стиля «запрос/ответ» или событийно-ориентированного подхода является произвольным, если выполняются все приведенные ниже условия:

- сетевые разделы не отключают один микросервис от другого;
- при составлении списка лиц, на которых я подписана, не возникает неожиданной задержки;

[1] FaaS – это подход, который ускоряет вычисления для некоторой части логики, такой как обновление записи в базе данных, по запросу и хорошо подходит для событийно-ориентированных проектов.

- все контейнеры, в которых работают мои службы, поддерживают стабильные IP-адреса;
- мне не нужно переключаться между наборами учетных данных или сертификатами.

Но эти и многие другие условия – это именно то, что характеризует облако. Программное обеспечение для облачной среды предназначено для получения необходимых результатов, даже когда сеть нестабильна, некоторые компоненты внезапно требуют больше времени для получения результатов, хосты и зоны доступности исчезают, а объемы запросов внезапно увеличиваются на порядок.

В этом контексте событийно-ориентированный подход и подход типа «запрос/ответ» могут давать очень разные результаты. Но я не утверждаю, что одно всегда лучше другого. Оба этих подхода являются действительными и применимыми. Однако вы должны применять нужные шаблоны в нужное время, чтобы компенсировать особые проблемы, которые приходят с облаком.

Например, чтобы быть готовым к перепадам в объеме запросов, вы делаете так, чтобы у вас была возможность масштабировать емкость, создавая или удаляя экземпляры своих служб. Наличие этих экземпляров также обеспечивает определенный уровень устойчивости, особенно когда они распределены по областям отказов (зоны доступности). Но когда вам необходимо применить новую конфигурацию к сотням экземпляров микросервиса, вам нужен какой-то тип службы конфигурации, который отвечает за их сильно распределенное развертывание. Эти и многие другие шаблоны в равной степени применимы к микросервисам независимо от того, реализуют ли они протокол типа «запрос/ответ» или событийно-ориентированный протокол.

Но некоторые проблемы можно обрабатывать по-разному в зависимости от этого протокола. Например, какой тип компенсирующих механизмов нужно установить для учета кратковременного (это может быть меньше секунды или может быть в последнюю минуту) разделения связности сети, при котором связанные микросервисы отрезаются друг от друга? Нынешняя реализация службы Connections' Posts потерпит неудачу, если она не сможет добраться до микросервисов Connections или Posts. Ключевым шаблоном, используемым при таком типе сценария, является *повторная попытка*: клиент, выполняющий запрос к службе через сеть, попытается сделать это снова, если запрос, который он сделал, не даст результатов. Повторная попытка – это способ сгладить сбои в сети, позволяя протоколу вызова оставаться синхронным.

С другой стороны, учитывая, что событийно-ориентированный протокол по своей природе является асинхронным, механизмы компенсации для устранения одних и тех же опасностей могут быть совершенно разными. В этой архитектуре вы будете использовать систему обмена сообщениями, такую как RabbitMQ или Apache Kafka, чтобы удерживать события через сетевые разделы. Ваши сервисы будут реализовывать протоколы для поддержки этой архитектуры, такие как наличие цикла управления, который выполняет непрерывную проверку на предмет новых событий, представляющих интерес, в хранилище событий. Вспомните HTTP-запрос с помощью метода POST из микросервиса Posts непосредственно к обработчику событий службы Connections' Posts: вы используете систему обмена сообщениями для замены этой жесткой связи. На рис. 4.17 показаны различия в шаблонах, используемых для обработки этой черты распределенных систем.

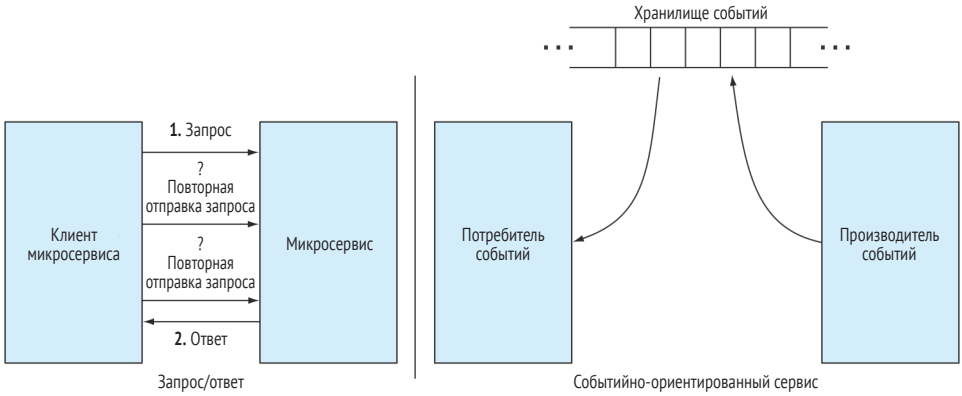


Рис. 4.17 ❖ Повторная отправка запроса – это ключевой шаблон, используемый в микросервисных архитектурах типа «запрос/ответ» для компенсации нарушения связности сети. В событийно-ориентированых системах использование хранилища событий является ключевым методом, который компенсирует неустойчивость сети

В оставшейся части книги мы подробно рассмотрим эти шаблоны, всегда делая акцент на проблемах, которые приходят с облачным контекстом. Выбор одного из стилей вызова в пользу другого не является решением само по себе. Данный выбор наряду с шаблонами, которые их дополняют, – это решение. Я научу вас, как выбирать правильный протокол и применять дополнительные шаблоны для его поддержки.

Резюме

- Подход типа «запрос/ответ» и событийно-ориентированный подход используются для соединения компонентов, составляющих программное обеспечение для облачной среды.
- Микросервис может реализовывать оба этих протокола.
- В идеальных стабильных условиях программное обеспечение, реализованное с использованием одного подхода, может дать те же результаты, что и программное обеспечение, реализованное с использованием другого подхода.
- Но в облаке, где решением является распределенная система, а среда постоянно меняется, результаты могут сильно отличаться.
- Некоторые архитектурные шаблоны в равной степени применимы к программному обеспечению для облачной среды в соответствии со стилем типа «запрос/ответ» и событийно-ориентированным стилем.
- Но другие шаблоны специально используются для того или иного протокола вызова.
- CQRS играет важную роль в событийно-ориентированных системах.

Глава 5

Избыточность приложения: горизонтальное масштабирование и отсутствие фиксации состояния

О чем рассказывается в этой главе:

- горизонтальное масштабирование как центральный принцип приложений для облачной среды;
- подводные камни приложений с фиксацией текущего состояния в программном обеспечении для облачной среды;
- что значит для приложения не фиксировать текущее состояние;
- службы с фиксацией текущего состояния, и как они используются приложениями, не фиксирующими его;
- почему не стоит использовать «липкие» сессии.

В заголовке присутствует слово «масштабирование», но на самом деле речь здесь идет не просто о масштабировании. Существует множество причин для того, что, вероятно, является основным принципом программного обеспечения для облачной среды: избыточность. Независимо от того, являются ли компоненты вашего приложения микро- или макрокомандой, можно ли настроить их с помощью переменных среды или встроить конфигурацию в файлы свойств, независимо от того, полностью ли они реализуют варианты альтернативного поведения или нет, ключом к устойчивости к изменениям является тот факт, что единой точки отказа не существует. У приложений всегда есть несколько развернутых экземпляров.

Но тогда, поскольку любой из этих экземпляров приложения может удовлетворить запрос (и вы вскоре увидите, почему *это* важно), вам нужно, чтобы эти экземпляры вели себя как единая логическая сущность. На рис. 5.1 это ясно показано.

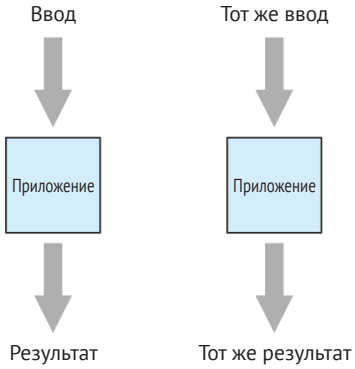


Рис. 5.1 ❖ При одинаковом вводе результат, создаваемый экземпляром приложения, должен быть одинаковым независимо от того, существует ли один, два или сто экземпляров

Хотя это выглядит довольно просто, все может быть немного сложнее, потому что контекст, в котором работает каждый экземпляр приложения, будет разным. Входные данные, поступающие с вызовом приложения (здесь я не имею в виду какой-либо конкретный шаблон вызова – это может быть «запрос/ответ» или событийно-ориентированный подход), – не единственное, что влияет на результат. Каждый экземпляр приложения будет работать в своем собственном контейнере – это может быть виртуальная машина Java, хост (виртуальный или физический) или контейнер Docker (или что-то аналогичное), – и значения среды в этом контексте могут влиять на выполнение приложения. Значения конфигурации приложения также будут предоставляться каждому экземпляру приложения. И в этой главе основной интерес для нас представляет история взаимодействия пользователя с приложением.

Рисунок 5.2 охватывает контекст вокруг экземпляров приложения и показывает проблемы, связанные с достижением паритета во внешних воздействиях на приложение.

Глава 6 посвящена тому, что оказывает влияние на системную среду и конфигурацию приложения. В этой главе я расскажу об истории запросов.

Я начну с того, что расскажу о преимуществах развертывания нескольких экземпляров приложения, и вы сразу увидите, что происходит, когда это пересекается с приложениями, фиксирующими текущее состояние. Я делаю это в контексте примера с поваренной книгой, с которого мы начали в предыдущей главе, разбивая этот монолит на отдельные микросервисы, которые затем развертываются и управляются независимо друг от друга. Я ввела локальное состояние в один из этих микросервисов, а именно хранение маркеров аутентификации. Да, «липкие» сессии – это распространенный шаблон, когда речь идет о таком типе сохранения состояния сеанса, но для облака это не подходит, и я расскажу, почему. Я также ввожу понятие *службы с фиксацией текущего состояния*. Это особый тип службы, тщательно разработанной для управления сложностями состояния. И наконец, я покажу вам, как отделить части вашего программного обеспечения с фиксацией текущего состояния от частей, не являющихся таковыми.

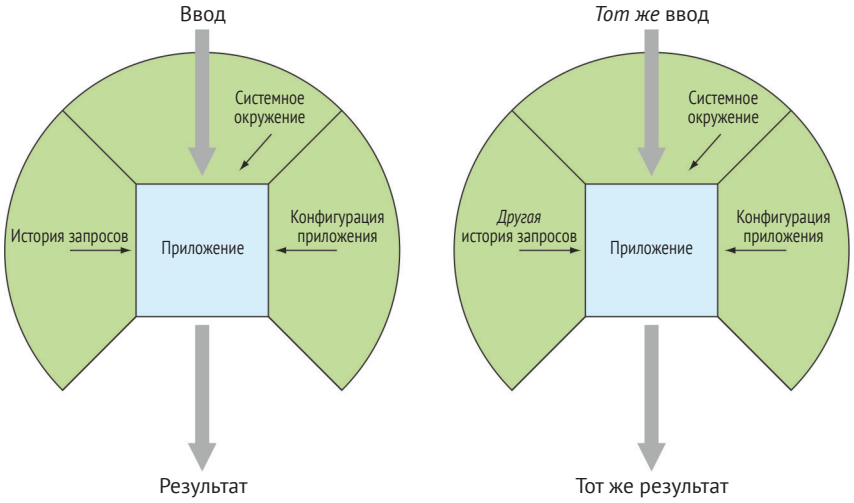


Рис. 5.2 ❖ Приложение для облачной среды должно обеспечивать согласованные результаты, несмотря на возможные различия в других контекстных факторах

5.1. У приложений для облачной среды есть много развернутых экземпляров

В облаке преобладающей моделью увеличения или уменьшения емкости приложений для обработки изменяющихся объемов запросов является горизонтальное масштабирование. Вместо добавления или уменьшения емкости для одного экземпляра приложения (вертикальное масштабирование) растущие или падающие объемы запросов обрабатываются путем добавления или удаления экземпляров приложения.

Это не означает, что экземпляры приложений нельзя распределить с существенными вычислительными ресурсами; Google Cloud Platform (GCP) сейчас предлагает тип машины с 1,5 Тб памяти, а AWS предлагает машину с почти 2 Тб. Но изменение характеристик компьютера, на котором работает приложение, – существенное событие. Скажем, например, вы посчитали, что 16 Гб памяти достаточно для вашего приложения, и в течение некоторого времени все работает нормально. Но затем объем ваших запросов возрастает, и теперь вы хотели бы увеличить эту цифру. В облачных средах, таких как AWS, Azure или GCP, нельзя изменить тип машины для работающего хоста. Вместо этого вам придется создать новую машину с 32 Гб памяти (даже если вам нужно всего около 20 Гб, потому что типа машины с 20 Гб памяти не существует), развернуть там свое приложение, а затем выяснить, как перенести пользователей на новые экземпляры с минимальными перебоями.

Сравните это с моделью горизонтального масштабирования. Вместо того чтобы предоставлять своему приложению 16 Гб памяти, вы даете четыре экземпляра с 4 Гб памяти для каждого. Когда вам нужно больше емкости, вы просто запрашиваете пятый экземпляр приложения, делаете его доступным, например регистрируя его с помощью динамического маршрутизатора, и теперь общий объем па-

мяти составляет 20 Гб. Вам не только предоставляется более точный контроль над потреблением ресурсов, но и порядок достижения большего масштаба становится гораздо проще.

Однако гибкое масштабирование – не единственная мотивация для использования нескольких экземпляров приложения. Это также и высокая доступность, надежность и эффективность работы. Возвращаясь к первому примеру в этой книге, в гипотетическом сценарии, изображенном на рис. 1.2 и 1.3, именно несколько экземпляров приложения позволило Netflix оставаться работоспособным даже после сбоя инфраструктуры AWS. Ясно, что если вы развернете свое приложение как один объект, это будет единственная точка отказа.

Наличие нескольких экземпляров также приносит выгоду, когда дело доходит до эксплуатации программного обеспечения в рабочем окружении. Например, приложения все чаще запускаются на платформах, которые предоставляют набор услуг сверх необработанных вычислений, хранилищ и организации сетей. Например, команды приложений (dev и ops) больше не должны предоставлять свою собственную операционную систему. Вместо этого они могут просто отправить свой код на платформу, а она создаст среду выполнения и развернет приложение. Если платформе (которая с точки зрения приложения является частью инфраструктуры) необходимо обновить ОС, в идеале приложение должно оставаться работающим на протяжении всего процесса обновления. Пока хост обновляется на ОС, выполняемые на нем рабочие нагрузки должны быть остановлены, но если у вас есть другие экземпляры приложений, работающие на других хостах (вспомните обсуждение в главе 3 относительно распределения экземпляров приложения), вы можете просматривать свои хосты по одному. Пока один экземпляр приложения переводится в автономный режим, другие экземпляры приложения по-прежнему обслуживают трафик.

Наконец, когда вы объединяете воедино горизонтальное масштабирование приложений и разложенную на части архитектуру программного обеспечения на базе микросервисов, то получаете большую гибкость при общем потреблении ресурсов системы. Как продемонстрировало программное обеспечение кулинарного сообщества, с которым вы познакомились в предыдущей главе, наличие отдельных компонентов приложения позволяет масштабировать API службы Posts, который вызывается намного большим количеством клиентов, чем те, что были описаны в нашем сегодняшнем сценарии, для обработки больших объемов трафика, в то время как другие приложения, такие как API Connections, имеют меньше экземпляров, обрабатывающих гораздо меньшие объемы. Посмотрите на рис. 5.3.

5.2. Приложения с фиксацией текущего состояния в облаке

Как видно, потребность в таких вещах, как гибкое масштабирование, отказоустойчивость и эксплуатационная эффективность, приводит к тому, что несколько экземпляров приложения работает как часть вашего ПО. Эти же цели, а также архитектура с несколькими экземплярами, которую я только что описала, тесно связаны с фиксацией или отсутствием фиксации текущего состояния ваших приложений. Но вместо того, чтобы обсуждать эти элементы только в теории, для начала давайте обратимся к конкретному примеру.

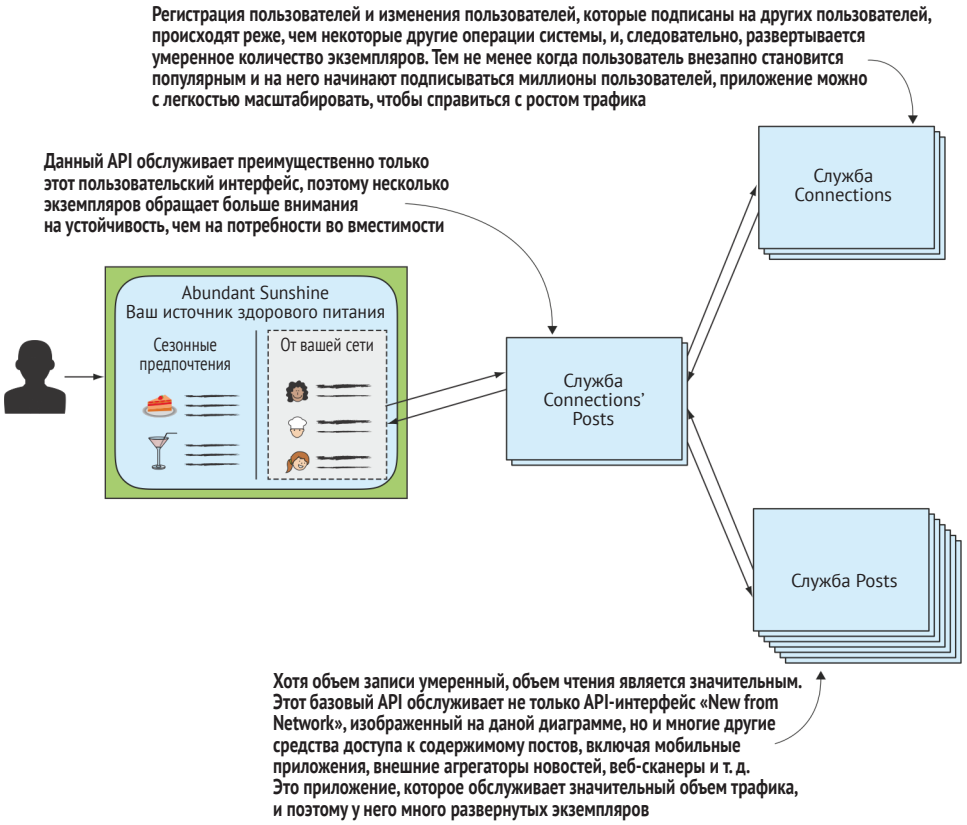


Рис. 5.3 ❖ Архитектура приложений для развертывания нескольких экземпляров обеспечивает значительный выигрыш в эффективности использования ресурсов, а также в устойчивости и других эксплуатационных преимуществах

Я начну с приложения из предыдущей главы, в частности с того, что было написано в стиле «запрос/ответ». Эта реализация в значительной степени отслеживает состояние, поскольку она даже хранит все данные приложения, пользователей, соединения и посты в базе данных в оперативной памяти. Кроме того, это монолитное приложение. Службы Connections' Posts, Connections и Posts являются частью одного проекта и в конечном итоге скомпилированы в один файл JAR (архив Java). Для начала мы исправим это.

Мы найдем исходный код для данного примера в облачном репозитории, в частности в каталоге/модуле cloud-native-statelessness, и начнем с реализации, которая вначале демонстрирует недостатки переноса состояния в ваших приложениях, а позже создадим решение. После клонирования репозитория, пожалуйста, скачайте определенный тег и перейдите в каталог cloudnative-statelessness:

```
git clone https://github.com/cdavisafc/cloudnative-abundantsunshine.git
git checkout statelessness/0.0.1
cd cloudnative-statelessness
```

5.2.1. Разложение монолита на части и привязка к базе данных

Давайте сначала посмотрим, как я разбила прежде монолитное приложение на три отдельных сервиса. В каталоге cloudnative-statelessness теперь содержится только файл pom.xml и подкаталог/подмодуль для каждого из трех микросервисов. Два микросервиса, cloudnative-posts и cloudnative-connections, полностью автономны. Они не зависят от других микросервисов.

Третий микросервис, приложение cloudnative-connectionsposts, также в основном отключен от двух других, при этом единственное реальное указание на какую-либо зависимость наблюдается в файле application.properties:

```
management.endpoints.web.exposure.include=*
connectionpostscontroller.connectionsUrl=http://localhost:8082/connections/
connectionpostscontroller.postsUrl=http://localhost:8081/posts?userIds=
connectionpostscontroller.usersUrl=http://localhost:8082/users/
INSTANCE_IP=127.0.0.1
INSTANCE_PORT=8080
```

Напомним, что это приложение отправляет запрос в службу *Connections* для получения списка лиц, на которых подписан конкретный пользователь, а затем отправляет запрос в службу *Posts* для получения постов, размещенных любым из этих лиц. URL-адреса в этом приложении настроены таким образом, чтобы облегчить доступ к данным службам по протоколу HTTP. (Обратите внимание, что настройка этих URL-адресов в файле application.properties представляет собой облачный антишаблон. Мы исправим это в следующей главе.)

Возвращаясь теперь к хранилищу данных пользователя, подключений и постов, независимо от того, запускаете ли вы это приложение в облаке или нет, почти наверняка необходимо, чтобы данные приложения не просто находились в памяти, а хранились в каком-то месте на постоянном диске. То, что я прежде хранила их только в базе данных H2, связано с тем, что это просто удобно; постоянное хранение данных не было уместным в предыдущей главе. В конечном счете мы также должны позаботиться об устойчивости этих постоянных данных, и скоро я расскажу об этом более подробно. Я добавила зависимость от базы данных MySQL в файлы pom.xml приложений Connections и Posts. Файл POM теперь включает в себя обе эти зависимости:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Зависимость H2 присутствовала ранее; зависимость MySQL – новая. Я сохранила зависимость H2 прежде всего для того, чтобы позволить использовать ее в тестировании. Если при запуске предоставляется URL-адрес MySQL, Spring Boot JPA создаст и настроит MySQL-клиента. В противном случае он будет использовать H2.

Давайте приведем этот код в действие.

Настройка

Как и в примерах, приведенных в предыдущей главе, для запуска у вас должны быть установлены стандартные инструменты; последние два в этом списке – новые:

- Maven;
- Git;
- Java 1.8;
- Docker;
- какой-нибудь MySQL-клиент, например интерфейс командной строки `mysql` (CLI);
- какой-нибудь Redis-клиент, например `redis-cli`.

Создание микросервисов

В каталоге cloudnative-statelessness введите следующую команду:

```
mvn clean install
```

После выполнения этой команды будет создано каждое из трех приложений с файлом с расширением JAR в целевом каталоге каждого модуля.

Запуск приложений

Перед запуском любого из микросервисов необходимо запустить службу MySQL и создать базу данных поваренной книги. Чтобы запустить службу MySQL, мы воспользуемся Docker. Предполагая, что у вас уже установлен Docker, вы можете сделать это с помощью приведенной ниже команды:

```
docker run --name mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=password \
 -d mysql:5.7.22
```

Чтобы создать базу данных, можно подключиться к своему серверу MySQL через клиентский инструмент по вашему выбору. Используя интерфейс командной строки `mysql`, можно ввести это:

```
mysql -h 127.0.0.1 -P 3306 -u root –p
```

Затем введите пароль, `password`. В командной строке MySQL можно выполнить следующую команду:

```
mysql> create database cookbook;
```

Теперь вы готовы к запуску приложений – во множественном числе. Это первое, на что нужно обратить внимание: поскольку вы разделили программное обеспечение на три независимых микросервиса, вам нужно запустить три разных файла JAR. Вы будете запускать каждое приложение локально, поэтому каждый сервер приложений Spring Boot (по умолчанию Tomcat) должен запускаться на отдельном порте. Вы это сделаете, а потом, в случае со службами Posts and Connections, предоставите URL-адрес службы MySQL в командной строке. Таким образом, в трех окнах терминала вы выполните следующие три команды:

```
java -Dserver.port=8081 \
-Dspring.datasource.url=jdbc:mysql://localhost:3306/cookbook \
-jar cloudnative-posts/target/cloudnative-posts-0.0.1-SNAPSHOT.jar
```

```
java -Dserver.port=8082 \
-Dspring.datasource.url=jdbc:mysql://localhost:3306/cookbook \
-jar cloudnative-connections/target/cloudnative-connections-0.0.1-SNAPSHOT.jar

java -jar cloudnative-connectionposts/target/cloudnative-connectionposts-0.0.1-SNAPSHOT.jar
```

Мне нравится настраивать свой терминал, как показано на рис. 5.4: я нахожусь в каталоге cloudnative-statelessness во всех окнах. Такое расположение позволяет мне просматривать базу данных в крайнем правом углу, просматривать вывод журнала для каждого из трех микросервисов сверху (я выполняю команды Java, которые я только что дала вам в каждом из этих трех окон) и выполнять команды `curl`, чтобы протестировать свои службы в большем окне в левом нижнем углу.

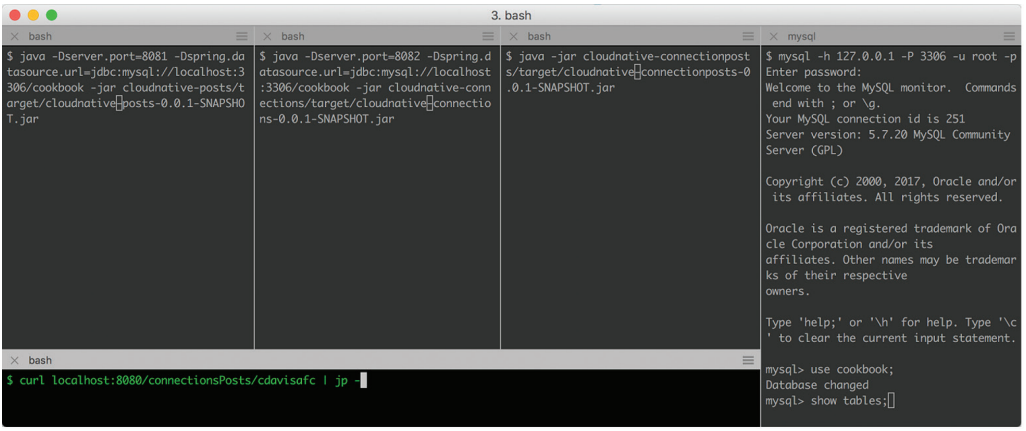


Рис. 5.4 ❖ Конфигурация моего терминала позволяет мне отправлять запросы микросервисам, наблюдая за результатами в других окнах

Теперь вы можете выполнить следующие команды `curl` для проверки каждого микросервиса:

```
curl localhost:8081/posts
curl localhost:8082/users
curl localhost:8082/connections
curl localhost:8080/connectionsposts/cdavisafc
```

В частности, следите за всеми тремя верхними окнами при выполнении последней команды `curl`. Вы увидите, что этот единственный запрос в конечном итоге затрагивает все три микросервиса.

С помощью этой версии программного обеспечения вы теперь можете реализовать топологию развертывания, как показано на рис. 5.3; различные приложения масштабируются независимо. Ваше текущее развертывание, однако, довольно безоблачно: все работает локально, у вас есть только отдельные экземпляры каждого приложения, а ваша конфигурация встроена в JAR-файлы с включением значений в файлы application.properties. Однако приложения не фиксируют текущее состояние. Вы можете остановиться и запустить любой экземпляр приложения без потери данных.

Не обращая особого внимания на это, я просто пробежалась по реализации шаблона, который позволяет сделать ваши приложения такими, чтобы они не фиксировали текущее состояние. Я переместила состояние во внешнее хранилище. Однако причина, по которой я до сих пор не рассказывала об этом подробно, заключается в том, что подключение к внешней базе данных является настолько знакомым большинству из вас, что мы можем упустить моменты, на которые я хочу обратить внимание, говоря об отсутствии фиксации состояния. Итак, чтобы изучить его, я снова хочу ввести состояние в один из микросервисов – тип состояния, которое может легко проникнуть в ваши проекты, если вы не сосредоточены на том, чтобы не пускать его.

> **ПРИМЕЧАНИЕ** Распространенный способ, с помощью которого состояние может проникнуть, – сохранение состояния сеанса.

5.2.2. Плохая обработка состояния сеанса

До сих пор клиент службы Connections' Posts мог просто указывать имя пользователя в строке запроса и получать посты, написанные людьми, на которых он был подписан. Вы можете запросить этот набор сообщений для любого пользователя в системе, безо всякого обременения. Однако вам нужно, чтобы Макс мог получать только посты пользователей, на которых он подписан, и то же самое и в случае с Гленом. Для этого вы попросите клиента службы Connections' Posts аутентифицироваться перед подачей любого контента.

Чтобы продолжить, пожалуйста, скачайте приведенный ниже тег для того же репозитория, с которым вы работаете:

```
Git checkout statelessness/0.0.2
```

Я добавила контроллер входа в систему для реализации службы Connections' Posts. При вызове функции входа в систему, которая для простоты принимает в качестве входных данных только имя пользователя, генерируется маркер, который затем передается последующим вызовам службы Connections' Posts. Если переданный маркер признан действительным, возвращается набор постов; если маркер недействителен, возвращается ответ `HTTP 1.1/401 Unauthorized`.

Контроллер входа в систему является частью службы Connections' Posts и находится в файле LoginController.java. Как видно из приведенного ниже кода, после того как маркер входа будет создан, он сохраняется в хешмапе в оперативной памяти, которая связывает маркер с именем пользователя.

Листинг 5.1 ❖ LoginController.java

```
package com.corneliadavis.cloudnative.connectionsposts;

import ...

@RestController
public class LoginController {

    @RequestMapping(value="/login", method = RequestMethod.POST)
    public void whoareyou(
        @RequestParam(value="username", required=false) String username,
        HttpServletResponse response) {

        if (username == null)
```

```
            response.setStatus(400);
        else {
            UUID uuid = UUID.randomUUID();
            String userToken = uuid.toString();

            CloudnativeApplication.validTokens.put(userToken, username);
            response.addCookie(new Cookie("userToken", userToken));
        }
    }
}
```

Этот хешмап объявляется в файле CloudnativeApplication.java, как показано ниже.

Листинг 5.2 ❖ CloudnativeApplication.java

```
public class CloudnativeApplication {

    public static Map<String, String> validTokens
        = new HashMap<String, String>();

    public static void main(String[] args) {
        SpringApplication.run(CloudnativeApplication.class, args);
    }
}
```

Я внесла еще одно важное изменение в метод, который обслуживает посты из пользовательских подключений, как показано ниже, во фрагменте кода, из файла ConnectionsPosts.java. Вместо того чтобы указывать имя пользователя как часть URL-адреса службы, служба не принимает имена пользователей, а ищет маркер в куки-файле, который передается службе.

Листинг 5.3 ❖ ConnectionsPostsController.java

```
    @RequestMapping(method = RequestMethod.GET, value="/connectionsposts")
    public Iterable<PostSummary> getByUsername(
        @CookieValue(value = "userToken", required=false) String token,
        HttpServletResponse response) {

        if (token == null)
            response.setStatus(401);
        else {
            String username =
                CloudnativeApplication.validTokens.get(token);
            if (username == null)
                response.setStatus(401);
            else {

                // Код для получения соединений и соответствующих постов;

                return postSummaries;
            }
        }
        return null;
    }
```

Можно переделать это приложение и заново развернуть службу Connections' Posts, чтобы протестировать данный функционал. В каталоге cloudnative-statelessness выполните следующую команду для создания проекта:

```
mvn clean install
```

Теперь, как и раньше, запустите микросервисы в трех окнах терминала с помощью этих команд:

```
java -Dserver.port=8081 \
-Dspring.datasource.url=jdbc:mysql://localhost:3306/cookbook \
-jar cloudnative-posts/target/cloudnative-posts-0.0.1-SNAPSHOT.jar

java -Dserver.port=8082 \
-Dspring.datasource.url=jdbc:mysql://localhost:3306/cookbook \
-jar cloudnative-connections/target/cloudnative-connections-0.0.1-SNAPSHOT.jar

java -jar cloudnative-connectionposts/target/cloudnative-connectionposts-0.0.1-SNAPSHOT.jar
```

Служба Connections' Posts больше не вызывается с именем пользователя как часть URL-адреса. Вызовы новой конечной точки перед выполнением любого входа в систему приведут к ошибке HTTP. Чтобы увидеть это, включите параметр `-i` в команду `curl`:

```
$ curl -i localhost:8080/connectionsposts
HTTP/1.1 401
X-Application-Context: application
Content-Length: 0
Date: Mon, 27 Nov 2018 03:42:07 GMT
```

Вы войдете в систему с помощью приведенной ниже команды; используйте одно из имен пользователей, предварительно загруженных в пример:

```
$ curl -X POST -i -c cookie localhost:8080/login?username=cdavisafc
HTTP/1.1 200
X-Application-Context: application
Set-Cookie: userToken=f8dfd8e2-9e8b-4a77-98e9-49aaed30c218
Content-Length: 0
Date: Mon, 27 Nov 2018 03:44:42 GMT
```

И теперь, когда вы вызываете службу Connections' Posts, передавая куки-файл с параметром командной строки `-b`, вы получите следующий ответ:

```
$ curl -b cookie localhost:8080/connectionsposts | jp -
[
  {
    “date”: “2019-02-01T19:09:41.000+0000”,
    “usersname”: “Max”,
    “title”: “Chicken Pho”
  },
  {
    “date”: “2019-02-01T19:09:41.000+0000”,
    “usersname”: “Glen”,
    “title”: “French Press Lattes”
  }
]
```

Я еще не говорила об этом явно, но вы могли заметить, что наша реализация больше не относится к типу stateless (без фиксации текущего состояния). Действительные маркеры хранятся в памяти. Чтобы вы не думали, что этот пример вымышленный, я говорю о хешмапе в моем основном приложении Spring Boot – уверяю вас, он довольно типичен для шаблона, обычно встречающегося сегодня в приложениях. Работая в компании Pivotal, недавно я вывела на рынок новый продукт для кеширования (Pivotal Cloud Cache) и провела множество бесед с разработчиками и архитекторами, которые искали эффективные способы обработки состояния сеанса HTTP, применяемые многими их приложениями. Хотя здесь я не использую явно ни один из интерфейсов HTTP, чтобы сделать код максимально простым, базовая структура остается той же.

Несмотря на это предполагаемое ограничение в реализации, если вы повторно несколько раз выполните последнюю команду `curl`, то будете постоянно получать ожидаемый ответ. Если все работает нормально, в чем тогда проблема? Проблема в том, что на этом этапе вы по-прежнему запускаете приложение в среде, которая не относится ни к типу «cloud», ни к типу «cloud-native». У вас есть только один экземпляр от каждого приложения, и при условии что они остаются работоспособными, а служба Connections' Posts продолжает работать, ваше программное обеспечение функционирует, как и ожидалось.

Но вы знаете, что в облаке все постоянно меняется, и из предыдущего раздела вы также знаете, что приложения почти всегда имеют несколько развернутых экземпляров, поэтому давайте проведем несколько экспериментов.

Во-первых, давайте смоделируем зацикливание службы Connections' Posts. В реальных условиях это может произойти в случае сбоя самого приложения, но еще более вероятно, что это произойдет из-за развертывания новой версии приложения или изменения инфраструктуры, в результате которого приложение будет создано заново в контексте обновленной инфраструктуры. Чтобы смоделировать это, остановите приложение, нажав сочетание клавиш **Ctrl-C** в правом окне, и повторно выполните команду `java`:

```
java -jar cloudnative-connectionposts/target/cloudnative-connectionposts-0.0.1-SNAPSHOT.jar
```

Теперь когда вы попробуете выполнить команду `curl` и передадите действительный маркер аутентификации, в ответ получите ошибку `HTTP 1.1/401 Unauthorized`:

```
$ curl -i -b cookie localhost:8080/connectionsposts
HTTP/1.1 401
X-Application-Context: application
Content-Length: 0
Date: Mon, 27 Nov 2018 04:12:07 GMT
```

Я уверена, что это вас не удивляет. Вы хорошо знаете, что маркеры хранятся в памяти, и когда вы остановили и перезапустили приложение, вы потеряли все, что было в памяти. Однако я надеюсь, что вы поняли главное: вы больше не можете рассчитывать на то, что такого рода зацикливания приложений не будут иметь место. Изменение – это правило, а не исключение.

Давайте теперь рассмотрим второй сценарий: развертывание нескольких экземпляров приложения. Как только у вас появятся эти несколько экземпляров, вам потребуется балансировщик нагрузки для маршрутизации трафика между

ними. Вы можете установить это самостоятельно, запустив что-то вроде nginx и сконфигурировав в нем все свои экземпляры, но это именно то, что делают за вас облачные платформы, поэтому я воспользуюсь этим как возможностью познакомить вас с одной из них. Как оказалось, у Kubernetes есть простая в использовании, локально развертываемая версия, что делает ее идеальным вариантом для того, что я хочу продемонстрировать. У Kubernetes есть активное сообщество, которое может предложить поддержку, когда это необходимо, и должна сказать вам, что Kubernetes – это потрясающая технология, с которой я люблю работать. Если вы знакомы и имеете доступ к другой облачной платформе, такой как Cloud Foundry, Heroku, Docker, OpenShift или др., во что бы то ни стало проведите эти эксперименты там.

Знакомство с платформой для облачной среды

Kubernetes – это платформа для запуска приложений. Она включает в себя возможности, которые позволяют развертывать, отслеживать и масштабировать приложения, обеспечивая мониторинг работоспособности и автоматическое восстановление, о чем я говорила в предыдущих главах, и которые позволят вам довольно мощно тестировать различные примеры и шаблоны. Например, когда экземпляры приложения по какой-либо причине теряются, Kubernetes запустит новые экземпляры, чтобы заменить их. Как я уже неоднократно упоминала (и буду продолжать это делать), платформа для облачной среды обеспечит поддержку и даже реализацию множества шаблонов, которые я рассматриваю в этой книге.

Чтобы запустить приложение в Kubernetes, оно должно быть контейнеризированным. У вас должен быть образ Docker (или нечто подобное), в котором содержится ваше приложение. Я не буду подробно описывать контейнеризацию, но расскажу о шагах, которые нужно выполнить, чтобы ваше приложение можно было объединить в образ Docker и сделать его доступным для Kubernetes.

Дистрибутив проекта для Kubernetes с открытым исходным кодом, который вы будете использовать здесь, носит название Minikube (**https://github.com/kubernetes/minikube**). В то время как в рабочем окружении Kubernetes всегда будет развертываться как мультиузловая распределенная система (вероятно, в зонах доступности, о чем мы говорили в предыдущих главах). Minikube предлагает одноузловое развертывание, которое позволяет быстро приступить к работе на собственной рабочей станции.

Инструкции по установке Minikube включены в файл README в репозитории GitHub и содержат информацию по запуску в Linux, Windows и macOS. Перед установкой Minikube также необходимо установить интерфейс командной строки Kubernetes, `kubectl` (**https://kubernetes.io/docs/tasks/tools/install-kubectl/**). После того как вы выполнили необходимые действия (например, установили VirtualBox на своем компьютере) и установили Minikube, можно приступить к развертыванию нашего приложения в Kubernetes.

Я предоставила вам манифесты развертывания для всех компонентов, из которых состоит наш пример. На этом этапе у вас есть четыре компонента, которые вы ранее запускали локально: база данных MySQL и каждый из трех микросервисов (Connections, Posts и Connections' Posts). Для развертывания программного обеспечения в топологии, приведенной на рис. 5.5, выполните приведенные ниже действия.

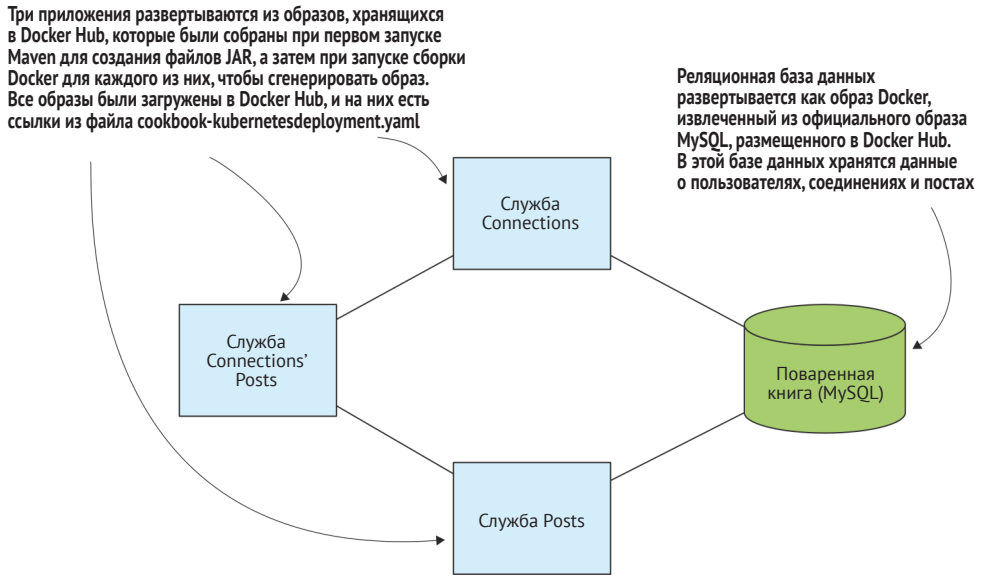


Рис. 5.5 ❖ Топология развертывания программного обеспечения поваренной книги, реорганизованная в отдельные компоненты и развернутая в облачной среде. На данный момент у каждого приложения есть только один развернутый экземпляр

Развертывание и настройка базы данных

Вы будете запускать в Kubernetes точно такой же образ Docker, что и при локальном запуске нашего примера. Используемый вами манифест развертывания – mysql-deploy.yaml. Вы поручаете Kubernetes запуск и управление этим компонентом с помощью данной команды:

```
kubectl create -f mysql-deployment.yaml
```

Можно наблюдать за состоянием развертывания следующим образом:

```
$ kubectl get all
NAME                           READY    STATUS    RESTARTS     AGE
pod/mysql-75d7b44cd6-dbnvp     1/1      Running   0            30s

NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
service/kubernetes   ClusterIP   10.96.0.1      none>         443/TCP          14d
service/mysql-svc    NodePort    10.97.144.19   <none>        3306:32591/TCP   6h14m

NAME                    READY UP-TO-DATE AVAILABLE AGE
deployment.apps/mysql   1/1   1          1         30s

NAME                               DESIRED CURRENT READY AGE
replicaset.apps/mysql-75d7b44cd6   1       1       1     30s
```

Вы заметите, что на выходе отображаются многочисленные сущности, которые ассоциированы с MySQL. Приведем краткий обзор: у вас есть *развертывание* MySQL, которое запускает приложение в *модуле* (образы Docker запускаются в модулях) и может быть доступно через *службу* MySQL. *Наборы реплик* указывают количество копий определенной рабочей нагрузки, которые должны быть запущены.

Для создания базы данных поваренной книги вы будете использовать тот же механизм, который вы использовали при локальном запуске. Вам нужно знать только строку подключения для передачи ее своему клиенту MySQL. Это можно получить из Minikube с помощью данной команды:

```
minikube service mysql –url
```

Если вы используете интерфейс командной строки `mysql` для доступа, то можно выполнить приведенную ниже команду для доступа к базе данных, а затем следующую команду для создания базы данных:

```
$ mysql -h $(minikube service mysql-svc --format “{{.IP}}”) \
  -P $(minikube service mysql-svc --format “{{.Port}}”) -u root -p
mysql> create database cookbook;
Query OK, 1 row affected (0.00 sec)
```

Ваш сервер базы данных запущен, и база данных, которую будет использовать ваше приложение, создана.

Настройка и развертывание служб Connections и Posts

Способы настройки и развертывания служб Connections и Posts практически идентичны. Каждая из них должна знать строку подключения и учетные данные для базы данных MySQL, которую вы только что развернули, и каждая из них будет работать в своем собственном контейнере (и модуле Kubernetes). Для каждой службы существует манифест развертывания, и вы должны отредактировать их для вставки строки подключения MySQL. Чтобы получить URL-адрес, который будет вставлен в каждый файл, выполните приведенную ниже команду:

```
minikube service mysql-svc --format “jdbc:mysql://{{.IP}}:{{.Port}}/cookbook”
```

Ответ при запуске этой команды выглядит так:

```
jdbc:mysql://192.168.99.100:32713/cookbook
```

Теперь отредактируйте файлы cookbook-deploy-connections.yaml и cookbook-deployposts.yaml, заменив строку `<insert jdbc url here>` на URL-адрес `jdbc`, возвращаемый из предыдущей команды `minikube`. Например, последние строки файла cookbook-deploy-kubernetes-connections.yaml будут выглядеть примерно так:

```
- name: SPRING_APPLICATION_JSON
value: ‘{“spring”:{“datasource”:{“url”:«jdbc:mysql://192.168.99.100:32713/cookbook»}}}’
```

Затем вы можете развернуть обе службы, выполнив две эти команды:

```
kubectl create -f cookbook-deployment-connections.yaml
kubectl create -f cookbook-deployment-posts.yaml
```

Запуск команды `kubectl get all` снова покажет, что теперь у вас есть два микросервиса, работающих в дополнение к базе данных MySQL:

```
$ kubectl get all
NAME                              READY STATUS  RESTARTS AGE
pod/connections-7dffdc87c4-p8fc8  1/1   Running 0        12s
pod/mysql-75d7b44cd6-dbnvp        1/1   Running 0        13m
pod/posts-6b7486dc6d-wmvmv        1/1   Running 0        12s
```

```
NAME                      TYPE     CLUSTER-IP     EXTERNAL-IP    PORT(S)
service/connections-svc   NodePort  10.106.214.25  <none>      80:30967/TCP
service/kubernetes        ClusterIP 10.96.0.1      <none>      443/TCP
service/mysql-svc         NodePort  10.97.144.19   <none>      3306:32591/TCP
service/posts-svc         NodePort  10.99.106.23   <none>      80:32145/TCP

NAME                          READY   UP-TO-DATE AVAILABLE AGE
deployment.apps/connections   1/1     1          1         12s
deployment.apps/mysql         1/1     1          1         13m
deployment.apps/posts         1/1     1          1         12s

NAME                                    DESIRED CURRENT READY AGE
replicaset.apps/connections-7dffdc87c4  1       1       1     12s
replicaset.apps/mysql-75d7b44cd6        1       1       1     13m
replicaset.apps/posts-6b7486dc6d        1       1       1     12s
```

Чтобы протестировать, что каждая служба работает правильно, используйте две приведенные ниже команды для получения примеров подключений и постов, которые были загружены в вашу базу данных:

```
$ curl $(minikube service --url connections-svc)/connections
[
  {
    “id”: 4,
    “follower”: 2,
    “followed”: 1
  },
  {
    “id”: 5,
    “follower”: 1,
    “followed”: 2
  },
  {
    “id”: 6,
    “follower”: 1,
    “followed”: 3
  }
]
$ curl $(minikube service --url posts-svc)/posts
[
  {
    “id”: 7,
    “date”: “2019-02-03T04:36:28.000+0000”,
    “userId”: 2,
    “title”: “Chicken Pho”,
    “body”: “This is my attempt to re-create what I ate in Vietnam...”
  },
  {
    “id”: 8,
    “date”: “2019-02-03T04:36:28.000+0000”,
    “userId”: 1,
    “title”: “Whole Orange Cake”,
    “body”: “That’s right, you blend up whole oranges, rind and all...”
  },
```

```
  {
    “id”: 9,
    “date”: “2019-02-03T04:36:28.000+0000”,
    “userId”: 1,
    “title”: “German Dumplings (Kloesse)”,
    “body”: “Russet potatoes, flour (gluten free!) and more...”
  },
  {
    “id”: 10,
    “date”: “2019-02-03T04:36:28.000+0000”,
    “userId”: 3,
    “title”: “French Press Lattes”,
    “body”: “We’ve figured out how to make these dairy free, but just as good!...»
  }
]
```

Настройка и развертывание службы Connections’ Posts

Наконец, давайте развернем службу, которая собирает и возвращает набор постов, сделанных людьми, на которых подписан конкретный человек. Эта служба не имеет прямого доступа к базе данных; вместо этого она выполняет сервисные вызовы для служб Connections и Posts. В развертываниях, которые вы только что выполнили в предыдущем разделе, эти службы теперь работают по URL-адресам, которые вы только что протестировали, и вам нужно лишь настроить эти адреса в службе Connections’ Posts. Это можно сделать, отредактировав манифест развертывания, cookbook-deployment-connectionsposts-stateful.yaml. Для трех URL-адресов есть заполнители, и они должны быть заполнены значениями, полученными с помощью приведенных ниже команд:

Posts URL	`minikube service posts-svc --format «http://{{.IP}}:{{.Port}}/posts?userIds=» --url`
Connections URL	`minikube service connections-svc --format «http://{{.IP}}:{{.Port}}/connections/» --url`
Users URL	`minikube service connections-svc --format «http://{{.IP}}:{{.Port}}/users/» --url`

Последние строки вашего манифеста развертывания будут выглядеть примерно так:

```
        - name: CONNECTIONPOSTSCONTROLLER_POSTSURL
          value: “http://192.168.99.100:31040/posts?userIds=”
        - name: CONNECTIONPOSTSCONTROLLER_CONNECTIONSURL
          value: “http://192.168.99.100:30494/connections/”
        - name: CONNECTIONPOSTSCONTROLLER_USERSURL
          value: http://192.168.99.100:30494/users/
```

Наконец, мы развертываем службу с помощью этой команды:

```
kubectl create \
-f cookbook-deployment-connectionsposts-stateful.yaml
```

Теперь вы можете протестировать эту службу, как и раньше, выполнив приведенные ниже команды:

```
curl -i $(minikube service --url connectionsposts-svc)/connectionsposts
curl -X POST -i -c cookie \
$(minikube service --url connectionsposts-svc)/login?username=cdavisafc
curl -i -b cookie \
$(minikube service --url connectionsposts-svc)/connectionsposts
```

Я обещаю, что скоро будет лучше. Я знаю, что вся эта конфигурация, осуществляемая вручную, удручает вас, но не беспокойтесь. Следующая глава посвящена настройке приложения, и, используя надлежащие практики, от некоторых из этих утомительных мер можно будет избавиться.

Чтобы подготовиться к следующей демонстрации, я попрошу вас выполнить потоковую передачу журналов для службы Connections' Posts. В новом окне терминала выполните приведенную ниже команду, предоставив имя `connectionsposts pod` (это можно увидеть при запуске команды `kubectl get pods`):

```
kubectl logs -f pod/<name of your connectionsposts pod>
```

Теперь вы можете повторить последнюю команду `curl`, показанную ранее, и увидеть результат действия в журналах `connections`. Вы готовы к работе. И при условии, что в случае со службой Connections' Posts не будет никаких циклов, это развертывание по-прежнему будет хорошо обслуживать ваших пользователей. Но что происходит, когда вы масштабируете службу для нескольких экземпляров? Для этого выполните команду:

```
kubectl scale --replicas=2 deploy/connectionsposts
```

При выполнении команды `get kubectl` мы снова видим ряд интересных моментов:

```
NAME                                READY STATUS  RESTARTS AGE
pod/connections-7dffdc87c4-cp7z7    1/1   Running 0        10m
pod/connectionsposts-5dc77f8bf9-8kgld 1/1   Running 0        5m4s
pod/connectionsposts-5dc77f8bf9-mvt89 1/1   Running 0        81s
pod/mysql-75d7b44cd6-dbnvp          1/1   Running 0        36m
pod/posts-6b7486dc6d-kg8cp          1/1   Running 0        10m

NAME                          TYPE      CLUSTER-IP     EXTERNAL-IP PORT(S)
service/connections-svc       NodePort  10.106.214.25  <none>      80:30967/TCP
service/connectionsposts-svc  NodePort  10.100.25.18   <none>      80:32237/TCP
service/kubernetes            ClusterIP 10.96.0.1      <none>      443/TCP
service/mysql-svc             NodePort  10.97.144.19   <none>      3306:32591/TCP
service/posts-svc             NodePort   10.99.106.23  <none>      80:32145/TCP

NAME                          READY UP-TO-DATE AVAILABLE AGE
deployment.apps/connections   1/1   1          1         10m
deployment.apps/connectionsposts 2/2   2          2         5m4s
deployment.apps/mysql         1/1   1          1         36m
deployment.apps/posts         1/1   1          1         10m

NAME                                     DESIRED CURRENT READY AGE
replicaset.apps/connections-7dffdc87c4   1       1       1     10m
replicaset.apps/connectionsposts-5dc77f8bf9 2       2       2     5m4s
replicaset.apps/mysql-75d7b44cd6         1       1       1     36m
replicaset.apps/posts-6b7486dc6d         1       1       1     10m
```

Вы видите, что был создан второй модуль, в котором запущен экземпляр службы Connections' Posts. Но там остается только одна служба `connectionposts-svc`. Теперь она будет выполнять балансировку нагрузки для запросов в обоих экземплярах приложения. Также видно, что и контроллер развертывания, и контроллер репликации выказывают желание запустить два экземпляра приложения в любой момент.

Не делайте этого в рабочем окружении!

Нельзя не отметить, что ввод команды для предоставления второго экземпляра приложения Connections' Posts *в командной строке ни при каких обстоятельствах не следует осуществлять в рабочем окружении.*

Как только вы это сделаете, вы создадите снежинку, программную топологию, которая нигде не записана. Манифест развертывания, который вы использовали для первоначального развертывания программного обеспечения, фиксирует первоначальную топологию и, скорее всего, то, что вы ожидаете, работает в рабочем окружении, но реальность отличается от того, что было записано. Когда люди вручную применяют изменения к работающей системе, эта система больше не воспроизводится из артефактов, которые записываются и контролируются как часть ваших эксплуатационных методов.

Правильная практика для достижения такого масштабирования – изменить манифест развертывания, загрузить его в систему управления и применить к облачной среде. Kubernetes поддерживает это, обновляя работающую систему, вместо того чтобы создавать новую с помощью таких команд, как `kubectl apply`. В данном случае я иду коротким путем, только чтобы вам было проще.

Перед тестированием функциональности своего приложения в новой топологии развертывания выполните потоковую передачу журналов для нового экземпляра приложения. В другом окне терминала выполните команду `kubectl logs -f` с именем нового модуля, как видно из вывода команды `kubectl apply`:

```
kubectl logs -f po/<name of your new pod>
```

Теперь давайте еще раз протестируем функциональность приложения, выполнив последнюю команду `curl` еще несколько раз:

```
curl -i -b cookie \
$(minikube service --url connectionsposts-svc)/connectionsposts
```

Следите за обоими окнами, в которых вы выполняете потоковую передачу журналов приложений. Если балансировщик нагрузки направляет трафик в первый экземпляр, вы увидите активность в этом потоке, и результаты будут возвращены, как и ожидалось. Однако если трафик направляется во второй экземпляр, этот журнал покажет, что была попытка получить доступ к приложению с недействительным маркером, и вы получите ошибку 401. Но конечно же, маркер действителен; проблема в том, что второй экземпляр не знает о нем.

Я уверена, мы все согласны в том, что это ужасный пользовательский опыт. Иногда команда `curl` возвращает запрошенную информацию, а иногда сообщает, что пользователь не был аутентифицирован. Когда после этого пользователь по-

чесывает голову и думает: «Разве я уже не выполнил вход?» – и обновляет страницу, он может получить действительный ответ, но затем после следующего запроса снова появится сообщение о том, что он не аутентифицирован.

Использование Kubernetes требует от вас создания образов Docker

Вы, наверное, заметили, что я не просила вас создать приложение, как в предыдущих примерах. Здесь я пропустила шаги, потому что, для того чтобы развернуть приложения в Kubernetes, они должны быть контейнеризованы, а процесс сборки несколько сложен. Вам нужно создать JAR-файлы, выполнить команду `docker build` для создания образов Docker, загрузить эти образы в репозиторий образов, обновить манифест развертывания, чтобы он указывал на эти образы, а затем выполнить развертывание.

Я включила сюда файл Dockerfile, который использовала для создания каждого образа контейнера. Но для краткости манифесты развертывания, которые я предоставляю, указывают на образы Docker, которые я уже создала и загрузила в собственный репозиторий Docker Hub. Вот команды, которые я выполнила, чтобы проделать все это:

Источник сборки:

```
mvn clean install
```

Создание образов Docker:

```
docker build --build-arg \
jar_file=cloudnative-connectionposts/target\
/cloudnative-connectionsposts-0.0.1-SNAPSHOT.jar \
-t cdavisafc/cloudnative-statelessness-connectionsposts-stateful .

docker build --build-arg \
jar_file=cloudnative-connections/target\
/cloudnative-connections-0.0.1-SNAPSHOT.jar \
-t cdavisafc/cloudnative-statelessness-connections .

docker build --build-arg \
jar_file=cloudnative-posts/target/cloudnative-posts-0.0.1-SNAPSHOT.jar \
-t cdavisafc/cloudnative-statelessness-posts .
```

Загрузка в Docker Hub:

```
docker push cdavisafc/cloudnative-statelessness-connectionposts-stateful
docker push cdavisafc/cloudnative-statelessness-connections
docker push cdavisafc/cloudnative-statelessness-posts
```

Хотя в этом простом примере легко понять, почему это происходит, давайте посмотрим, что происходит с точки зрения архитектуры. На рис. 5.6 показано желаемое поведение нашего приложения. Я показываю одну сущность для приложения Connections' Posts, чтобы *логически* обозначить, что у вас есть только одно приложение. Как я говорила в начале главы, один и тот же ввод должен давать одинаковый результат независимо от количества экземпляров приложения и от того, к какому приложению направляется конкретный запрос.

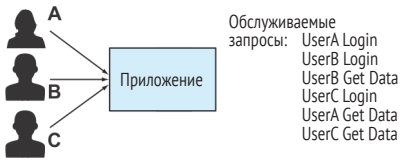


Рис. 5.6 ❖ Логически у вас есть одно приложение, которое обслуживает серию запросов. Поскольку каждый пользователь выполняет вход перед попыткой выполнения каких-либо вызовов для получения данных, вы ожидаете, что каждый из них будет успешным

Но в данной текущей реализации у вас это не получилось, потому что ваше приложение фиксирует текущее состояние. Вот простой способ визуализировать то, что происходит. Посмотрите на рис. 5.7. На нем видно, что ваше логическое приложение делится на два экземпляра и последовательность запросов распределяется по этим экземплярам. На данной диаграмме я не стала изменять последовательность запросов, а лишь распределила их по разным экземплярам. Теперь легко можно увидеть, что если ваше приложение учитывает только локальные запросы, во многих случаях его функциональность будет нарушена.

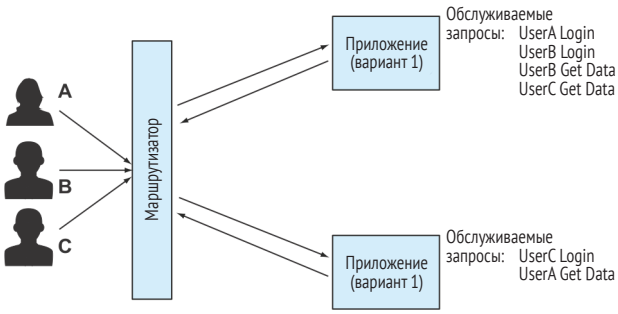


Рис. 5.7 ❖ Когда одна логическая сущность развертывается как несколько экземпляров, необходимо принять меры к тому, чтобы распределенный набор событий продолжал рассматриваться как единое целое

Возвращаясь к нашему примеру с поваренной книгой, рис. 5.8 резюмирует поведение, которое представляет ваша текущая реализация.

Как тогда решить эту проблему? Давайте рассмотрим одно «решение», которое сегодня широко распространено, но не подходит для облачной среды, – «липкие» сессии (потом я покажу решение для облачной среды).

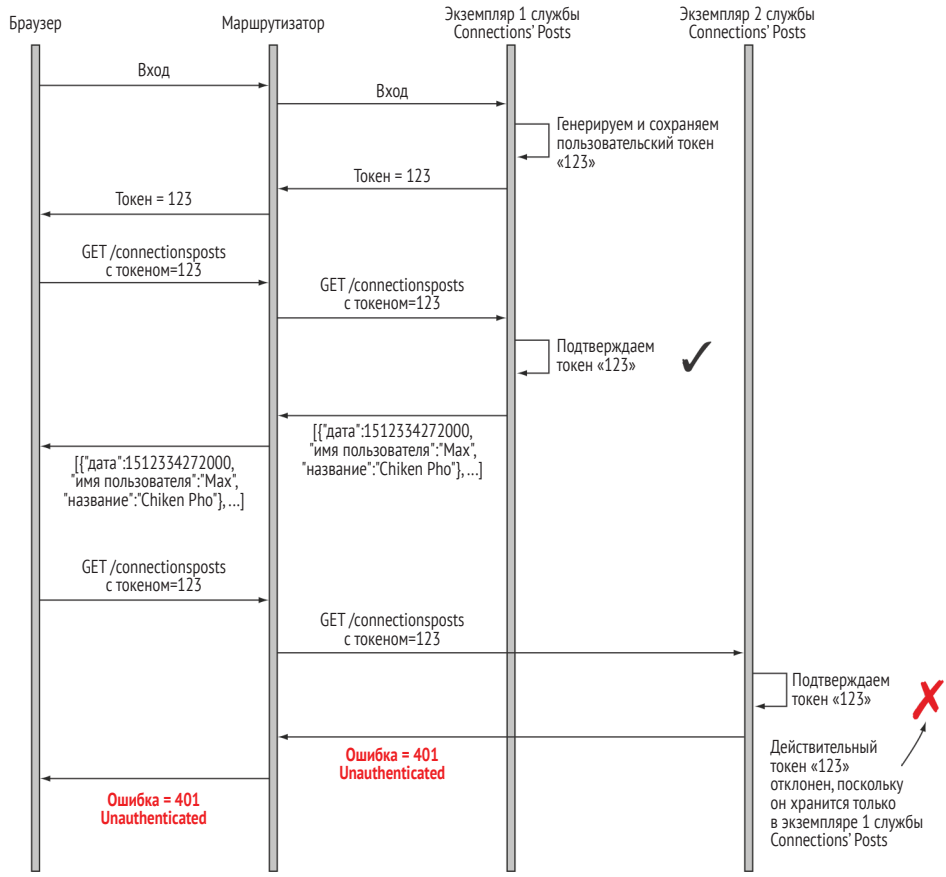


Рис. 5.8 ❖ Когда вы добавляете второй экземпляр приложения Connections 'Posts, его локальный список допустимых токенов отличается от первого. Это как раз та самая проблема, связанная с приложениями с сохранением состояния в программном обеспечении для облачной среды

5.3. HTTP-сессии и «липкие» сессии

Как насчет «липких» сессий? Почти на протяжении всего времени, как мы использовали балансировщики нагрузки для распределения запросов по нескольким экземплярам приложений – а это намного дольше, чем проектирование этих приложения облачным способом, – мы применяли «липкие» сессии. Почему и дальше нельзя использовать их для работы с сервисами, фиксирующими текущее состояние?

Во-первых, позвольте мне кратко объяснить эту методику. *Липкие сессии* – это шаблон реализации, в котором приложение включает идентификатор сессии в ответ на первый запрос пользователя – своего рода отпечаток пальца этого пользователя, если хотите. Этот идентификатор затем включается, обычно с помощью куки-файла, во все последующие запросы, что эффективно позволяет балансиров-

щику нагрузки отслеживать отдельных пользователей, которые взаимодействуют с приложением. Этот балансировщик нагрузки, который отвечает за принятие решения по поводу того, куда отправить конкретный запрос, запомнит, к какому экземпляру был получен доступ первым, а затем приложит все усилия, чтобы направлять все запросы, содержащие этот идентификатор сессии, к одному и тому же экземпляру приложения. Если этот экземпляр приложения имеет локальное состояние, у запросов, последовательно направляемых к этому экземпляру, локальное состояние будет доступным.

На рис. 5.9 показана эта последовательность: когда в запросе присутствует идентификатор сессии, маршрутизатор ищет экземпляр, которому соответствует этот идентификатор, и отправляет запрос этому экземпляру.

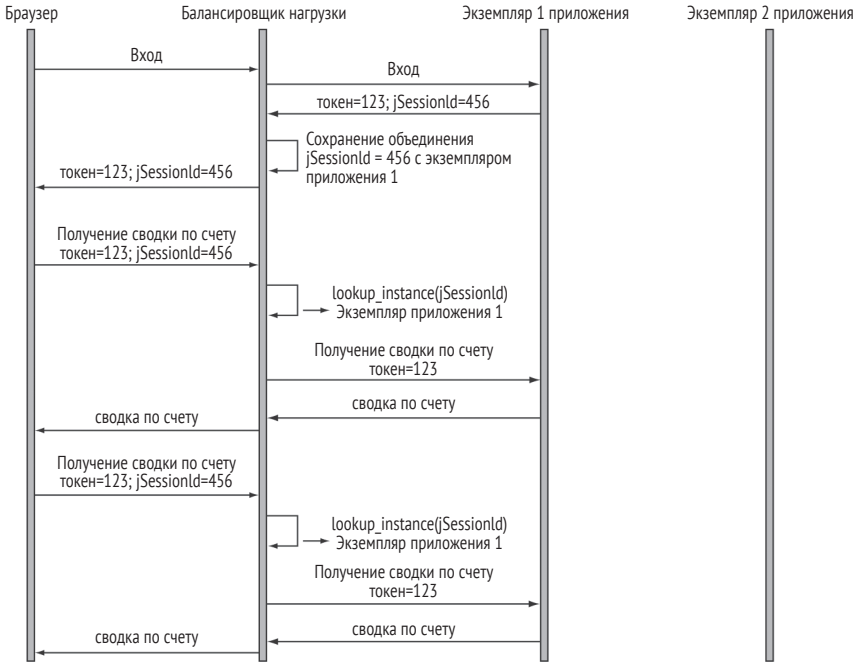


Рис. 5.9 ❖ «Липкие» сессии реализуются балансировщиком нагрузки и используются для привязки конкретного пользователя к конкретному экземпляру приложения

Разве это не проще, вместо того чтобы гарантировать, что каждое ваше приложение абсолютно не фиксирует текущее состояние?

Вы обратили внимание на ту часть, где была фраза «приложит все усилия»? Несмотря на все усилия, маршрутизатор может быть не в состоянии отправить запрос «правильному» экземпляру. Возможно, что этот экземпляр исчез или недоступен из-за сетевых аномалий. Например, на рис. 5.10 экземпляр приложения 1 недоступен, поэтому маршрутизатор отправит запрос пользователя другому экземпляру приложения. Поскольку у этого экземпляра нет локального состояния, которое было у экземпляра 1, пользователь снова будет испытывать негативное поведение, которое я демонстрировала ранее.

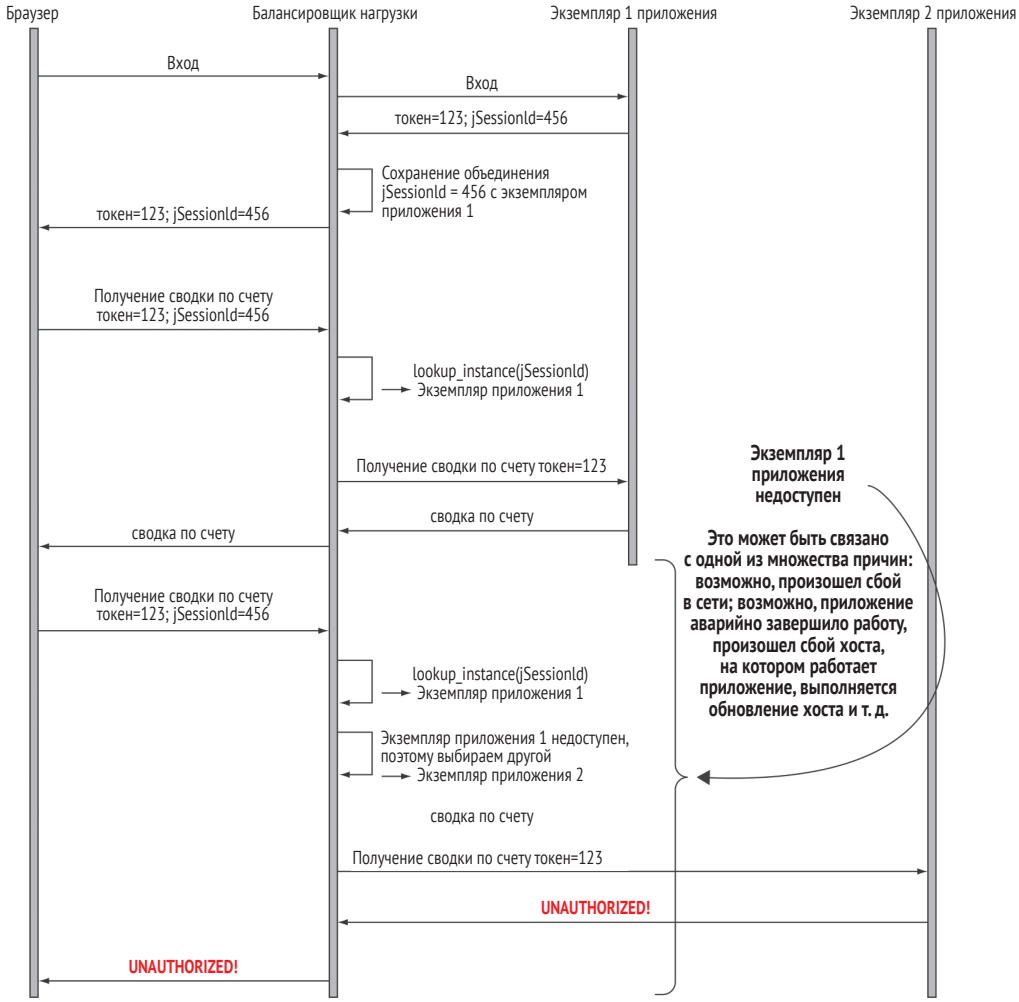


Рис. 5.10 ❖ Балансировщик нагрузки, поддерживающий «липкие» сессии, попытается привязать определенных пользователей к конкретным экземплярам приложения, но может быть вынужден отправить запрос другому экземпляру. В программном обеспечении для облачной среды вы должны допускать, что экземпляры приходят и уходят с некоторой регулярностью. Если вы этого не сделаете, опыт взаимодействия пострадает

Разработчики в течение некоторого времени оправдывали использование «липких» сессий, утверждая, что такие аномалии, как исчезающие экземпляры или перебои в работе сети, являются редкими, и что когда они действительно происходят, плохой пользовательский опыт хотя и нежелателен, но является приемлемым. Это плохой аргумент по двум причинам. Во-первых, ресайклинг экземпляров приложений становится все более распространенным явлением из-за непредвиденных или преднамеренных изменений инфраструктуры. Во-вторых, нетрудно реализовать что-нибудь получше, а именно сохранение состояния сеан-

са в подключенном резервном хранилище, и этот подход дает много других преимуществ. Давайте посмотрим.

5.4. Службы с фиксацией текущего состояния и приложения без фиксации состояния

Правильный подход к решению этой проблемы предложен в заголовке данной главы: сделать приложения такими, чтобы они не фиксировали текущее состояние. Чтобы вначале продемонстрировать этот антишаблон – службы с фиксацией состояния, а теперь показать правильный шаблон, я выбрала конкретный пример с аутентификацией пользователя, потому что это область, в которой я так часто встречалась с реализацией некачественных решений. Чтобы оправдать использование «липких» сессий, часто приводится аргумент, что у нас не может быть пользователей, предоставляющих свои учетные данные при каждом запросе. Хотя это и надежный аргумент, это не означает, что экземпляры приложения должны содержать это состояние.

5.4.1. Службы с фиксацией состояния – это специальные службы

Конечно, состояние где-то должно быть. В целом приложение должно иметь какое-то состояние, чтобы быть полезным. Например, ваш банковский сайт не принесет много пользы, если вы будете не в состоянии увидеть остаток на счете. Поэтому, когда я предлагаю, чтобы приложения не фиксировали состояние, я в действительности говорю о том, что нам не нужно, чтобы состояние было везде в нашей архитектуре.

> **ПРИМЕЧАНИЕ** В приложениях для облачной среды есть места, где находится состояние, и что не менее важно, есть места, где его нет.

Приложение не фиксирует состояние. Состояние находится в данных службах. В наши дни мы часто это слышим. Признаюсь, я не из тех, кто следует каким-либо указаниям, если я не понимаю, почему этот совет является обоснованным, поэтому давайте выполним проверку, кратко изучив, что нам нужно сделать, чтобы наше программное обеспечение работало хорошо, если бы у нас было много состояния в наших приложениях. Помните: одна из главных вещей, которую мы должны предвидеть, – это постоянные изменения.

В этом случае в любое время, когда внутреннее состояние приложения изменяется, и это состояние находится либо в памяти, либо на локальном диске, его нужно сохранить на случай, если экземпляр приложения потеряется. Существует несколько подходов для сохранения состояния, но все они включают репликацию. Один из вариантов, который совершенно не зависит от знания логики приложения, – это делать снапшоты; просто делайте копии памяти и диска с определенным интервалом.

Но как только мы начинаем говорить о снапшотах, у нас появляется множество решений относительно того, как создавать и управлять ими. Как часто вы их делаете? Как гарантировать их согласованность (то есть захватить состояние, которое не является непреднамеренно случайным, поскольку изменения произошли в процессе создания снапшота)? Сколько времени займет их восстановление?

Компромисс между максимальным периодом времени, за который могут быть потеряны данные в результате инцидента (RTO), и промежутками времени, в течение которых система может оставаться недоступной в случае аварии (RPO), по своей сути сложен. Даже если вы незнакомы с деталями получения снапшотов, RTO и RPO (я потратила более десяти лет, работая на крупного поставщика систем хранения данных ☺), одного этого описания, вероятно, было достаточно, чтобы уровень стресса немного повысился.

Еще один подход к созданию копий данных для обеспечения устойчивости – сделать этап репликации частью хранилища данных. Здесь события приложения запускают хранилище первичной копии данных, равно как одну или несколько реплик. Чтобы гарантировать доступность реплик, когда первичный сервер по какой-либо причине исчезает, копии распределяются по границам отказов: копии хранятся на разных хостах, в разных зонах доступности или на разных устройствах хранения. Но затем, как только эти процессы пересекают такие границы, в вашем распоряжении появляется распределенная система, работающая с данными, и, честно говоря, такие проблемы сложно решить.

Вы наверняка слышали о теореме CAP, которая гласит, что только два из трех свойств: *согласованность данных*, *доступность* и *устойчивость к разделению* – могут поддерживаться в любой системе. Поскольку распределенная система всегда страдает от случайного разбиения сетевого трафика, распределенные системы данных могут быть *либо* согласованными, *либо* доступными. Детальное изучение теоремы CAP и проблем распределенных сервисов с фиксацией состояния выходит за рамки обсуждаемого вопроса, но приведенное ниже замечание подтверждает мою точку зрения.

> **ПРИМЕЧАНИЕ** Обработка состояния в системах на базе облака, которые по определению являются сильно распределенными, требует особого внимания и сложных алгоритмов. Вместо того чтобы решать эти проблемы в каждом приложении, из которого состоит наше программное обеспечение, мы сконцентрируем решения только на отдельных частях нашей архитектуры для облачной среды. Эти части – *службы с фиксацией состояния*.

Таким образом, вы помещаете состояние в специально предназначенные для этого службы с фиксацией состояния и удаляете это состояние из своих приложений. Вскоре мы проделаем это в контексте нашего приложения для поваренной книги. Но сначала я хочу указать на ряд других преимуществ приложений без фиксации состояния, помимо того что они избегают сложности устойчивости распределенных данных. Когда ваши приложения не фиксируют состояние, платформа приложений для облачной среды может легко создавать новые экземпляры приложения, когда старые экземпляры потерялись. Нужно только запустить новый экземпляр из того же базового состояния, в котором он запускал оригинал, и в добрый путь. Уровень маршрутизации может равномерно распределять нагрузку по нескольким экземплярам, количество которых можно регулировать в зависимости от объема запросов, которые необходимо обработать; никаких операций, за исключением регистрации координат новых экземпляров, не нужно. Экземпляры можно с легкостью перемещать. Нужно обновить хост, на котором работают ваши экземпляры? Никаких проблем, просто запустите новый экземпляр и направьте трафик.

Вы можете разумно управлять несколькими *версиями* приложения, развернутыми и работающими бок о бок (вспомните важность параллельного развертывания в наших предыдущих беседах о непрерывной доставке). Можно направить часть своего трафика к самой последней версии, а другой трафик достигнет предыдущих версий, и, несмотря на изменчивость компонентов приложения, состояние по-прежнему будет последовательно обрабатываться в службах с фиксацией состояния.

Теперь, чтобы было ясно, нет абсолютно ничего плохого в том, что приложение хранит данные в памяти и даже на локальном диске. Но на эти данные можно рассчитывать только на время одного вызова приложения, которое сгенерировало их в первую очередь. Конкретный пример – приложение, которое будет загружать образ, обрабатывать его каким-то образом и возвращать его новый рендеринг. Обработка может быть многоэтапной и сохранять промежуточные результаты на диске, но это локальное хранилище эфемерно, оно существует только до тех пор, пока окончательный образ не будет сгенерирован и возвращен вызывающей стороне. Рискуя зря потерять время, нельзя рассчитывать на то, что эти данные будут доступны, когда в тот же экземпляр приложения поступит следующий запрос.

Задача разработчиков заключается в том, чтобы четко указывать, какое состояние важно сохранить, а какое нет, и проектировать свои приложения таким образом, чтобы любые данные, необходимые при вызовах, помещались в службы с фиксацией состояния. После тщательного рассмотрения этого элемента проектирования у вас будет лучшее из обоих миров. Часть вашей общей реализации может управляться с учетом масштабов и текучести системы, на которой она работает (приложение без фиксации состояния), а другая часть может быть спроектирована для выполнения более сложной задачи по управлению данными (состояние).

5.4.2. Создание приложений без сохранения состояния

Давайте теперь вернемся к нашему примеру с поваренной книгой. Когда мы закончили, мы реализовали простую аутентификацию пользователя. Но поскольку вы сохранили действительные маркеры в памяти, если запрос был перенаправлен на экземпляр, который не хранит маркер пользователя, пользователя попросят выполнить вход, даже если он уже это сделал.

Есть простое решение: вы будете использовать хранилище типа «ключ/значение», которое будет использоваться для сохранения действительных маркеров входа в систему, и привяжете свое приложение к службе с фиксацией состояния. Каждый экземпляр приложения будет включать в себя эту привязку (то, как записывается эта привязка, предзнаменует следующую главу о конфигурации приложения), поэтому любой запрос можно направить на любой экземпляр приложения, и действительные маркеры будут доступными. Рисунок 5.11 показывает эту топологию, обновляя рис. 5.8, который содержал предыдущий поток.

Я реализовала данное решение в каталоге/модуле cloudnative-statelessness репозитория для облачной среды. Топология развертывания, которую вы получите с помощью приведенных ниже инструкций, показана на рис. 5.12. Решение находится в основной ветви вашего репозитория, поэтому, если вы пред-

варительно клонировали и скачали более ранний тег, можете переключиться на главную ветку:

```
git checkout master
```

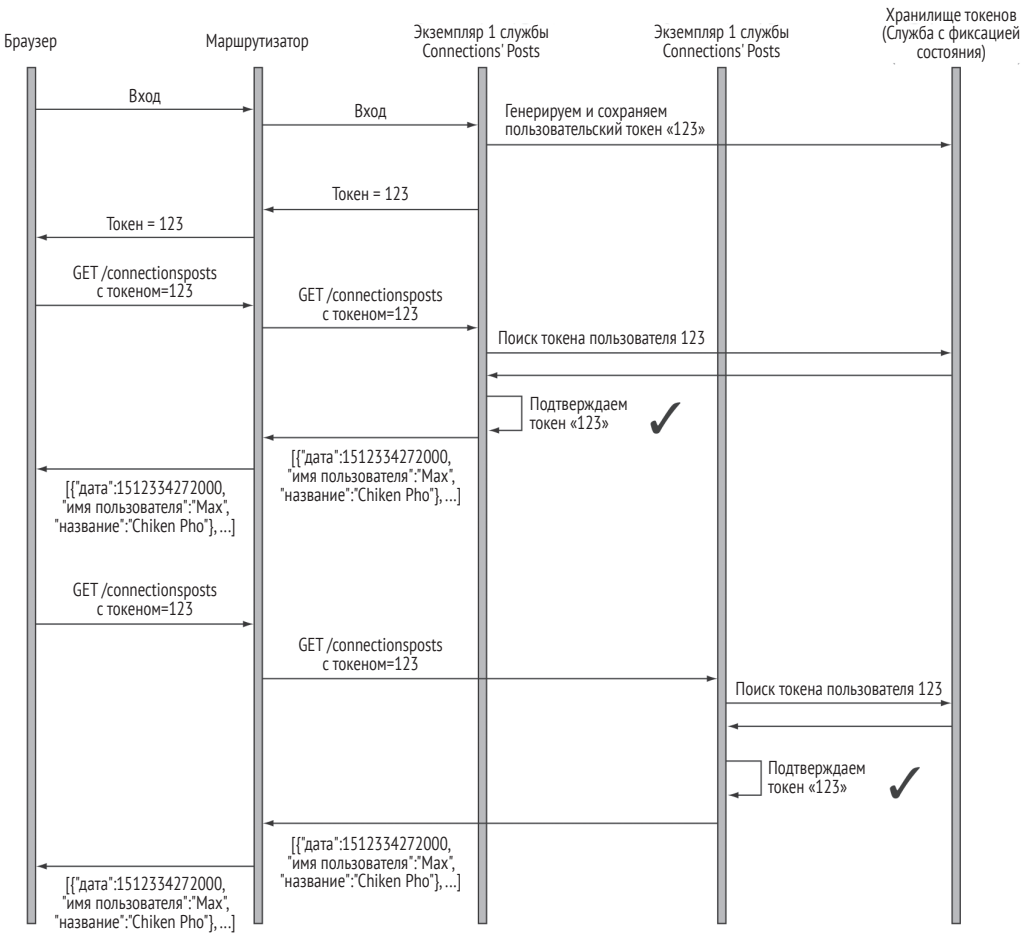


Рис. 5.11 ❖ Хранение действительных токенов в службе с сохранением состояния, которая привязана ко всем экземплярам службы Connections' Posts, позволяет приложениям быть stateless (без сохранения состояния), но наше ПО сохраняет состояние

В данной реализации используется Redis в качестве службы с сохранением состояния. Запустите эту службу через другое развертывание в своей среде Minikube. Выполните эту команду, чтобы запустить сервер Redis:

```
kubectl create -f redis-deployment.yaml
```

После запуска контейнера можно подключиться к нему, используя интерфейс командной строки Redis (или другого клиента, если хотите) с помощью приведенной ниже команды, а также просмотреть ключи, хранящиеся в Redis:

```
redis-cli -h $(minikube service redis-svc --format “{{.IP}}”) \
-p $(minikube service redis-svc --format “{{.Port}}”)
> keys *
```

Теперь замените предыдущую службу Connections' Posts на новую. Изменения кода в приложении минимальны.

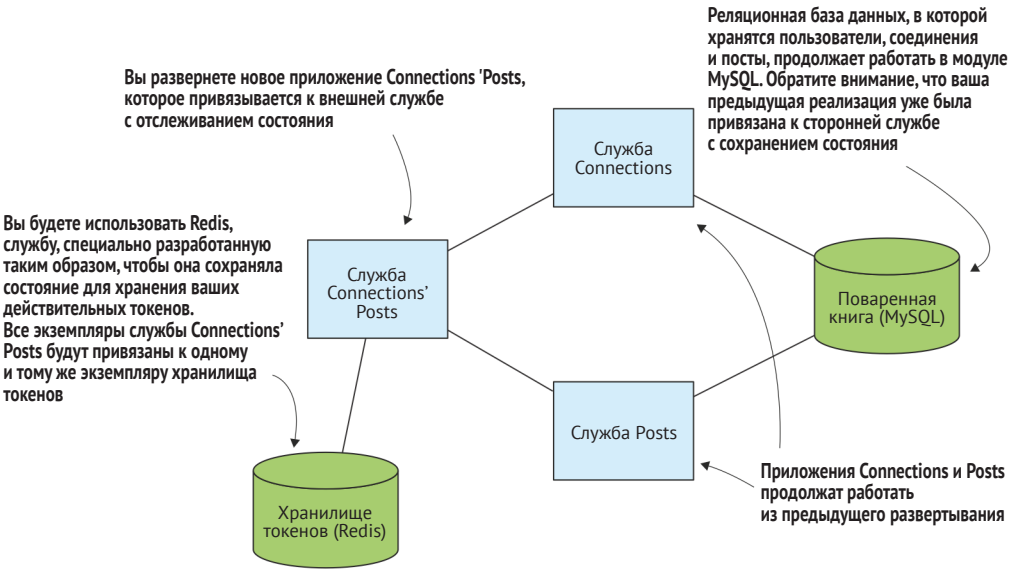


Рис. 5.12 ❖ Ваше новое развертывание заменяет службу Connections' Posts версией без сохранения состояния и добавляет сервер Redis для хранения токенов аутентификации

Сначала в конфигурации и главном классе Spring Boot `CloundnativeApplication` удалите локальное хранилище маркеров и добавьте конфигурацию для клиента Redis, который будет использоваться для подключения к вашему хранилищу маркеров.

Листинг 5.4 ❖ CloudnativeApplication.java

```
public class CloudnativeApplication {

@Value(“${redis.hostname}”)
    private String redisHostName;
    @Value(“${redis.port}”)
    private int redisPort;

    @Bean
    public RedisConnectionFactory redisConnectionFactory() {

        return new LettuceConnectionFactory
          (new RedisStandaloneConfiguration(redisHostName, redisPort));
    }
```

```
    public static void main(String[] args) {
        SpringApplication.run(CloudnativeApplication.class, args);
    }
}
```

Далее вместо хранения маркеров в удаленном локальном хранилище, в коде файла LoginController.java вы используете клиента Redis, чтобы сохранить маркер для имени пользователя в вашем внешнем хранилище с фиксацией состояния.

Листинг 5.5 ❖ LoginController.java

```
...
~~CloudnativeApplication.validTokens.put(userToken, username);~~  ← Мы убрали это

ValueOperations<String, String> ops = this.template.opsForValue();
ops.set(userToken, username);
```

Затем, вместо того чтобы извлекать маркеры из удаленного локального хранилища, в ConnectionsPostsController вы используете клиента Redis для получения имени пользователя через маркер, предоставленный посредством куки-файла.

Листинг 5.6 ❖ ConnectionsPostsController.java

```
...
~~String username
    = CloudnativeApplication.validTokens.get(token);~~  ← Мы убрали это

ValueOperations<String, String> ops = this.template.opsForValue();
String username = ops.get(token);
```

Чтобы прояснить ваши действия здесь, я заставлю вас удалить старую версию приложения Connections' Posts с помощью этой команды:

```
kubectl delete deploy connectionsposts
```

Перед развертыванием новой версии приложения Connections' Posts необходимо настроить информацию о соединении Redis в манифесте развертывания. Отредактируйте файл cookbookdeployment-connectionsposts-stateless.yaml, вставив имя хоста Redis и порт в соответствующие места. Вы можете получить значения имени хоста и порта с помощью этих двух команд:

```
minikube service redis --format “{{.IP}}”
minikube service redis --format “{{.Port}}”
```

После завершения этот YAML-файл будет выглядеть примерно так:

```
        - name: CONNECTIONPOSTSCONTROLLER_POSTSURL
          value: “http://192.168.99.100:31040/posts?userIds=”
        - name: CONNECTIONPOSTSCONTROLLER_CONNECTIONSURL
          value: “http://192.168.99.100:30494/connections/”
        - name: CONNECTIONPOSTSCONTROLLER_USERSURL
          value: “http://192.168.99.100:30494/users/”
        - name: REDIS_HOSTNAME
          value: “192.168.99.100”
        - name: REDIS_PORT
          value: «32410»
```

Теперь вы можете развернуть новое приложение с помощью этой команды:

```
kubectl create \
-f cookbook-deployment-connectionsposts-stateless.yaml
```

И протестировать свое программное обеспечение с помощью обычной серии команд:

```
curl -i $(minikube service --url connectionsposts-svc)/connectionsposts
curl -X POST -i -c cookie \
$(minikube service --url connectionsposts-svc)/login?username=cdavisafc
curl -i -b cookie \
$(minikube service --url connectionsposts-svc)/connectionsposts
```

Используя метод POST в команде `curl`, мы создадим новый ключ в хранилище Redis, что можно увидеть, выполнив команду `keys *`, запущенную через интерфейс командной строки Redis. А теперь давайте вернем топологию к тому, что было у вас, когда стали свидетелями проблем с приложением с фиксацией состояния. Масштабируйте приложение Connections' Posts на два экземпляра:

```
kubectl scale --replicas=2 deploy/connectionsposts
```

Теперь вы можете выполнять потоковую передачу журналов для обоих экземпляров (используя команду `kubectl logs -f <podname>`, как вы это делали прежде) и повторить последнюю команду `curl`, чтобы убедиться, что оба экземпляра теперь могут видеть все действительные маркеры и правильно обслуживать ответы:

```
curl -i -b cookie \
 $(minikube service --url connectionsposts-svc)/connectionsposts
```

Да, это действительно так просто. Конечно, вы должны быть осторожны с тем, чтобы сделать свои приложения не сохраняющими состояние, возможно, нарушая старые привычки, но это простой шаблон, и в результате вы получите огромные преимущества. Некоторые из вас будут работать над более сложными проблемами, связанными с управлением распределенными службами с фиксацией состояния, но большинство из нас (включая и меня) может легко сделать наши приложения не сохраняющими состояние и просто воспользоваться тяжелой работой и инновациями в сфере сервисов с сохранением состояния для облачной среды.

Однако топология нашего приложения теперь более сложна, и при большем распределении возникает ряд проблем. Что происходит, когда вам нужно переместить службу с отслеживанием состояния в новые координаты? Например, она получает новый URL-адрес, или вам нужно обновить учетные данные, которые вы используете для подключения к ней (то, что мы еще даже не затрагивали). Что происходит, когда доступ к этой службе прерывается даже на мгновение? Мы будем решать эти и другие проблемы по мере продвижения. Далее вы узнаете, как управлять конфигурацией приложения таким образом, чтобы можно было легко адаптироваться к изменяющейся природе вашего облака и требованиям к приложению.

Резюме

- Приложения с фиксацией состояния плохо работают в контексте для облачной среды.
- Последовательность взаимодействий с пользователем, логически воспринимаемая как захваченная в состоянии сессии, является распространенным способом проникновения состояния в ваши приложения.
- Службы с фиксацией состояния – особый тип служб, которые должны решать серьезные проблемы устойчивости данных в распределенной среде на базе облака.
- Большинство приложений не должно фиксировать состояние и перекладывать обработку состояния на службы с сохранением состояния.
- Создание приложений без сохранения состояния – процесс не сложный, и при его осуществлении у вас появляются существенные преимущества в облачной среде.

Глава 6

Конфигурация приложения: не только переменные среды

О чем идет речь в этой главе:

- требования к конфигурации приложения;
- разница между значениями конфигурации системы и приложения;
- правильное использование файлов свойств;
- правильное использование переменных среды;
- серверы конфигурации.

В начале предыдущей главы я показала иллюстрацию, которая повторяется здесь на рис. 6.1. Эта диаграмма показывает, что – хотя кажется простым, что одни и те же входные данные для любого из нескольких экземпляров приложения дадут одинаковые результаты, – на эти результаты влияют и другие факторы, а именно история запросов, системная среда и любая конфигурация приложения. В главе 5 вы изучили методы устранения воздействия первого из них, гарантируя, что любое состояние, вытекающее из последовательности запросов, будет храниться в общей сторонней службе, и это позволило экземплярам приложения не фиксировать состояние.

В этой главе рассматриваются два оставшихся фактора: системная среда и конфигурация приложения. Ни один из них не является абсолютно новым, когда речь идет о программном обеспечении для облачной среды. На функциональность приложения всегда влиял контекст, в котором оно работает, и применяемая конфигурация. Но новые архитектуры, о которых я говорю в этой книге, приносят с собой новые вызовы. Я начинаю главу со знакомства с некоторыми из них. Далее я расскажу о том, что называю *уровнем конфигурации* приложения, – механизме, позволяющем и системной среде, и конфигурации приложения проникать в приложение. Затем я подробно расскажу об использовании значений системной среды; вы, вероятно, слышали фразу «сохранить конфигурацию в переменных среды», и я объясняю, что это. И наконец, я сконцентрируюсь на конфигурации приложения с оглядкой на «антиснежинки», о которых я говорила в предыдущих главах, а также объясню содержимое рис. 6.1.

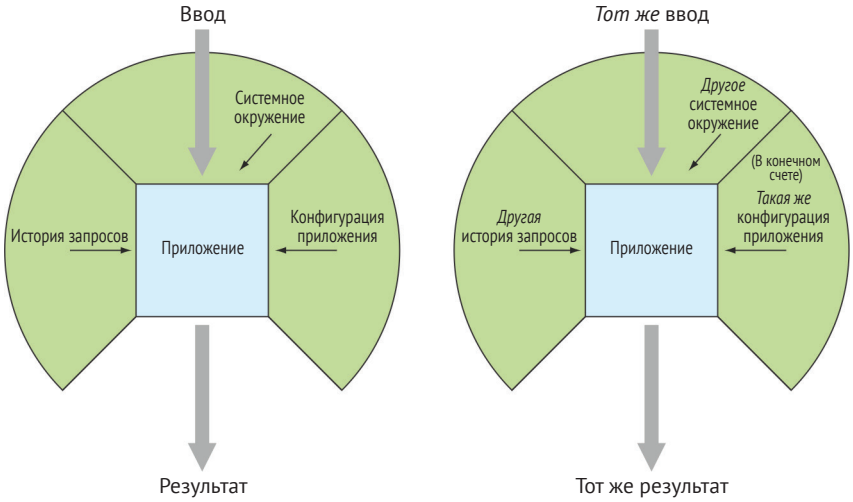


Рис. 6.1 ❖ Приложение для облачной среды должно обеспечивать согласованные результаты, несмотря на различия в контекстных факторах, таких как история запросов (разные экземпляры приложения обрабатывали разный набор запросов) и значения окружения системы (например, IP-адреса). Конфигурация приложения должна быть одинаковой для всех экземпляров, но иногда может отличаться

6.1. Почему мы вообще говорим о конфигурации?

Почему я вообще говорю о конфигурации приложения? Разработчики знают, что лучше не стоит жестко программировать конфигурацию в своем программном обеспечении (верно?). Файлы свойств существуют практически в каждой платформе программирования, и у нас были передовые методики на протяжении очень долгого периода времени. В конфигурации приложения нет ничего нового.

Но контекст для облачной среды является новым – достаточно новым, что даже опытным разработчикам потребуется развивать шаблоны и методы, которые они используют для правильной обработки конфигурации приложения. Приложения для облачной среды по своей природе более распределены, по сравнению с тем, какими они были прежде. Мы адаптируемся к увеличению рабочих нагрузок, запуская дополнительные экземпляры приложения, вместо того чтобы выделять, например, больше ресурсов для одного. Сама по себе облачная инфраструктура также постоянно меняется, гораздо больше, чем инфраструктура, на которой мы развертывали приложения в последние несколько десятилетий. Эти основные различия в платформе открывают новые способы, которыми контекстуальные факторы презентуют себя приложению, и, следовательно, требуют новых способов обработки их приложениями. Давайте кратко рассмотрим некоторые из этих различий.

6.1.1. Динамическое масштабирование – увеличение и уменьшение количества экземпляров приложения

В предыдущей главе была представлена концепция нескольких экземпляров приложения, и она уже должна была дать вам хорошее представление о том, как это влияет на ваши проекты. Здесь я хочу обратить ваше внимание на два нюанса, которые влияют на конфигурацию приложения.

Во-первых, хотя в прошлом вы могли развернуть несколько экземпляров приложения, скорее всего, их число было относительно небольшим. Когда к этим экземплярам приложения необходимо было применить конфигурацию, это легко можно было сделать с помощью «ручной» доставки конфигурации (даже если этот вариант далек от идеального). Конечно, вы могли использовать сценарии и другой инструментарий, чтобы применить изменения в конфигурации, но нельзя было избежать использования промышленной автоматизации. Теперь, когда приложения масштабируются до сотен или тысяч экземпляров, и эти экземпляры постоянно перемещаются, полуавтоматический подход больше не будет работать. Проекты вашего программного обеспечения и методы эксплуатации должны гарантировать, что *все* экземпляры приложений работают с одинаковыми значениями конфигурации, и вы должны иметь возможность обновлять эти значения с нулевым временем простоя приложения.

Второй фактор еще интереснее. До сих пор я рассматривала конфигурацию приложения с точки зрения результатов, генерируемых приложением. Я сосредоточилась на необходимости того, чтобы какой-либо экземпляр приложения давал одинаковый вывод при одинаковом вводе. Но конфигурация приложения также может заметно повлиять на то, как потребители этого приложения находят его и подключаются к нему. Если говорить прямо, если IP-адрес и/или порт экземпляра приложения изменяются, и в какой-то мере мы учитываем эти (системные) данные конфигурации, несет ли приложение ответственность за то, чтобы эти изменения были известны всем возможным потребителям приложения? Краткий ответ – да, и я расскажу об этом, хотя и не полностью, до глав 8 и 9 (в этом состоит суть обнаружения служб). Пока просто примите во внимание тот факт, что конфигурации приложений имеют сетевой эффект.

6.1.2. Изменения инфраструктуры, вызывающие изменения в конфигурации

Мы все уже слышали это: облако принесло с собой использование низкоуровневых, обычных серверов, а также из-за своей внутренней архитектуры и надежности (или отсутствия таковой) некоторых встроенных компонентов более высокий процент вашей инфраструктуры может выйти из строя в любой момент времени. Все это правда, но аппаратный сбой по-прежнему является лишь одной из причин изменения инфраструктуры, и, вероятно, его малой частью.

Гораздо более частое и необходимое изменение инфраструктуры происходит в ходе обновлений.

Например, приложения все чаще запускаются на платформах, которые предоставляют набор услуг сверх необработанных вычислений, хранилищ и сетей. К примеру, команды разработки приложений (dev и ops) больше не должны предоставлять собственную операционную систему. Вместо этого они могут прос-

то отправить свой код на платформу, которая создаст среду выполнения, чтобы затем развернуть и запустить приложение. Если платформа, которая с точки зрения приложения является частью инфраструктуры, нуждается в обновлении до версии операционной системы (например, из-за уязвимости ОС), то это представляет собой изменение инфраструктуры.

Остановимся на примере обновления ОС, который иллюстрирует множество типов изменений инфраструктуры, поскольку требует, чтобы приложение было остановлено и перезапущено. Это одно из преимуществ приложений для облачной среды: у вас есть несколько экземпляров, которые предотвращают простои системы в целом. Прежде чем убрать экземпляр, который по-прежнему работает на старой ОС, новый экземпляр приложения сначала будет запущен на узле, на котором уже установлена новая версия ОС. Этот новый экземпляр будет работать на другом узле и явно в другом контексте, нежели старый, и ваше приложение и программное обеспечение в целом должны будут адаптироваться. На рис. 6.2 изображены различные этапы этого процесса; обратите внимание, что IP-адрес и порт отличаются до и после обновления.

Обновление – не единственная преднамеренная причина изменений инфраструктуры. Интересный метод обеспечения безопасности, получивший широкое признание, требует частого повторного развертывания экземпляров приложений, потому что при постоянно меняющейся поверхности атаки труднее проникнуть в систему, чем при долгоживущей[1].

6.1.3. Обновление конфигурации приложения с нулевым временем простоя

До сих пор я приводила примеры изменений, которые, если хотите, были применены к приложению из внешнего источника. Масштабирование экземпляров приложения не является непосредственным применением изменения к этому экземпляру. Вместо этого наличие нескольких экземпляров налагает контекстную изменчивость на весь этот набор.

Но иногда приложение, работающее в рабочем окружении, просто требует применения новых значений конфигурации. Например, веб-приложение может отображать авторские права внизу каждой страницы, а когда после декабря наступает январь, вам нужно обновить дату без повторного развертывания всего приложения.

Переключение между наборами учетных данных, при котором пароли, используемые одним системным компонентом для получения доступа к другому, периодически обновляются, служит еще одним примером, который обычно требуется при обеспечении безопасности организации. Это должно быть так же просто, как располагать командой, которая эксплуатирует приложение в рабочем окружении (и как ожидается, это та же команда, которая его создала!), предоставила новые секреты, пока система как единое целое продолжает работать в обычном режиме.

---

[1] Для получения дополнительной информации см. статью Джастина Смита «The Three Rs of Enterprise Security: Rotate, Repave, and Repair» на странице **http://mng.bz/gNKe**.

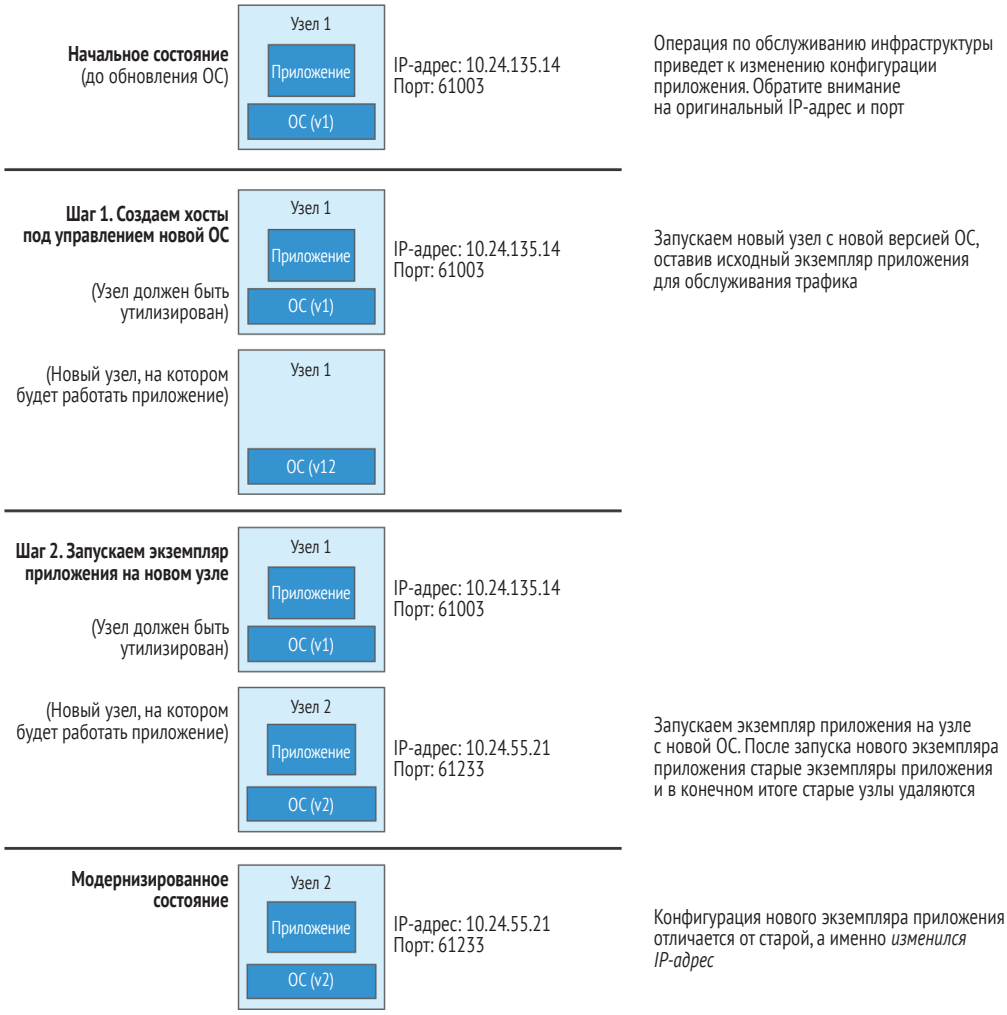


Рис. 6.2 ❖ Изменения в конфигурации приложения часто происходят из-за изменений в инфраструктуре, которые являются либо ожидаемыми (как показано здесь, последовательное обновление), либо неожиданными

Такие типы контекстных изменений представляют собой изменения в данных конфигурации приложения и, в отличие от таких вещей, как изменения в инфраструктуре, обычно находятся под контролем самой команды разработки приложения. Это различие может вызвать у вас соблазн обращаться с этим типом изменчивости в более «ручной» манере. Но, как вы скоро увидите, с точки зрения приложения обработка преднамеренных и навязанных изменений с использованием аналогичных подходов не только возможна, но и крайне желательна.

Вот в чем состоит хитрость во всех этих сценариях: создать правильные абстракции, параметризируя развертывание приложения, чтобы элементы, которые будут различаться в зависимости от контекста, могли быть разумно внедрены

в приложение в нужное время. Как и в случае с любым шаблоном, ваша цель состоит в том, чтобы иметь проверенный, протестированный и повторяемый способ проектирования для этих требований.

Начать с этого повторяющегося шаблона можно в самом приложении, создав метод, позволяющий четко определить точные данные конфигурации для приложения, что позволит вводить значения по мере необходимости.

6.2. Уровень конфигурации приложения

Если вы читали какую-нибудь книгу о программном обеспечении для облачной среды, то почти наверняка слышали о 12-факторном приложении (**https://12factor.net**) – наборе шаблонов и методов, рекомендуемых для приложений на базе микросервисов. Одним из наиболее часто упоминаемых факторов является фактор № 3: «Сохраняйте конфигурацию в среде выполнения». Читая, по общему признанию, краткое описание по адресу **https://12factor.net/config**, видно, что в нем советуется сохранять данные конфигурации для приложения в переменные среды.

Частью обоснованного аргумента для такого подхода является то, что практически все операционные системы поддерживают концепцию переменных среды, и все языки программирования предлагают способ доступа к ним. Это не только обеспечивает переносимость приложения, но и может сформировать основу для согласованных методов эксплуатации, независимо от типов систем, на которых работает ваше приложение. Следуя этому руководству на Java, вы могли бы иметь такой код, например, для доступа и использования данных конфигурации, хранящихся в этих переменных среды:

```
public Iterable<Post> getPostsByUserId(
    @RequestParam(value=”userIds”, required=false) String userIds,
    HttpServletResponse response) {
    String ip;
    ip = System.getenv(“INSTANCE_IP”);
    ...
}
```

Хотя этот подход, безусловно, позволит использовать ваш код в разных средах, из этого простого совета или, по крайней мере, из предыдущей реализации возникает пара недостатков. Первое: переменные среды – не лучший вариант для всех типов данных конфигурации. Вскоре вы увидите, что они хорошо подходят для конфигурации системы, и в меньшей степени – для конфигурации приложения. И второе: распространение вызовов `System.getenv` (или чего-то похожего в других языках) по всей вашей кодовой базе затрудняет отслеживание конфигурации вашего приложения.

Лучшим подходом является наличие определенного уровня конфигурации в вашем приложении – места, куда вы можете обратиться, чтобы увидеть параметры конфигурации приложения. По мере прохождения этой главы вы увидите, что существуют различия в способах настройки системной среды и конфигурации приложения, но этот слой конфигурации приложения является общим для обоих (рис. 6.3). В Java для этого используются файлы свойств, и большинство языков предоставляет аналогичную конструкцию.

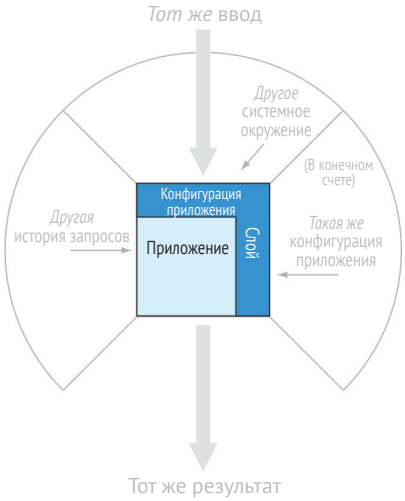


Рис. 6.3 ❖ Приложения имеют определенный уровень конфигурации, который поддерживает и системное окружение, и конфигурацию приложения. Этот уровень позвоеляет реализации использовать значения независимо от того, как предоставляются их значения

Хотя вы почти наверняка знакомы с использованием файлов свойств, у меня есть способ, которым я хочу поделиться, – он поможет вам, когда мы подробно будем рассматривать различия между конфигурацией системы и приложения позже в этой главе.

Самым большим преимуществом подхода, который я здесь описываю, может быть то, что файлы свойств представляют собой единое логическое место для всех параметров конфигурации, которые должны быть определены. (У вас может быть несколько файлов свойств, но, как правило, все они размещены в одном месте в структуре проекта.) Это позволяет разработчику или оператору приложения легко просматривать и осваивать параметры конфигурации приложения. Помните мое предыдущее замечание касательно того, что вызовы `System.getenv` разбросаны по всему телу кода? Представьте, что вы разработчик, работающий с существующей кодовой базой, и вам нужно «пролистать» десятки файлов исходного кода, чтобы увидеть, какие данные действуют в качестве ввода для приложения. Брр! Файлы свойств могут оказаться полезными.

Самым большим недостатком способа использования файлов свойств в настоящее время является то, что они обычно объединяются в развертываемый артефакт (в случае с Java – в файл JAR), а файлы свойств часто содержат фактические значения конфигурации. Вспомните из главы 2, что одним из ключей к оптимизации жизненного цикла приложения «от разработки до эксплуатации» является то, что у вас есть один развертываемый артефакт, который используется по всему циклу. Поскольку в разных средах разработки, тестирования и производства контекст будет разный, у вас может возникнуть соблазн иметь разные файлы свойств для каждой среды, но тогда вам понадобятся разные сборки и разные развертываемые артефакты. Сделайте это, и вы снова вернетесь к общеизвестной фразе: «На моей машине все работает».

ПОДСКАЗКА Хорошая новость состоит в том, что у вас есть альтернатива наличию разных файлов свойств для разных развертываний.

Вот тут-то и вступает в действие уловка, о которой я говорила. Сначала я рассматривала файл свойств как спецификацию данных конфигурации для приложения, а затем как шлюз к контексту приложения. Файл свойств определяет переменные, которые могут использоваться повсеместно в коде, а значения привязываются к этим переменным из наиболее подходящих источников (системная среда или приложение) и в нужное время. Все языки предоставляют средства для доступа к переменным, определенным в этих файлах свойств по всему коду. Я уже использовала эту технику в примерах кода.

Давайте посмотрим на файл application.properties для службы Posts.

Листинг 6.1 ❖ application.properties

```
management.security.enabled=false
spring.jpa.hibernate.ddl-auto=update
spring.datasource.username=root
spring.datasource.password=password
ipaddress=127.0.0.1
```

Чтобы проследить поток из файлов свойств в код, давайте подробнее рассмотрим свойство `ip-address`. Я еще не обращала на это вашего внимания, но я выводила IP-адрес, по которому экземпляр приложения обслуживает трафик, в выводе журнала. Когда вы запускаете это программное обеспечение локально, будет выведено значение `127.0.0.1`. Но вы, возможно, заметили, что при развертывании служб в Kubernetes файлы журналов сообщали об одном и том же IP-адресе неправильно, потому что я делала то, о чем только что говорила, не совсем так, как надо: речь идет о привязке значений к этим переменным непосредственно в файле свойств. Совсем скоро я это исправлю. Я расскажу о том, как наши свойства получают свои значения в следующих двух разделах. Прямо сейчас я хочу сосредоточиться на файле свойств в качестве абстракции для реализации приложения. В файле PostsController.java вы найдете приведенный ниже код.

Листинг 6.2 ❖ PostsController.java

```
public class PostsController {

    private static final Logger logger
        = LoggerFactory.getLogger(PostsController.class);
    private PostRepository postRepository;

    @Value("${ipaddress}")
    private String ip;

    @Autowired
    public PostsController(PostRepository postRepository) {
        this.postRepository = postRepository;
    }
...
}
```

Локальная переменная `ip` извлекает свое значение из переменной среды `ipaddress`. Spring предоставляет для этого аннотацию `@Value`, чтобы упростить процесс.

Собираем части воедино: исходный код приложения определяет элементы данных, у которых могут быть введены значения, и извлекает эти значения из определенных свойств. В файле свойств перечислены все параметры конфигурации не только для того, чтобы облегчить ввод этих значений в приложение, но и с целью предоставить разработчику или оператору спецификацию данных конфигурации для приложения.

Но, опять же, совсем нехорошо, что у вас есть значение `127.0.0.1`, жестко закодированное в файле свойств. Некоторые языки, например Java, дают ответ на этот вопрос, позволяя переопределять значения свойств при запуске приложения. Например, вы можете запустить службу Posts с помощью этой команды, предоставив новое значение для `ipaddress`:

```
java -Dipaddress=192.168.3.42 \
     -jar cloudnative-posts/target/cloudnative-posts-0.0.1-SNAPSHOT.jar
```

Но я хочу снова привлечь ваше внимание к фактору № 3: «Сохраняйте конфигурацию в среде выполнения». Этот совет указывает на нечто важное. Это правда, что перемещение привязок значений из файлов свойств в командную строку устраняет необходимость наличия разных сборок для разных сред, но разные команды запуска теперь предлагают ошибкам конфигурации новый способ проникнуть в вашу эксплуатацию. Если вместо этого вы храните IP-адрес в переменной среды, можно использовать приведенную ниже команду для запуска приложения в любой из сред. Приложение просто поглотит контекст, в котором оно работает:

```
java -jar cloudnative-posts/target/cloudnative-posts-0.0.1-SNAPSHOT.jar
```

Некоторые языковые платформы поддерживают отображение переменных среды в свойства приложения.

Например, в Spring Framework установка переменной среды `IPADDRESS` приведет к тому, что это значение будет введено в свойство `ipaddress`. Мы дойдем до этого, но я собираюсь добавить еще одну абстракцию, чтобы дать вам еще больше гибкости и обеспечить ясность кода. Я хочу обновить строку `ipaddress` в файле свойств:

```
ipaddress=${INSTANCE_IP:127.0.0.1}
```

Теперь в этой строке указано, что значение `ipaddress` будет специально получено из переменной среды `INSTANCE_IP`, и если эта переменная не определена, для `ipaddress` будет установлено значение *по умолчанию* `127.0.0.1`. Видите, ничего страшного, если в файле свойств есть значения, пока они представляют разумные значения по умолчанию, и вы планируете то, как значения будут переопределены, если значение по умолчанию неверно.

Давайте соединим все это воедино в диаграмме – рис. 6.4. Исходный код приложения ссылается на свойства, которые определены в файле свойств. Файл свойств действует как спецификация параметров конфигурации приложения и будет четко указывать, какие значения могут поступать из переменных среды.

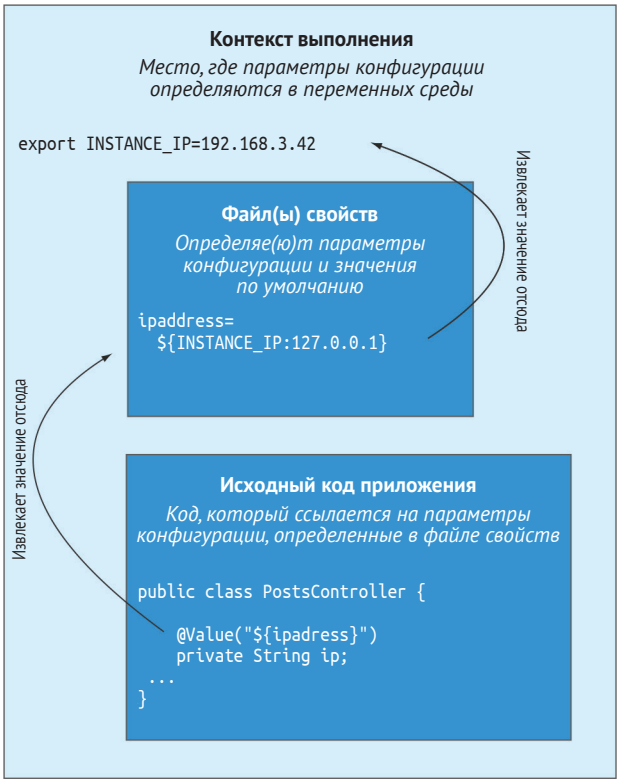


Рис. 6.4 ❖ Исходный код приложения ссылается на свойства, которые определены в файле свойств. Файл свойств действует как спецификация параметров конфигурации приложения и может указывать на то, что значения должны поступать из переменных среды (`INSTANCE_IP`)

Файлы свойств, написанные таким образом, скомпилированы в один развертываемый артефакт, который теперь можно инстанциировать в любой среде. Артефакт специально разработан для поглощения контекста этой среды. Это добродетель – и ключевой шаблон для правильной настройки ваших приложений в облачном контексте!

Но я не все вам рассказала. Все, что я сказала, на 100 % верно, но вследствие опущения я подразумевала, что файлы свойств всегда получают исходные значения из переменных среды (хотя я и намекнула, что это не всегда так). Это всего лишь одно место, откуда могут поступать данные конфигурации. Есть и альтернативы. И различия, как правило, проводятся в соответствии с тем, есть ли у вас системная конфигурация или данные конфигурации приложения. Давайте посмотрим на них.

6.3. Инъекция значений системы/среды

Под *системными значениями* я подразумеваю значения, над которыми разработчик приложения или оператор не имеет прямого контроля. Ого! Что? В мире, где я провела большую часть карьеры, это абсолютно безумная концепция. Компьютеры и компьютерные программы детерминированы, и если вы подаете все вводные данные одинаково, то можете полностью контролировать вывод. Предложение передать часть этого контроля поставило бы многих профессионалов программного обеспечения в неудобное положение. Но переход в облако требует именно этого, что возвращает нас к концепции, о которой я говорила в главе 2: *изменение – это правило, а не исключение*. Частичный отказ от контроля также позволяет системам работать более независимо, в конечном счете делая доставку программного обеспечения более гибкой и продуктивной.

Системные переменные отражают ту часть контекста приложения, которая обычно предоставляется инфраструктурой. Я полагаю, что это представляет состояние инфраструктуры. Как мы уже обсуждали, наша работа в качестве разработчиков заключается в обеспечении согласованности результатов приложений, несмотря на то что они работают в контексте, который неизвестен априори и постоянно меняется.

Чтобы исследовать это немного подробнее, давайте рассмотрим конкретный пример: включение IP-адреса в вывод журнала. В прошлом вы, возможно, не рассматривали IP-адрес как нечто постоянно меняющееся, но в облаке он таковым и является. Экземпляры приложения постоянно создаются и каждый раз получают новый IP-адрес. Включение IP-адреса в вывод журнала особенно интересно при запуске в облачной среде, поскольку позволяет отслеживать, какой конкретный экземпляр приложения обслуживает определенный запрос.

6.3.1. Давайте посмотрим, как это работает: использование переменных среды для конфигурации

Для начала я приглашаю вас обратно в репозиторий cloudnative-abundantsunshine, в частности в каталог и модуль cloudnative-appconfig. Рассматривая реализацию службы Connections' Posts, видно, что файл свойств уже отражает определение `ipaddress`, показанное в предыдущем разделе. Вот как это выглядит:

```
ipaddress=${INSTANCE_IP:127.0.0.1}
```

Приложению требуется значение `ipaddress`, и у инфраструктуры такое значение есть. Как тогда соединить их? Вот где срабатывает фактор № 3: переменные среды являются постоянными практически во всех средах; инфраструктура и платформы знают, как их предоставить, а фреймворки приложений знают, как их потреблять. Использование этой вездесущности важно. Это позволяет вам устанавливать передовые методы независимо от того, работает ли ваше приложение в Linux (любой версии!), macOS или Windows.

Чтобы увидеть все это в действии, я хочу развернуть последнюю версию приложения в Kubernetes.

Настройка

Как и в предыдущих главах, для запуска примеров необходимо установить следующие стандартные инструменты:

- Maven;
- Git;
- Java 1.8;
- Docker;
- какой-нибудь тип клиента MySQL, например интерфейс командной строки `mysql`;
- какой-нибудь тип клиента Redis, например `redis-cli`;
- Minikube.

Создание микросервисов (не обязательно)

Мы будем разворачивать приложения в Kubernetes. Для этого требуются образы Docker, поэтому я предварительно создала их и сделала их доступными в Docker Hub. Поэтому создание микросервисов из исходного кода не требуется. Тем не менее изучение кода – иллюстративный процесс, поэтому я предлагаю вам выполнить некоторые из этих шагов, даже если вы сами не создавали код.

В каталоге cloudnative-abundantsunshine скачайте приведенный ниже тег и перейдите в каталог cloudnative-appconfig:

```
git checkout appconfig/0.0.1
cd cloudnative-appconfig
```

Затем, чтобы собрать код (необязательно), введите приведенную ниже команду:

```
mvn clean install
```

В ходе выполнения этой команды будет выполнена сборка трех приложений с созданием файла JAR в целевом каталоге каждого модуля. Если вы хотите развернуть эти JAR-файлы в Kubernetes, вы также должны выполнить команды `docker build` и `docker push`, как описано в разделе «Использование Kubernetes требует создания образов Docker» в главе 5. Если вы это сделаете, вам также нужно будет обновить файлы YAML развертывания Kubernetes, чтобы указать путь к своим образам, а не к моим. Я не буду повторять здесь эти шаги. В предоставленных мною манифестах развертывания указан путь к образам, хранящимся в моем репозитории Docker Hub.

Запуск приложений

Запустите Minikube, если вы еще этого не сделали, как описано в разделе 5.2.2 главы 5. Чтобы начать с чистого листа, удалите все развертывания и службы, которые могли остаться после предыдущей работы. Для этого я предоставила вам скрипт: `delete-DeploymentComplete.sh`. Этот простой bash-скрипт позволяет поддерживать работоспособность служб MySQL и Redis. При вызове скрипта без параметров удаляются только три развертывания микросервисов; при вызове скрипта с аргументом `all` MySQL и Redis также удаляются. Убедитесь, что ваша среда очищена с помощью этой команды:

```
$kubectl get all
NAME                          READY STATUS    RESTARTS AGE
pod/mysql-75d7b44cd6-jzgsk    1/1   Completed 0        2d3h
pod/redis-6bb75866cd-tzfms    1/1   Completed 0        2d3h

NAME                 TYPE           CLUSTER-IP   EXTERNAL-IP PORT(S)      AGE
service/kubernetes   ClusterIP      10.96.0.1    <none>      443/TCP      2d5h
```

```
service/mysql-svc     NodePort       10.107.78.72  <none>   3306:30917/TCP 2d3h
service/redis-svc     NodePort       10.108.83.115 <none>   6379:31537/TCP 2d3h

NAME                  READY UP-TO-DATE AVAILABLE AGE
deployment.apps/mysql 1/1    1         1         2d3h
deployment.apps/redis 1/1    1         1         2d3h

NAME                              DESIRED CURRENT READY AGE
replicaset.apps/mysql-75d7b44cd6 1       1       1     2d3h
replicaset.apps/redis-6bb75866cd 1       1       1     2d3hNAME
```

Обратите внимание, что MySQL и Redis остались в рабочем состоянии. Если вы удалили Redis и MySQL, разверните их с помощью этих команд:

```
kubectl create -f mysql-deployment.yaml
kubectl create -f redis-deployment.yaml
```

После завершения развертывание будет выглядеть так, как показано на рис. 6.5.

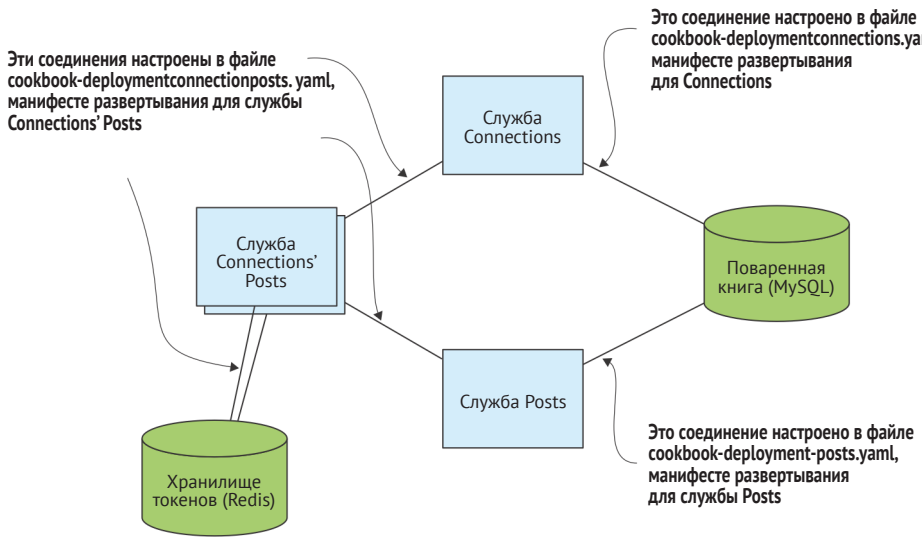


Рис. 6.5 ❖ Эта топология развертывания программного обеспечения в настоящее время требует большого количества ручного редактирования соединений между службами. Ручные конфигурации будут постепенно устраняться по мере изучения шаблонов, более привычных для облачной среды

У вас будет по одной службе Connections and Posts и два экземпляра службы Connections' Posts. Для достижения этой топологии на данный момент вам, возможно, по-прежнему придется редактировать манифесты развертывания. Эти шаги, кратко изложенные ниже, подробно описаны в главе 5:

1) настройте службу Connections так, чтобы она указывала на базу данных MySQL. Найдите URL-адрес с помощью этой команды и вставьте его в соответствующее место в манифесте развертывания:

```
minikube service mysql-svc \
  --format "jdbc:mysql://{{.IP}}:{{.Port}}/cookbook"
```

2) разверните службу Connections:

```
kubectl create -f cookbook-deployment-connections.yaml
```

3) настройте службу Posts таким образом, чтобы она указывала на базу данных MySQL. Используйте тот же URL-адрес, который вы получили с помощью команды в шаге 1, и вставьте его в соответствующее место в манифесте развертывания;
4) разверните службу Posts:

```
kubectl create -f cookbook-deployment-posts.yaml
```

5) настройте службу Connections' Posts таким образом, чтобы она указывала на службы Posts, Connections и Users, а также на службу Redis. Эти значения можно найти с помощью приведенных ниже команд:

URL-адрес службы Posts	`minikube service posts-svc --format «http://{{.IP}}:{{.Port}}/posts?userIds=» --url`
URL-адрес службы Connections	`minikube service connections-svc --format «http://{{.IP}}:{{.Port}}/connections/» --url`
URL-адрес службы Users	`minikube service connections-svc --format «http://{{.IP}}:{{.Port}}/users/» --url`
IP-адрес Redis	`minikube service redis-svc --format «{{.IP}}»`
Порт Redis	`minikube service redis-svc --format «{{.Port}}»`

6) разверните службу Connections' Posts:

```
kubectl create -f cookbook-deployment-connectionsposts.yaml
```

На этом развертывание завершено, но я бы хотела обратить ваше внимание на строки в манифестах развертывания, которые касаются обсуждаемой темы: настройка системных значений. Ниже приводится фрагмент манифеста развертывания для службы Connections' Posts:

Листинг 6.3 ❖ cookbook-deploy-connectionsposts.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: connectionsposts
  labels:
    app: connectionsposts
spec:
  replicas: 2
  selector:
    matchLabels:
      app: connectionsposts
  template:
    metadata:
      labels:
        app: connectionsposts
    spec:
      containers:
      - name: connectionsposts
```

```
image: cdavisafc/cloudnative-appconfig-connectionposts:0.0.1
env:
  - name: INSTANCE_IP
    valueFrom:
      fieldRef:
        fieldPath: status.podIP
```

Вы видите раздел с пометкой `env` как часть спецификации этой службы. Да, именно здесь вы определяете переменные среды для контекста, в котором будет работать ваше приложение. Kubernetes поддерживает несколько способов предоставления значения. В случае с `INSTANCE_IP` значение извлекается из атрибутов, предоставленных самой платформой Kubernetes. Только Kubernetes знает IP-адрес модуля (объекта, в котором будет запускаться приложение), и к этому значению можно получить доступ в манифесте развертывания через атрибут `status.podIP`. Когда Kubernetes устанавливает контекст времени выполнения, он заполняет его значением `INSTANCE_IP`, которое, в свою очередь, выводится в приложение с помощью файла свойств.

Рисунок 6.6 резюмирует все вышесказанное. Обратите внимание, что поле с пометкой «Linux Container» точно такое же, что и на рис. 6.4. Здесь вы видите уровень конфигурации приложения, работающий в контексте Kubernetes. На рис. 6.6 показано, как этот контекст взаимодействует с уровнем конфигурации. Эта диаграмма довольно изощренная:

- у Kubernetes есть API, который позволяет предоставить манифест развертывания;
- этот манифест позволяет определять переменные среды;
- после создания развертывания Kubernetes создает модуль и контейнеры внутри модуля и «засеивает» каждый из них множеством значений.

Но, несмотря на относительную сложность, содержимое контейнера Linux остается простым; приложение будет извлекать значения из переменных среды. Оно защищено ото всей этой сложности с помощью переменных среды в качестве абстракции. Вот почему фактор № 3 попадает в самую точку; он управляет простотой и изяществом.

Если вы посмотрите на код листинга 6.3, то увидите, что Java-класс `Utils` используется для генерации тега, объединяющего IP-адрес и порт, на котором работает приложение. Этот тег затем включается в вывод журнала. Когда создается экземпляр данного класса, контейнер Linux уже запущен, включая установленную переменную среды `INSTANCE_IP`. Это приводит к инициализации свойства `ipaddress`, которое затем помещается в класс `Utils` с аннотацией `@Value`. Хотя это не относится к теме переменных среды, для полноты изложения я также отмечу, что создала класс `ApplicationContextAware` и реализовала слушателя, который ожидает инициализации встраиваемого контейнера сервлетов. А в это время порт, на котором запущено приложение, уже установлен, и его можно найти через `EmbeddedServletContainer`.

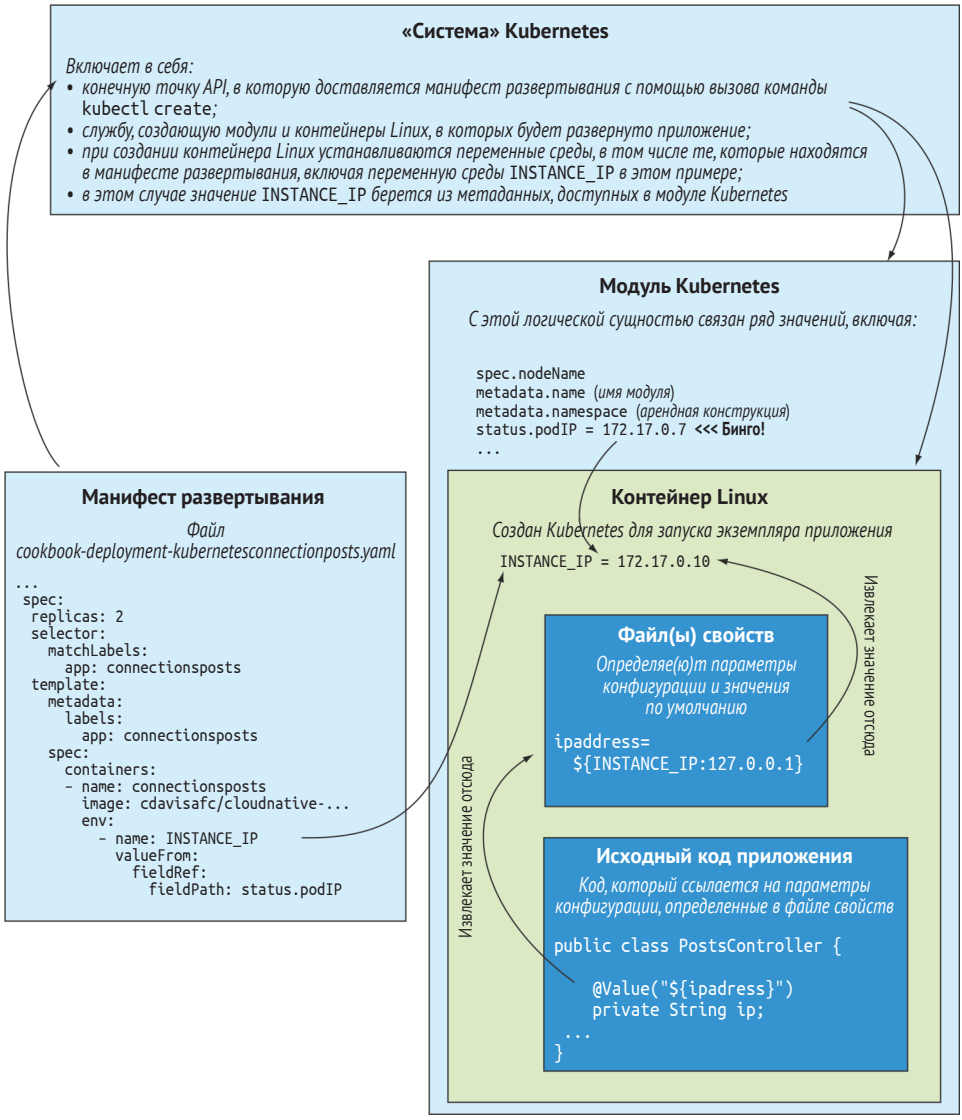


Рис. 6.6 ❖ Ответственный за развертывание и управление экземплярами приложения, Kubernetes устанавливает переменные среды, определенные в манифесте развертывания, извлекая значения из сущностей инфраструктуры, которые он установил для приложения

Листинг 6.4 ❖ Utils.java

```
public class Utils implements ApplicationContextAware,
                    ApplicationListener<ServletWebServerInitializedEvent> {

    private ApplicationContext applicationContext;
    private int port;
    @Value("${ipaddress}")
    private String ip;

    public String ipTag() {
        return "[" + ip + ":" + port +"] ";
    }

    @Override
    public void setApplicationContext(
                          ApplicationContext applicationContext)
                                                  throws BeansException {
        this.applicationContext = applicationContext;
    }

    @Override
    public void onApplicationEvent(ServletWebServerInitializedEvent
                                embeddedServletContainerInitializedEvent) {
        this.port = embeddedServletContainerInitializedEvent
                        .getApplicationContext().getWebServer().getPort();
    }
}
```

Хорошо, теперь пришло время увидеть, как все это работает.

Если вы заново создали службу MySQL, не забудьте создать базу данных поваренной книги, подключившись к серверу с помощью клиента MySQL и выполнив команду `create database`. Например:

```
$mysql -h $(minikube service mysql-svc --format «{{.IP}}») \
 -P $(minikube service mysql-svc --format «{{.Port}}») -u root -p
mysql> create database cookbook;
Query OK, 1 row affected (0.00 sec)
```

В дополнение к тому, что я подробно описала здесь, вы можете осуществлять потоковую передачу журналов из служб Connections и Posts, но самом деле я хочу обратить внимание на вывод журнала службы Connections' Posts. Давайте вызовем эту службу несколько раз. Напомню, что первым шагом является аутентификация, а затем вы можете получить доступ к постам с помощью простой команды `curl`:

```
# authenticate
curl -X POST -i -c cookie \
   $(minikube service --url connectionsposts-svc)/login?username=cdavisafc
# get the posts - repeat this command 4 or 5 times
curl -i -b cookie \
   $(minikube service --url connectionsposts-svc)/connectionsposts
```

Kubernetes не поддерживает агрегированную потоковую передачу журналов, поэтому я предложила вам несколько раз вызвать службу, прежде чем просмат-

ривать журналы. Однако теперь вы можете просмотреть журналы обоих экземпляров с помощью одной команды:

```
$ kubectl logs -lapp=connectionsposts
...
... : Tomcat started on port(s): 8080 (http) with context path ‘’
... : Started CloudnativeApplication in 16.502 seconds
... : Initializing Spring FrameworkServlet ‘dispatcherServlet’
... : FrameworkServlet ‘dispatcherServlet’: initialization started
... : FrameworkServlet ‘dispatcherServlet’: initialization completed
... : Starting without optional epoll library
... : Starting without optional kqueue library
... : [172.17.0.7:8080] getting posts for user network cdavisafc
... : [172.17.0.7:8080] connections = 2,3
... : [172.17.0.7:8080] getting posts for user network cdavisafc
... : [172.17.0.7:8080] connections = 2,3
...
... : Started CloudnativeApplication in 15.501 seconds
... : Initializing Spring FrameworkServlet ‘dispatcherServlet’
... : FrameworkServlet ‘dispatcherServlet’: initialization started
... : FrameworkServlet ‘dispatcherServlet’: initialization completed
... : Starting without optional epoll library
... : Starting without optional kqueue library
... : [172.17.0.4:8080] getting posts for user network cdavisafc
... : [172.17.0.4:8080] connections = 2,3
... : [172.17.0.4:8080] getting posts for user network cdavisafc
... : [172.17.0.4:8080] connections = 2,3
```

Глядя на этот пример, видно, что у вас есть вывод обоих экземпляров службы Connections' Posts. Журналы не чередуются. Эта команда просто обращается к журналам из одного экземпляра и выгружает их, а затем делает то же самое для следующего экземпляра. Тем не менее можно увидеть, откуда идет вывод из двух разных экземпляров, потому что IP-адреса каждого из них были зарегистрированы; один экземпляр имеет IP-адрес 172.17.0.7, а другой – 172.17.0.4. Здесь видно, что два запроса отправились к экземпляру, обслуживающему трафик по адресу 172.17.0.4, а два запроса отправились к экземпляру на 172.17.0.7. Kubernetes инстанциировал значения в переменных среды, присутствующих в контексте каждого экземпляра, и приложение получило значение через файлы свойств, которые были созданы для доступа к переменным среды. Это не плохая схема.

Давайте посмотрим на переменные среды в работающих контейнерах. Это можете сделать, выполнив приведенную ниже команду, заменив имя модуля своим собственным:

```
$ kubectl exec connectionsposts-6c69d66bb6-f9bjn -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin...
CONNECTIONPOSTSCONTROLLER_POSTSURL=http://192.168.99.100:32119/posts?userIds=
CONNECTIONPOSTSCONTROLLER_CONNECTIONSURL=http://192.168.99.100:30955/connections/
CONNECTIONPOSTSCONTROLLER_USERSURL=http://192.168.99.100:30955/users/
REDIS_HOSTNAME=192.168.99.100
REDIS_PORT=31537
INSTANCE_IP=172.17.0.7
KUBERNETES_PORT_443_TCP_PROTO=tcp
...
```

Среди длинного списка отображаемых значений вы увидите INSTANCE_IP. По вашему указанию (вспомните строки в файле развертывания YAML службы Connections' Posts) Kubernetes установил это значение в качестве IP-адреса модуля. Это значение конфигурации приложения, установленное системой, в которой работает ваше приложение.

Надеюсь, данное упражнение помогло прояснить ситуацию. Но даже в этом случае я хочу предложить еще один ценный инструмент. Я еще не говорила вам, что у меня есть нечто, работающее как часть каждой из служб. Благодаря магии Spring Boot приложение автоматически реализует конечную точку, где вы можете просматривать среду, в которой работают ваши приложения. Выполните эту команду, чтобы увидеть вывод:

```
curl $(minikube service --url connectionsposts-svc)/actuator/env
```

Вывод в формате JSON не маленький, но вы увидите, что он включает в себя это:

```
...
  "systemEnvironment": {
    "PATH": "/usr/local/sbin:/usr/local/bin:...",
    "INSTANCE_IP": "172.17.0.7",
    "PWD": "/",
    "JAVA_HOME": "/usr/lib/jvm/java-1.8-openjdk",
    ...
  },
  "applicationConfig: [classpath:/application.properties]": {
    ...
    "ipaddress": "172.17.0.7"
  },
...
```

Среди доступных данных есть IP-адрес, которым вы манипулируете. Под строкой «systemEnvironment»: виден ключ INSTANCE_IP со значением 172.17.0.7, а под ключом applicationConfig вы видите строку, которая включает в себя тот же адрес, ассоциированный с ключом ipaddress. Соединение было установлено так, как вы и планировали.

Просматривая другие значения в этом выводе, кажется, что здесь есть много переменных окружения, и это действительно так, но вместе с тем видно, что сообщается и о многих других контекстных значениях. Например, можно увидеть идентификатор процесса (PID), версию операционной системы (os.version) и многие другие значения, которые не хранятся в переменных среды. Это говорит о том, что переменные среды не единственные контекстные значения вашего приложения. Конечная точка /actator/env сообщает о более расширенном наборе. Теперь я хотела бы перейти к другой части контекста приложения и рассмотреть иной способ внедрения значений.

6.4. Внедрение конфигурации приложения

Для типа данных конфигурации, который вы только что видели, со значениями, являющимися частью среды выполнения и управляющимися платформой времени выполнения, использование переменных среды – естественный и эффективный процесс. Но когда я впервые начала работать с системами для облачной

среды, то столкнулась с рационализацией фактора № 3 (сохраняйте конфигурацию в среде выполнения) с помощью ряда других вещей, которые нам нужны для управления конфигурацией приложения. В конечном итоге ответ заключается в том, что существуют более эффективные способы управления данными конфигурации приложений. Об этом я и хочу рассказать вам сейчас.

Когда дело доходит до запуска приложения в рабочем окружении, я бы сказала, что данные конфигурации так же важны, как и сама реализация, поскольку без надлежащей конфигурации программное обеспечение просто не будет работать. Это требует применения того же уровня строгости в управлении данными конфигурации приложения, как и в случае с управлением кодом. А именно:

- данные должны быть сохранены, а доступ контролироваться. Это настолько похоже на способ обработки исходного кода, что одним из наиболее распространенных инструментов, используемых для этого, является система управления версиями (SCC), например Git;
- вы никогда не будете устанавливать данные конфигурации вручную. Если вам нужно изменить конфигурацию, вы внесете изменения в репозиторий Git и вызовете некое действие, чтобы применить эту конфигурацию к работающей системе;
- конфигурации должны иметь несколько версий, чтобы у вас была возможность последовательно создавать развертывания повторно, основываясь исключительно на конкретной версии приложения, связанной с определенной версией конфигурации. Также важно знать, какие свойства использовались и в какое время, чтобы эксплуатационное поведение (хорошее и плохое) можно было сопоставить с примененной конфигурацией;
- некоторые данные конфигурации являются конфиденциальными, например учетные данные, используемые для обмена данными между компонентами в распределенной системе. Это приводит к дополнительным требованиям, которые требуют использования специальных хранилищ конфигурации, таких как Vault от компании HashiCorp.

Первая часть ответа на вопрос, как управлять данными конфигурации приложения, заключается в том, что они управляются в месте, которое я называю *хранилищем данных конфигурации*. (Я избегаю использования термина *база данных управления конфигурацией*, потому что он несет с собой некий груз и подразумевает определенные шаблоны, которые не применяются в мире облачной среды.) Хранилище конфигурации будет просто содержать пары типа «ключ/значение», вести историю версий и применять различные механизмы контроля доступа.

И вторая часть ответа заключается в том, что некая служба облегчает доставку этих данных с версиями и контролем доступа в приложение. Эта служба предоставляется сервером конфигурации. Давайте приступим к добавлению ее в наш пример.

6.4.1. Знакомство с сервером конфигурации

На данный момент наша реализация предлагает некоторый уровень контроля в службе Connections' Posts. Пользователь должен пройти аутентификацию, прежде чем служба предоставит результаты. Но две службы, которые в конечном итоге предоставляют данные, службы Connections и Posts, остаются широко открытыми. Давайте обезопасим их с помощью секретов. Мы будем использовать секре-

ты вместо процесса аутентификации и авторизации пользователя, поскольку эти службы не будут вызываться конкретным пользователем, выполнившим вход, а будут вызываться другим программным модулем (в нашем случае службой Connections' Posts). Например, здесь вы используете службу Posts, чтобы получить посты для группы пользователей, на которую подписан зарегистрированный пользователь, но в другой настройке вы можете использовать ту же службу, чтобы получать посты от любых блогеров, которые на данный момент находятся в тренде.

В этом примере я реализовала секреты, настроив секрет в обеих службах (службы Connections и Posts), а также сконфигурировав один и тот же секрет в клиенте (служба Connections' Posts). Прежде чем подробно рассмотреть реализацию, давайте сначала посмотрим, как управлять этими значениями.

Во-первых, вам нужно создать репозиторий исходного кода для хранения секретов. Можно создать его с нуля, или, чтобы упростить задачу, можете клонировать суперпростой репозиторий, который у меня есть, на странице **https://github.com/cdavisafc/cloud-native-config.git**. Вам нужно будет клонировать его, чтобы иметь возможность фиксировать изменения во время выполнения упражнения. Там вы увидите нечто, что подозрительно похоже на файл свойств: mycookbook.properties. Этот файл содержит два значения: секрет, который будет защищать службу Posts, и секрет, который будет защищать службу Connections:

```
com.corneliadavis.cloudnative.posts.secret=123456
com.corneliadavis.cloudnative.connections.secret=456789
```

Теперь мы установим службу, которая будет управлять доступом к этим значениям конфигурации, и для этого будем использовать Spring Cloud Confguration (**https://github.com/springcloud/spring-cloud-config**). Spring Cloud Configuration Server (SCCS) – это реализация с открытым исходным кодом, которая хорошо подходит для управления данными для распределенных систем (программное обеспечение для облачной среды). Он работает как веб-служба на базе протокола HTTP и обеспечивает поддержку организации данных на протяжении всего жизненного цикла доставки программного обеспечения. Я отсылаю вас к файлу README в репозитории для получения более подробной информации, а здесь я продемонстрирую несколько ключевых возможностей.

Давайте начнем собирать фрагменты воедино. Сначала скачайте приведенный ниже тег репозитория:

```
git checkout appconfig/0.0.2
```

Далее давайте запустим сервер конфигурации Spring Cloud. К счастью, для этого сервера уже есть образ Docker, и я предоставила вам манифест развертывания для Kubernetes. Перед созданием модуля с помощью обычной команды клонируйте репозиторий на странице **https://github.com/cdavisafc/cloudnative-config.git**, а затем замените URL-адрес в приведенном ниже фрагменте манифеста развертывания на URL-адрес вашего репозитория:

```
        env:
          - name: SPRING_CLOUD_CONFIG_SERVER_GIT_URI
            value: https://github.com/cdavisafc/cloud-native-config.git
```

После этого создайте службу с помощью команды

```
kubectl create -f spring-cloud-config-server-deployment.yaml
```

После запуска сервера вы можете получить доступ к конфигурациям с помощью этой команды:

```
$ curl $(minikube service --url sccs-svc)/mycookbook/dev | jq
{
  “name”: “mycookbook”,
  “profiles”: [
    “dev”
  ],
  “label”: null,
  “version”: “67d9531747e46b679cc580406e3b48b3f7024fc8”,
  “state”: null,
  “propertySources”: [
    {
      “name”: “https://github.com/cdavisafc/cloud-native-config.git/mycookbook.properties”,
      “source”: {
        “com.corneliadavis.cloudnative.connections.secret”: “456789”,
        “com.corneliadavis.cloudnative.posts.secret”: “123456”
      }
    }
  ]
}
```

Сервер конфигурации Spring Cloud поддерживает конфигурации тегирования как с метками Git, так и с профилями приложений. Мой репозиторий с примерами конфигурации включает в себя два файла конфигурации для приложения `mycookbook` – один для среды разработки и один для рабочего окружения. Выполнив предыдущую команду `curl` и заменив `/dev` на `/prod`, вы увидите значения для рабочего профиля. То, что вы сейчас установили, показано на рис. 6.7: репозиторий GitHub, в котором хранятся конфигурации, и служба конфигурации, которая управляет доступом.

Давайте рассмотрим обе стороны связи, которую вы хотите обезопасить с помощью своих секретов. В приведенном ниже листинге служба Posts (и аналогичным образом служба Connections) теперь будет проверять, соответствует ли переданный секрет тому, что было настроено, а служба Connections’ Posts передаст секрет, который был настроен в ней.

Листинг 6.5 ❖ PostsController.java

```
public class PostsController {
    ...

    @Value(“${com.corneliadavis.cloudnative.posts.secret}”)
    private String configuredSecret;
    ...

    @RequestMapping(method = RequestMethod.GET, value=”/posts”)
    public Iterable<Post> getPostsByUserId(
        @RequestParam(value=”userIds”, required=false) String userIds,
        @RequestParam(value=”secret”, required=true) String secret,
        HttpServletResponse response) {
        Iterable<Post> posts;

        if (secret.equals(configuredSecret)) {
```

```
            logger.info(utils.ipTag() +
                  "Accessing posts using secret " + secret);

            // Ищем посты в базе данных и возвращаем их;
            ...
        } else {
            logger.info(utils.ipTag() +
                "Attempt to access Post service with secret " + secret
                + " (expecting " + password + ")");
            response.setStatus(401);
            return null;
        }
    }
    ...
}
```

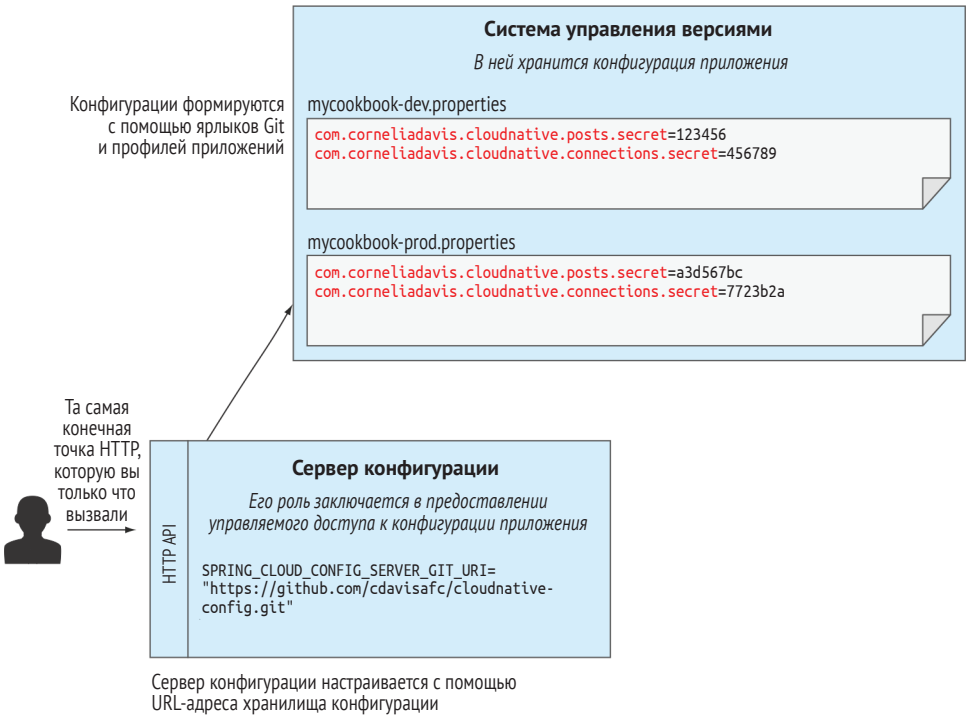


Рис. 6.7 ❖ Настройка приложения облегчается за счет использования системы управления версиями, которая сохранит значения конфигурации, и службы конфигурации, которая обеспечивает управляемый доступ к этим данным

В службе Connections' Posts секрет, который был сконфигурирован, будет передан в запросе к службам Connections или Posts, как показано в приведенном ниже листинге.

Листинг 6.6 ❖ ConnectionsPostsController.java

```
public class ConnectionsPostsController {
    ...

    @Value("${connectionpostscontroller.connectionsUrl}")
    private String connectionsUrl;
    @Value("${connectionpostscontroller.postsUrl}")
    private String postsUrl;
    @Value("${connectionpostscontroller.usersUrl}")
    private String usersUrl;
    @Value("${com.corneliadavis.cloudnative.posts.secret}")
    private String postsSecret;
    @Value("${com.corneliadavis.cloudnative.connections.secret}")
    private String connectionsSecret;

    @RequestMapping(method = RequestMethod.GET, value="/connectionsposts")
    public Iterable<PostSummary> getByUsername(
        @CookieValue(value = "userToken", required=false) String token,
        HttpServletResponse response) {

        if (token == null) {
            logger.info(utils.ipTag() + ...);
            response.setStatus(401);
        } else {
            ValueOperations<String, String> ops =
                this.template.opsForValue();
            String username = ops.get(token);
            if (username == null) {
                logger.info(utils.ipTag() + ...);
                response.setStatus(401);
            } else {
                ArrayList<PostSummary> postSummaries
                    = new ArrayList<PostSummary>();
                logger.info(utils.ipTag() + ...);

                String ids = "";
                RestTemplate restTemplate = new RestTemplate();

                // Получаем соединения;
                String secretQueryParam = "?secret=" + connectionsSecret;
                ResponseEntity<ConnectionResult[]> respConns
                    = restTemplate.getForEntity(
                        connectionsUrl + username + secretQueryParam,
                        ConnectionResult[].class);
                ConnectionResult[] connections = respConns.getBody();
                for (int i = 0; i < connections.length; i++) {
                    if (i > 0) ids += ",";
                    ids += connections[i].getFollowed().toString();
                }
                logger.info(utils.ipTag() + ...);

                secretQueryParam = "&secret=" + postsSecret;
                // Получаем посты для этих соединений;
                ResponseEntity<PostResult[]> respPosts
                    = restTemplate.getForEntity(
                        postsUrl + ids + secretQueryParam,
                        PostResult[].class);
```

```
                PostResult[] posts = respPosts.getBody();

                for (int i = 0; i < posts.length; i++)
                    postSummaries.add(
                        new PostSummary(
                            getUsersname(posts[i].getUserId()),
                            posts[i].getTitle(),
                            posts[i].getDate()));

                return postSummaries;
            }
        }
        return null;
    }
    ...
}
```

Помимо некоторых вещей, которые вы никогда не сделаете в реальной реализации, и я вернусь к ним через мгновение, ничто из этого не вызывает у вас никакого удивления. Но посмотрите на то, как значение конфигурации вводится в приложение. Файл свойств службы Connections' Posts выглядит так.

Листинг 6.7 ❖ Файл application.properties

```
management.endpoints.web.exposure.include=*
connectionpostscontroller.connectionsUrl=http://localhost:8082/connections/
connectionpostscontroller.postsUrl=http://localhost:8081/posts?userIds=
connectionpostscontroller.usersUrl=http://localhost:8082/users/
ipaddress=${INSTANCE_IP:127.0.0.1}
redis.hostname=localhost
redis.port=6379
com.corneliadavis.cloudnative.posts.secret=drawFromConfigServer
com.corneliadavis.cloudnative.connections.secret=drawFromConfigServer
```

Как я уже говорила, определенные здесь свойства могут просто выступать в качестве заполнителей. Оба секрета имеют значение `drawFromConfigServer`. (Это не инструкция, оно скорее произвольное. Здесь также можно было бы установить значение `foobar`.) И тогда у контроллера Connections' Posts появляются строки, которые выглядят так:

```
@Value(«${com.corneliadavis.cloudnative.posts.secret}»)
private String postsSecret;
@Value("${com.corneliadavis.cloudnative.connections.secret}")
private String connectionsSecret;
```

Выглядит знакомо, потому что это тот же метод, который использовался для внедрения значения системной конфигурации `INSTANCE_IP`. В этом и состоит суть. Уровень конфигурации приложения принимает точно такую же форму независимо от того, являются ли эти значения значениями системы/среды или вводятся значения конфигурации приложения.

На рис. 6.8 показано, как данные конфигурации приложения попадают в работающее приложение. Обратите внимание, что уровень конфигурации приложения с файлом свойств в центре остается таким же простым, как показано на рис. 6.4. Единственное, что изменилось, – это то, что сервер конфигурации предоставляет привязки к переменным, определенным в файле свойств.

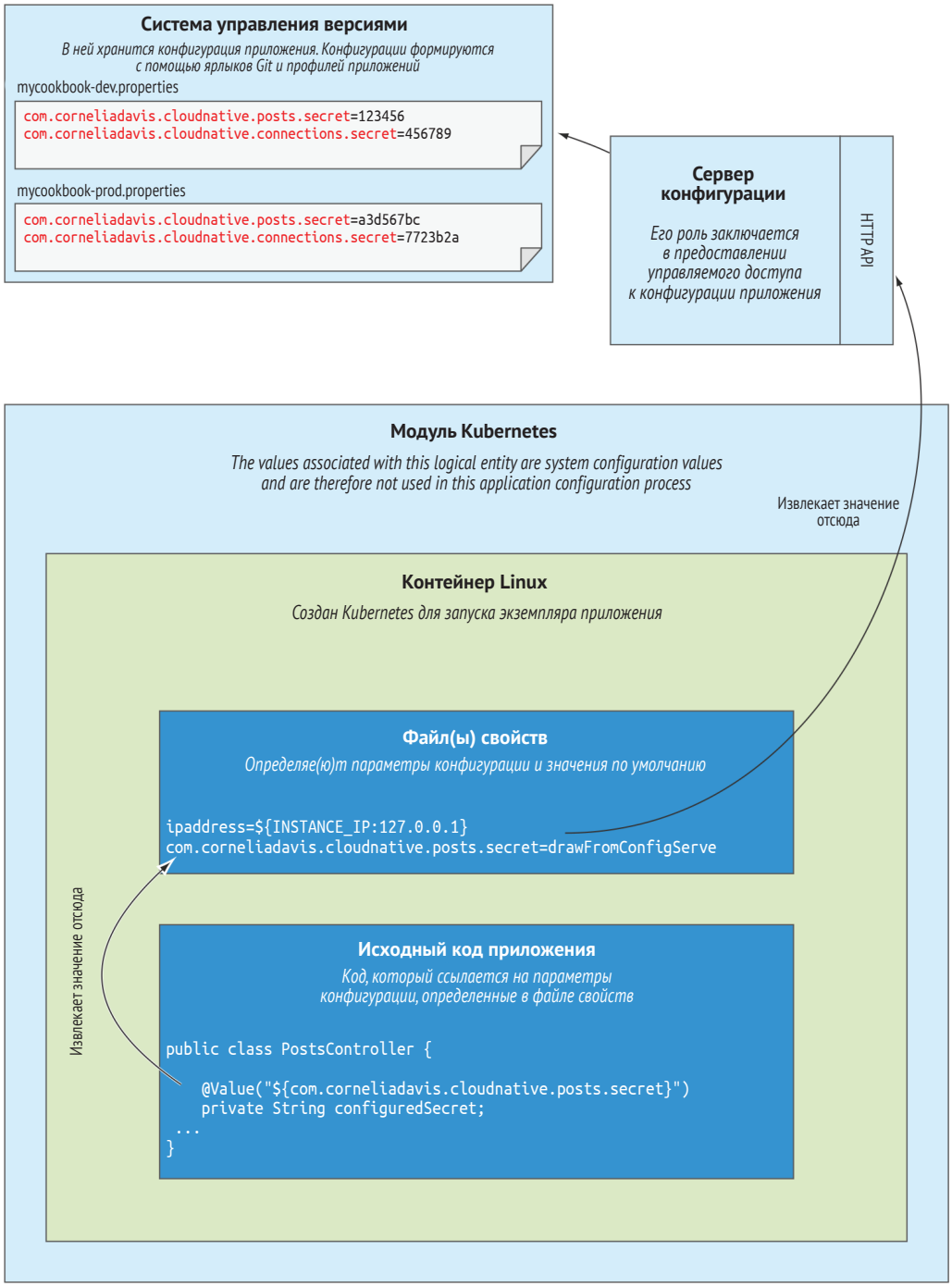


Рис. 6.8 ❖ Уровень конфигурации приложения зависит от файла свойств как средства ввода значений; эти значения поступают в ходе использования сервера конфигурации

Теперь давайте соединим конфигурацию приложения и системы воедино в одной диаграмме; см. рис. 6.9. Это именно то, что было реализовано в нашем примере. Еще раз обратите внимание, что шаблоны, используемые на уровне конфигурации приложения, одинаковы для обоих типов данных конфигурации. Отличие состоит в том, как значения вводятся в этот уровень. Что касается системных данных, они обрабатываются платформой (в данном случае Kubernetes) и прекрасно обслуживаются за счет использования переменных среды. Для данных конфигу-

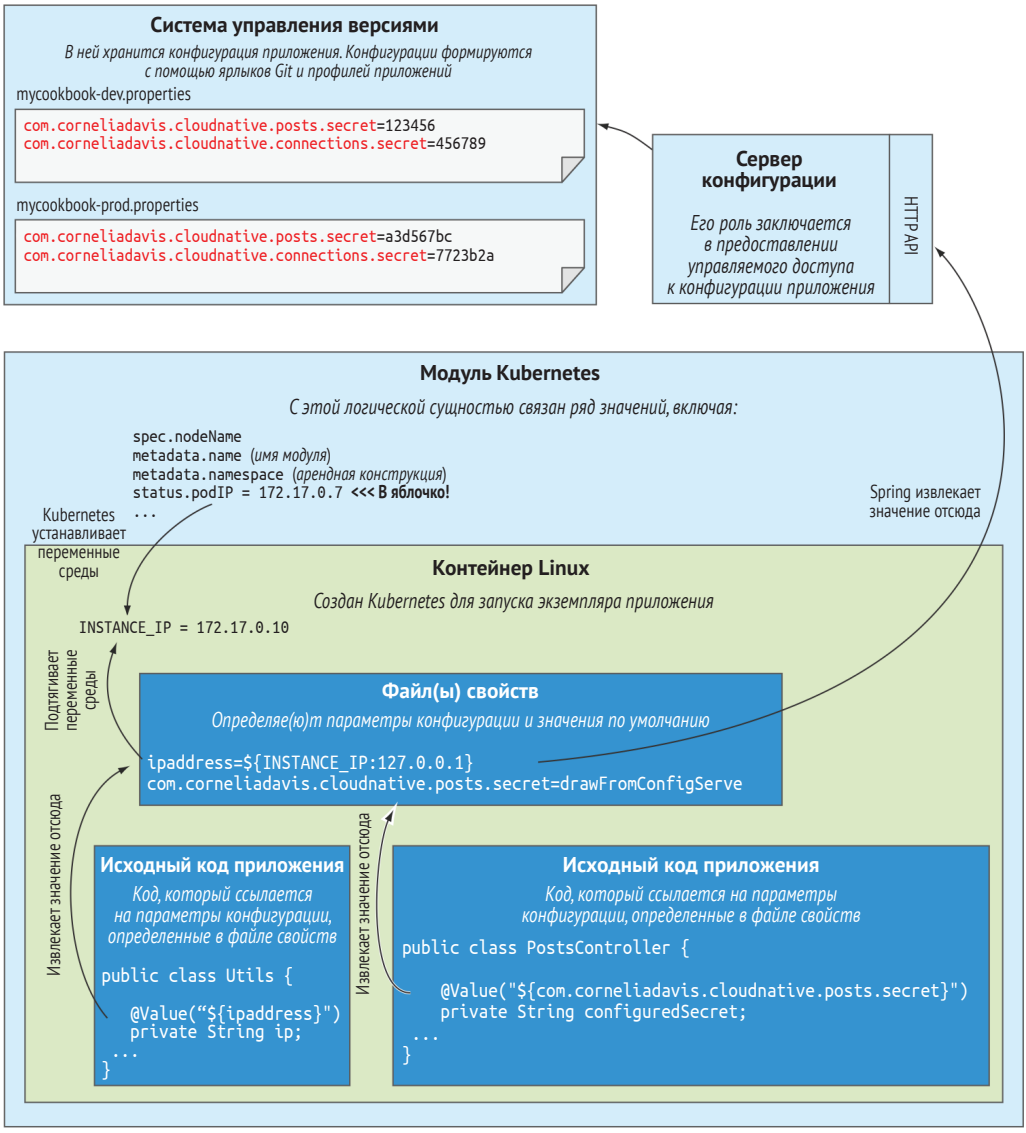


Рис. 6.9 ❖ Файл свойств действует как общий уровень конфигурации для конфигурации системы и конфигурации приложения. Данные конфигурации системы вводятся через переменные среды, а данные конфигурации приложения вводятся через службу конфигурации

рации приложения мы используем Spring (есть и другие варианты – смотрите мои будущие записи в блоге на эту тему), который выполняет вызов HTTP-интерфейса для сервера конфигурации Spring Cloud; этот подход позволяет создавать версии данных конфигурации приложения.

6.4.2. Безопасность добавляет больше требований

Я создала эту реализацию таким образом, чтобы можно было легко вести рассуждения об основных шаблонах проектирования для настройки приложений для облачной среды. Но есть несколько вещей, которые вы бы никогда не сделали, как я здесь:

- вы ни за что не станете передавать секреты в строке запроса; вместо этого они будут переданы в HTTP-заголовок или в теле;
- вы определенно не станете выводить значения секретов в файлах журнала;
- в хранилище конфигурации вы как минимум зашифруете любые конфиденциальные значения. SCCS поддерживает шифрование, а такие технологии, как Vault от компании HashiCorp, предоставляют дополнительные услуги для управления учетными данными;
- вы заметите, что каждый метод в контроллерах служб Posts и Connections теперь имеет фактически один и тот же код, охватывающий функциональность этого метода. Такой шаблонный код отвлекает от основной функциональности метода и повторяется слишком часто. Большинство современных платформ программирования обеспечивает относящиеся к безопасности абстракции, которые дают возможность для более изящной настройки этой функциональности.

6.4.3. Давайте посмотрим, как это работает: конфигурация приложения с использованием сервера конфигурации

Итак, ваша реализация будет прекрасно работать, если секреты, настроенные во всех приложениях, совпадают. Это напрямую связано с одной из ключевых задач при настройке приложений для облачной среды: они сильно распространенные! Обратите внимание, что свойства `mycookbook` определены не для одного приложения; одна и та же конфигурация используется для разных микросервисов. У меня есть единственное место, где я настраиваю секрет, и мои методы эксплуатации будут правильно соединять их.

Теперь, когда все готово, давайте удостоверимся, что то, что мы разработали, работает так, как и было заявлено.

Настройка

Если вы уже выполнили инструкции по настройке из примера в разделе 6.3.1, здесь больше не нужно ничего делать. Как всегда, вы можете создавать исполняемые файлы из исходного кода, собирать образы Docker и помещать их в свой репозиторий в Docker Hub. Но я уже выполнила это, сделала их доступными для вас в Docker Hub. Все файлы конфигурации указывают на соответствующие образы Docker.

Запуск приложений

Прежде всего очистите набор микросервисов, которые вы развернули в Kubernetes. Напомню, что я предоставила скрипт, который позволяет вам сделать это за один раз, если вы наберете эту команду:

```
./deleteDeploymentComplete.sh
```

Перед повторным развертыванием служб вам нужно подключить процесс развертывания к серверу конфигурации. Вы уже развернули сервер конфигурации (если вы еще не сделали этого, как было описано ранее, сделайте это сейчас). Теперь необходимо внедрить координаты этого сервера конфигурации в реализацию, чтобы Spring Framework мог использовать это соединение для поиска и ввода значений конфигурации. В манифестах развертывания Kubernetes для каждой из служб вы найдете определение переменной среды, у которой есть URL-адрес для SCCS. Вам необходимо указать конкретный URL-адрес во всех трех местах – манифестах развертывания для служб Posts, Connections и Connections' Posts. Вы можете получить правильное значение с помощью этой команды:

```
minikube service --url sccs-svc
```

Предполагая, что вы оставили службы Redis и MySQL в рабочем состоянии, эти URL-адреса не нужно обновлять. Вы можете развернуть службы Posts и Connections с помощью этих двух команд:

```
kubectl create -f cookbook-deployment-connections.yaml
kubectl create -f cookbook-deployment-posts.yaml
```

Опять же, теперь вам нужно обновить манифест развертывания для службы Connections' Posts, чтобы указать путь к URL-адресам свойств служб Posts и Connections. Напомню, что вы можете получить эти значения следующим образом:

URL-адрес службы Posts	`minikube service posts-svc --format «http://{{.IP}}:{{.Port}}/posts?userIds=» --url`
URL-адрес службы Connections	`minikube service connections-svc --format «http://{{.IP}}:{{.Port}}/connections/» --url`
URL-адрес службы Users	`minikube service connections-svc --format «http://{{.IP}}:{{.Port}}/users/» --url`

Теперь разверните службу Connections' Posts:

```
kubectl create -f cookbook-deployment-connectionsposts.yaml
```

Вызовите службу Connections' Posts, как вы это делали ранее, сначала выполнив аутентификацию, а затем получив посты:

```
# authenticate
curl -X POST -i -c cookie \
   $(minikube service --url connectionsposts-svc)/login?username=cdavisafc
# get the posts
curl -i -b cookie \
   $(minikube service --url connectionsposts-svc)/connectionsposts
```

Ничего не изменилось, верно? Этого и следовало ожидать, но давайте быстро посмотрим за кулисы, заглянув в файлы журналов службы Posts:

```
... : [172.17.0.4:8080] Accessing posts using secret 123456
... : [172.17.0.4:8080] getting posts for userId 2
... : [172.17.0.4:8080] getting posts for userId 3
```

Видно, что к этой службе был получен доступ с использованием секрета `123456`. Очевидно, что секреты были правильно сконфигурированы как для вызывающей службы (Connections' Posts), так и для вызываемой (Posts).

Итак, что происходит, если вам нужно обновить конфигурацию приложения? Вы хотите, чтобы новые значения вводились одинаково во всех экземплярах приложения для одной службы, и конечно же, когда значения должны быть вставлены в разные службы, этот процесс также должен быть скоординирован. Давайте попробуем. Первое, что я попрошу вас сделать, – это обновить значения секретов для профиля dev в `mycookbook`. Можно заменить эти значения по своему усмотрению. Затем вы должны зафиксировать эти изменения в своем репозитории и загрузить на GitHub. Из каталога cloud-native-config:

```
git add .
git commit -m “Update dev secrets.”
git push
```

Если вы теперь выполните финальную команду `curl`, которая обращается к данным службы Connections'Posts, все будет работать как положено. Но если вы снова посмотрите файл журнала службы Posts, то увидите, что при доступе по-прежнему используется, и успешно, секрет 123456. И здесь мы переходим к теме следующей главы – жизненному циклу приложения. Следует быть осторожными, когда применяются изменения конфигурации. Из приведенного выше примера видно, что новые учетные данные еще не были применены.

Но это будет сделано при запуске, поэтому я дам вам перезапустить службу Posts, удалив модуль. Поскольку вы инстанциировали развертывание в Kubernetes, где указано, что всегда должен работать один экземпляр службы Posts, Kubernetes немедленно создаст новый модуль для нового экземпляра службы Posts:

```
kubectl delete pod/posts-66bcfcbbf7-jvcqb
```

Теперь снова выполните команду `curl`. Вы увидите две вещи:

- во-первых, вызов службы окончится неудачно;
- и во-вторых, просмотр файла журнала службы Posts показывает, почему это происходит:

```
... : [172.17.0.7:8080] Attempt to access Post service with secret
     123456 (expecting abcdef)
```

Когда вы удалили и заново создали службу Posts, она подняла новые значения конфигурации. Однако в службе Connections' Posts по-прежнему находятся старые значения.

Старые отправлены, а новые ожидаются. Очевидно, что вам необходимо координировать обновление конфигурации между экземплярами, а иногда и между службами, но прежде чем вы сможете сделать это, вы должны изучить проблемы и шаблоны жизненного цикла приложения. Это тема следующей главы.

Резюме

- Архитектура программного обеспечения для облачной среды требует переоценки методов, которые вы используете для конфигурации приложения. Некоторые существующие практики остаются, но некоторые новые подходы также полезны.
- Конфигурация приложения для облачной среды не так проста, как просто сохранение конфигурации в переменных среды.

- Файлы свойств остаются важной частью правильной обработки конфигурации программного обеспечения.
- Использование переменных среды для конфигурации идеально подходит для данных конфигурации системы.
- Вы можете использовать платформы для облачной среды, такие как Kubernetes, для доставки значений среды в свои приложения.
- Конфигурацию приложения следует рассматривать так же, как исходный код: управляется в репозитории исходного кода, имеет несколько версий, а доступ контролируется.
- Серверы конфигурации, такие как Spring Cloud Configuration Server, используются для доставки значений конфигурации в ваши приложения.
- Теперь вам нужно подумать о том, когда применяется конфигурация, которая по своей сути связана с жизненным циклом приложений для облачной среды.

Глава 7

Жизненный цикл приложения: учет постоянных изменений

О чем идет речь в этой главе:

- обновления с нулевым временем простоя: сине-зеленое развертывание и последовательное обновление;
- канареечное развертывание;
- шаблоны периодической смены реквизитов доступа;
- жизненный цикл приложения и устранение неполадок;
- проверка работоспособности приложения.

Жизненный цикл приложения кажется довольно простым: приложение развертывается, запускается, работает некоторое время и в конечном итоге закрывается. Помимо хаоса, который возникает, когда выключение происходит неожиданно, этот жизненный цикл обычно скучен (или мы надеемся, что это так). Почему же тогда этой теме посвящена целая глава?

Прежде чем я отвечу на этот вопрос, позвольте мне сначала прояснить мое определение *жизненного цикла приложения*. Жизненный цикл приложения, который я здесь рассматриваю, заметно отличается от жизненного цикла разработки программного обеспечения (SDLC), о котором я уже не раз говорила. Он касается фаз, которые проходит ваше программное обеспечение в процессе *разработки* и *доставки* программного обеспечения – от проектирования и разработки до прохождения модульных тестов, а затем от интеграционных тестов до доставки в производство.

С другой стороны, жизненный цикл приложения – это этапы, которые проходит приложение после того, как оно готово к развертыванию в рабочем окружении. Главной проблемой является не работа, направленная на разработку или управление программным обеспечением, а состояние самого приложения. Оно развернуто? Работает или остановлено (из-за сбоя или это сделано преднамеренно)? Хотя это и естественно рассматривать как приложение, развертываемое как часть SDLC, основное внимание здесь уделяется рабочему состоянию приложения.

Чтобы помочь вам понять содержание этой главы, а также дать основу для последующих объяснений, на рис. 7.1 изображена довольно стандартная последовательность этапов, через которые проходит приложение. Вы заметите, что я исключила создание развертываемого артефакта из жизненного цикла приложения;

это часть SDLC. С другой стороны, вы, возможно, найдете любопытным тот факт, что в жизненный цикл приложения я включила наличие подготовленной, а затем и ликвидированной среды. Я попрошу вас потерпеть. Обещаю, все скоро станет ясно.

Рис. 7.1 ❖ Простое описание этапов жизненного цикла приложения

Итак, приложения запускаются и останавливаются. Чем это интересно при работе в нашей облачной среде? Как оказалось, это два фактора, которые характеризуют приложения для облачной среды: они сильно распределены и постоянно меняются.

Давайте начнем с первого. Вы уже знаете, что даже при развертывании нескольких экземпляров приложения в совокупности они должны вести себя как единый логический объект. Как тогда правильно вести себя, когда речь идет о чем-то вроде применения новой конфигурации к приложению или развертывания новой версии? Вы делаете это для всех приложений в режиме ожидания или есть другой подход? (Посмотрите на рис. 6.1, на котором показано, что «в конце концов» все экземпляры должны иметь одинаковую конфигурацию; это предвосхищает тему, которую я буду рассматривать в данной главе.)

Что касается постоянных изменений, помните, что приложения будут регулярно перемещаться из-за сбоя или события в управлении, такого как устранение уязвимости в операционной системе. Помимо того что это приводит к большому числу запусков и остановок, также существуют каскадные эффекты, которые мы должны учитывать. Например, когда другой микросервис зависит от только что запущенного приложения, приложению может потребоваться сделать доступной информацию о запущенном приложении для зависимых приложений.

Жизненные циклы приложений для облачной среды отличаются от жизненных циклов приложений, работавших в прошлые десятилетия, что предъявляет новые требования к дизайну приложений. Вот почему я сфокусировала всю эту главу на жизненном цикле приложения.

После краткого обзора некоторых эксплуатационных проблем, которые пересекаются с жизненным циклом приложения, я расскажу о жизненном цикле нескольких экземпляров приложения (а этого у вас всегда будет хватать). Как уже совершенно очевидно, ваше программное обеспечение состоит из множества приложений, работающих вместе, поэтому я расскажу о необходимости знать, как события жизненного цикла одного приложения влияют на другое. Затем я расскажу об эфемерном контексте, в котором сейчас находятся наши приложения, и о том, что это значит для дизайна ваших приложений. В среде, которая постоянно меняется, становится первостепенной возможность точной оценки работоспособности приложения и, если необходимо, принятие мер. Вы увидите раздел, посвященный этому. И наконец, я кратко расскажу о парадигме внесерверной об-

работки данных, или, скорее, парадигме программирования FaaS (функция как услуга) с точки зрения жизненного цикла приложения.

7.1. Сочувствие к операциям

Жизненный цикл приложения больше связан с эксплуатацией, чем с разработкой, но, будучи разработчиками, вам необходимо предоставлять программное обеспечение, которым можно эффективно управлять в рабочем окружении. Сочувствие к оператору приложения – тема, которая проходит через этот раздел. Черт возьми, в наше время вы, скорее всего, будете заниматься как эксплуатацией, так и разработкой, поэтому внимание, уделяемое здесь этой теме, своего рода эгоистично. Давайте кратко рассмотрим некоторые из основных проблем эксплуатации, взглянув на влияние жизненного цикла в облачной среде:

- *управляемость* – одной из первых задач по эксплуатации является устойчивость управления развертыванием приложений. Везде, где это возможно, функции управления должны быть автоматизированы, а когда необходимы какие-либо задачи, связанные с управлением, они должны выполняться эффективно и надежно. То, как вы проектируете свое программное обеспечение, может оказать заметное влияние. Например, как вы увидите, изменение в конфигурации приложения почти всегда требует перезапуска приложения, поэтому вы должны не торопясь решить, следует ли осуществлять какую-либо конфигурацию или вводить данные;
- *устойчивость* – к концу этой книги вы сможете оценить платформы, поддерживающие работу приложений от вашего имени, даже несмотря на то, что вокруг них происходит постоянный поток изменений. Иногда мне нравится рассматривать эти платформы как роботов – роботов, выполняющих целый ряд задач по эксплуатации, которые раньше выполняли люди. Но вот в чем дело: роботы не читают заметки о выпуске. Я делаю это заявление, подмигивая вам и определенно метафорически, но это намерение справедливо и для жизненного цикла приложения, как и для многих других аспектов вашего программного обеспечения для облачной среды. Например, если такая система, как Kubernetes, будет поддерживать новый экземпляр приложения в случае сбоя, у нее должен быть несубъективный способ обнаружить, что произошел сбой приложения или он происходит в данный момент. Вы как разработчик приложения должны убедиться в том, что у платформы есть отказоустойчивый способ обнаружить сбой приложения;
- *отзывчивость* – пользователи вашего программного обеспечения должны своевременно получать результаты, а то, что считать своевременным, зависит от варианта использования. Например, если пользователь загружает колоду PowerPoint в SlideShare, ничего такого, если эта колода не будет доступна другим пользователям в течение нескольких минут, предоставляя время для преобразования любого формата. С другой стороны, если пользователь использует сайт – агрегатор новостей в первый раз и видит, что на рендеринг страниц уходит десятки секунд, скорее всего, это будет его последний визит. На отзывчивость вашего приложения будет влиять множество факторов, а способ, которым жизненный цикл приложения связан с действиями пользователя, один. Например, если приложение запускается только после

того, как был сделан пользовательский запрос, пользователь почувствует цену такого запуска. В первом примере он, вероятно, этого не заметит. В последнем примере эта цена может означать разницу между оставшимся или потерянным клиентом;

- *управление расходами* – одним из главных обещаний облака является обещание эффективности затрат. Вместо обеспечения, настройки и управления инфраструктурой для пиковых нагрузок или, неизбежно, переоценки пиковых нагрузок вы можете использовать только те ресурсы, которые вам нужны в данный момент. Возможность масштабировать емкость приложений или зависеть от нагрузки трафика, и даже оптимизировать время простоя, является мощным рычагом в управлении IT-расходами. Эти операции масштабирования означают, что новые приложения запускаются, а старые останавливаются. Изящная обработка этих событий жизненного цикла приложения крайне важна.

В оставшейся части этой главы я расскажу о многих шаблонах, которые прекрасно соответствуют предыдущему списку проблем.

7.2. Жизненный цикл одного приложения и жизненные циклы нескольких приложений

Давайте начнем с конкретного примера: нашей службы Posts. В главе 6 вы добавили некую конфигурацию в эту службу, секрет, используемый для авторизации. Предположим, у вас есть более одного экземпляра приложения, каждый из которых работает с одной и той же конфигурацией (одним и тем же секретом). Все работает просто отлично, приложение функционирует точно так, как вы и хотите, а затем – ой! – вы случайно загружаете файл mycookbook.properties, содержащий секрет, в открытый репозиторий на GitHub.

Несмотря на то что вы быстро осознали свою ошибку и извлекли файл, ваш секрет мог быть скомпрометирован, поэтому вам нужно переключиться между наборами этих учетных данных. Для этого вы измените исходный файл конфигурации и через сервер конфигурации доставите эту конфигурацию всем работающим приложениям. Вот где нужно подумать о жизненном цикле приложения.

> **ВНИМАНИЕ!** Да, я понимаю, что в примере с работающим кодом я убедила вас сделать именно то, что сейчас охарактеризовала как проблему: сохранить учетные данные в открытом репозитории на GitHub. Я обращу ваше внимание на комментарии в главе 6, которые повторяю здесь: секреты должны храниться в репозитории, специально предназначенном для обработки конфиденциальной информации, – нечто вроде хранилища Vault от компании HashiCorp или CredHub от компании Predotal. Я использую в наших примерах GitHub только для простоты.

Я говорю «доставить эту конфигурацию работающим приложениям», но это не совсем честное утверждение. Я имею в виду, что хочу, чтобы вы прокрутили приложение – перезапустили его с применением новой конфигурации. Это именно то, что я велела вам сделать в конце главы 6. Я сказала вам удалить модуль Posts, заставив Kubernetes создать новый. Это урок номер один, когда речь заходит о конфигурации приложения и его жизненном цикле.

ПРИМЕЧАНИЕ Когда новая конфигурация применяется к вашему работающему приложению, приложение нужно создать заново, а следовательно, перезапустить с применением новой конфигурации.

Некоторые из вас могут чувствовать себя неуютно, читая эти инструкции. Перезапуск приложения может показаться расточительным занятием, если оно связано с расходами; требуется время, чтобы запустить приложение. А воссоздание среды выполнения идет еще дальше и обходится еще дороже. И если вы хорошо знаете Spring, то, возможно, уже подумали о легко активируемой конечной точке `/refresh`, которая при вызове обновит контекст приложения без полного перезапуска последнего. Я полагаю, что использование `/refresh` – это то, что может легко привести к неуправляемому развертыванию приложения.

Чтобы объяснить, почему, давайте рассмотрим приведенный далее сценарий. Допустим, у вас развернуто два экземпляра службы Posts. Как я уже говорила, один из приемов архитектур приложений для облачной среды, когда вы запускаете более одного экземпляра приложения, состоит в том, чтобы эти несколько экземпляров обслуживали результаты как одно логическое приложение. Когда к службе Posts делается запрос, это приводит к одному и тому же результату независимо от того, какой экземпляр использовался. На рис. 7.2 показаны два экземпляра службы Posts, запущенных балансировщиком нагрузки.

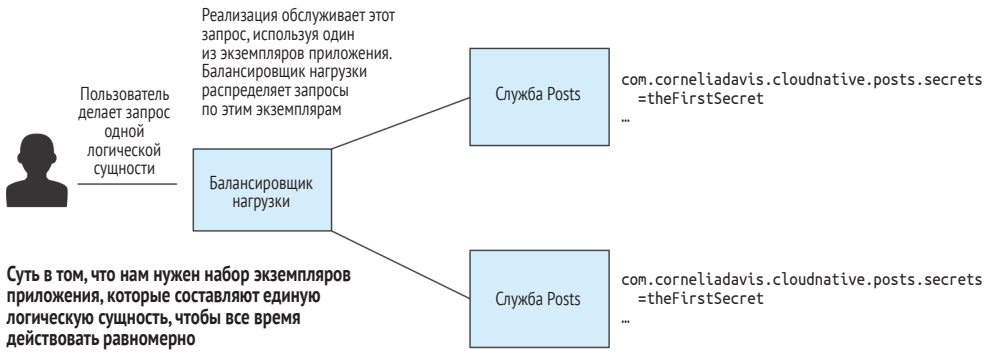


Рис. 7.2 ❖ Оба экземпляра службы Posts работают с одинаковой конфигурацией и действуют как единое логическое приложение

Теперь давайте рассмотрим, что произойдет, если вы используете команду `curl` для конечной точки `/refresh`. Эта команда будет работать только с одним экземпляром, эффективно обновляя секрет только для этого экземпляра. Теперь, как видно на рис. 7.3, один экземпляр работает с `theFirstSecret`, а другой – с `SecondSecret`. Ясно видно, что при выполнении запроса, подобного приведенному ниже, если балансировщик нагрузки направляет запрос к первому экземпляру, запрос будет выполнен успешно, но в случае со вторым экземпляром он потерпит неудачу. Наши два экземпляра определенно не действуют как единое логическое приложение:

```
$ curl http://myapp.example.com /posts?secret=theFirstSecret
```

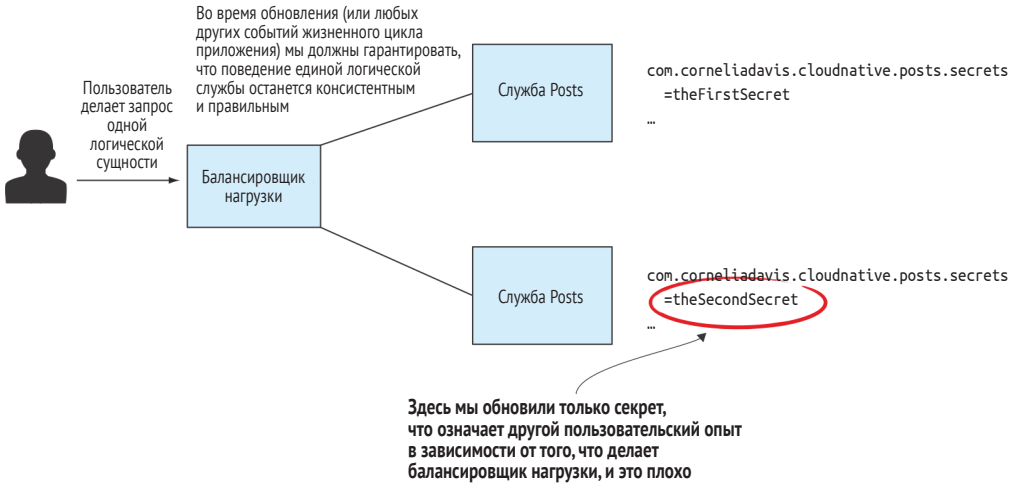


Рис. 7.3 ❖ Поскольку для двух экземпляров службы Posts применяются разные конфигурации, результат выполнения запроса будет сильно отличаться в зависимости от того, направлен запрос в первый экземпляр приложения или во второй. Этого следует избегать даже во время обновления с нулевым временем простоя

Итак, вы, наверное, думаете, что ответ прост: просто пролистайте все экземпляры и обновите конфигурацию. В конечном счете вы были бы правы, но прокручивать экземпляры таким образом, чтобы все продолжало работать, сложновато. Вы должны рассмотреть факторы, включая способ циклического обхода всех экземпляров приложения и состояние системы в целом в процессе обновления.

Чтобы обратиться ко всем экземплярам, у вас может возникнуть желание просто выполнить еще одну команду `/refresh curl`, но нет гарантии, что другой экземпляр приложения будет доступен для этой команды. Вот почему использование URL-адреса `/refresh` – это ювелирная работа. Вы пытаетесь использовать интерфейс пользователя для выполнения функции управления. Обновление конфигурации всех экземпляров приложения – это полностью функция управления, и вам нужно использовать инструмент, который будет работать на каждом экземпляре, и вы должны контролировать это. Нельзя находиться во власти балансировщика нагрузки. Функция управления обновлением конфигурации всех экземпляров должна располагаться за балансировщиком нагрузки, а не перед ним.

Я покажу вам один из этих инструментов управления (внимание! спойлер: он находится в Kubernetes), когда мы будем рассматривать конкретный пример в разделе 7.3, а сейчас давайте продолжим обсуждение, предполагая, что такая функция плоскости управления существует. Хотя механизм использования URL-адреса `/refresh` неверен, цель состоит в том, чтобы обновить конфигурацию приложения, и вам нужно применить новую конфигурацию с нулевым временем простоя. Рассмотрим три варианта:

- изменить конфигурацию во время работы приложения;
- установить второй набор экземпляров, ко всем из которых будет применена новая конфигурация, а затем переключить весь трафик с первого набора на второй одним махом. Это *сине-зеленое развертывание*;

- прокрутить экземпляры приложений, заменив их подмножество новыми, а затем перейти к следующему подмножеству. Это *последовательное обновление*.

Я бы хотела сразу же исключить первый вариант по нескольким причинам. Во-первых, многие приложения и платформы приложений применяют изменения конфигурации только во время запуска приложения. Например, с учетом того, как мы использовали файлы .property в наших примерах, это хорошая практика, так что изменения конфигурации не будут применены, пока контекст приложения не будет обновлен, и это довольно близко к перезапуску приложения (и именно то, что делает конечная точка `/refresh`).

Кроме того, применение изменений конфигурации без перезапуска может привести ваше приложение в состояние, которое невозможно воспроизвести. Допустим, приложение загружает справочные данные при запуске, а местоположение, из которого эти данные загружаются, указано в параметре конфигурации. Если вы измените этот параметр и инициируете загрузку данных из нового местоположения, справочные данные, загруженные в память, будут представлять собой комбинацию первой загрузки и второй, и функциональные возможности приложения будут отражать это состояние. Теперь предположим, что приложение вышло из строя и вы хотите устранить неполадки. У вас нет возможности получить экземпляр, у которого то же состояние, что и у аварийного приложения. Как вы уже неоднократно видели, воспроизводимые развертывания приложений абсолютно необходимы в облачном контексте, в котором часто создаются экземпляры приложений.

ПРИМЕЧАНИЕ Применение конфигурации приложения во время запуска (рис. 7.4) значительно упрощает методы эксплуатации и поэтому настоятельно рекомендуется.

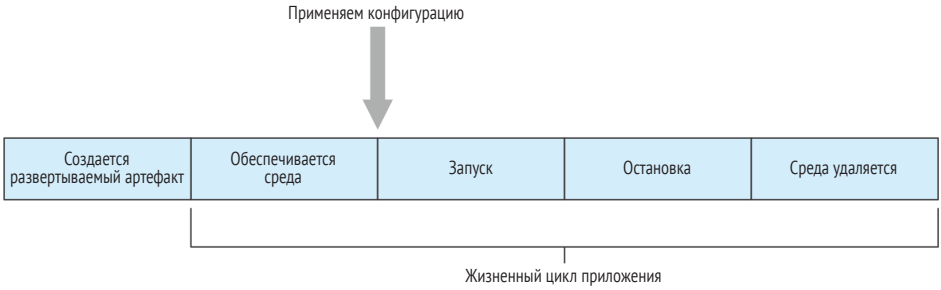


Рис. 7.4 ❖ Конфигурацию приложения лучше всего применять при запуске приложения. Большинство каркасов приложений естественно функционирует таким образом

Второй и третий варианты представляют ценность в контексте приложений для облачной среды. Давайте рассмотрим их более подробно.

7.2.1. Сине-зеленые обновления

Используемое для обновления конфигурации или версии вашего работающего приложения сине-зеленое развертывание, особенно с точки зрения разработчика, является самым простым подходом. Оно запускается, когда у вас работает одна версия приложения, «синяя» версия, и вы хотите развернуть новую версию, «зеленую».

На рис. 7.5, в котором предполагается, что развернуто несколько экземпляров вашего приложения для облачной среды, изображен этот процесс. Для начала у вас есть балансировщик нагрузки, обслуживающий трафик для всех синих экземпляров приложения. На следующем этапе вы развернете полный набор экземпляров новой, зеленой версии, а весь рабочий трафик пойдет на синюю версию. Теперь вы можете проверить правильность работы зеленых экземпляров, отправив им трафик, и после проверки вы перережете весь трафик с «синей» версии на «зеленую».

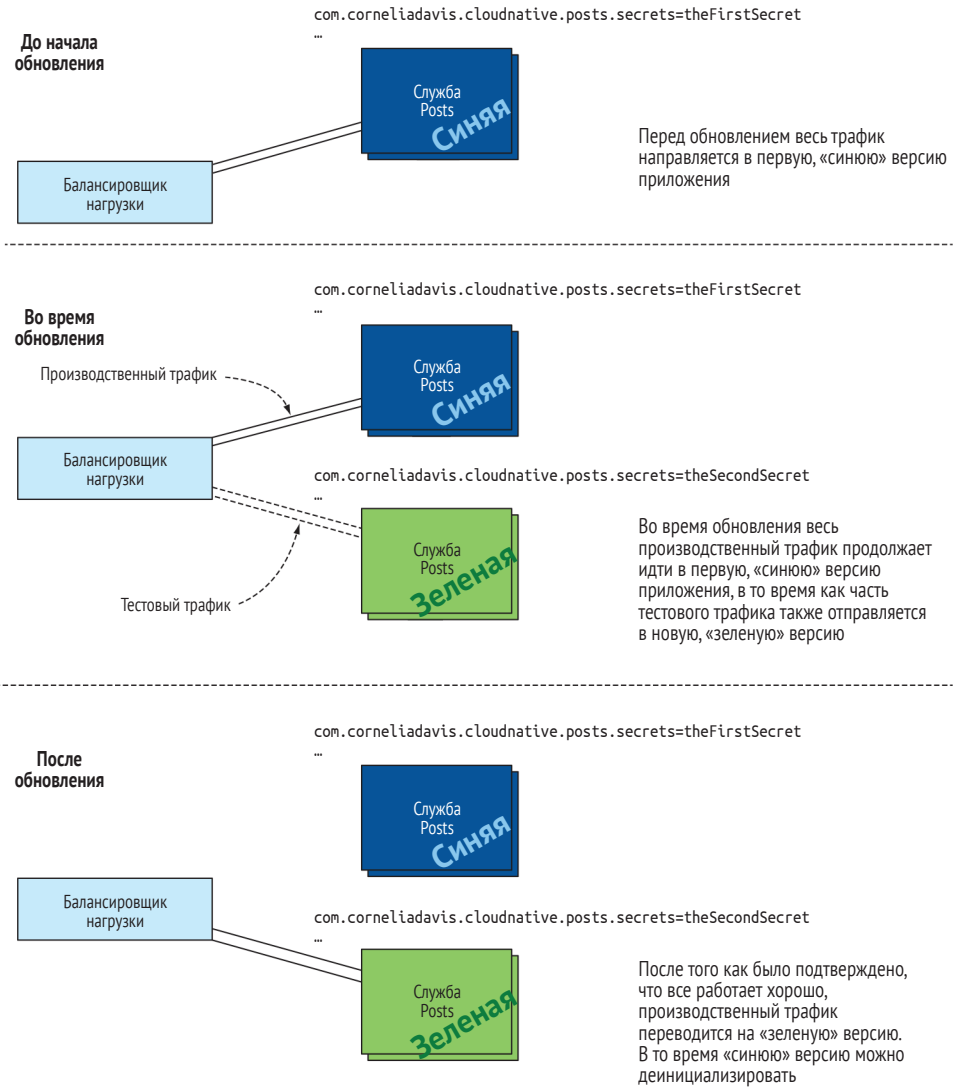


Рис. 7.5 ❖ Сине-зеленое развертывание используется, когда приложение не допускает одновременного запуска нескольких версий (когда нельзя сделать несколько версий действующими как единый логический экземпляр)

ПРИМЕЧАНИЕ Что делает сине-зеленое развертывание проще, чем последовательное обновление с точки зрения дизайна вашего приложения, так это то, что в первом случае одновременно запускается только одна версия приложения.

Предыдущее примечание действительно интересно. Идея того, что в рабочем окружении у вас одновременно есть только одна версия приложения, – это то, к чему вы привыкли, но когда вы устраняете данное предположение, это дает вам большую силу. Последовательное обновление – одно из таких вещей.

7.2.2. Последовательные обновления

Последовательное обновление также используется для обновления работающего приложения с нулевым временем простоя, и, как и в случае с сине-зеленым развертыванием, по завершении весь трафик к приложению будет направляться в новую версию приложения. Однако в процессе обновления все заметно отличается.

Этот процесс изображен на рис. 7.6. Вначале весь трафик выравнивается по нагрузке между несколькими экземплярами текущей версии приложения. Во время обновления вы постепенно переводите подмножество экземпляров текущей версии в автономный режим и выводите число экземпляров новой версии приложения онлайн, чтобы заменить их. При этом трафик распределяется по нагрузке между экземплярами исходной версии приложения и экземплярами новой версии приложения. Трафик подается в более чем одну версию приложения! После того как первая партия была успешно запущена, вы обновляете следующую партию и т. д. Последовательное обновление завершается, когда все старые версии приложения будут заменены новыми.

Я еще раз обращаю ваше внимание на рис. 6.1, который указывает, что в конечном итоге все экземпляры приложения будут работать с одинаковой конфигурацией. Теперь вы конкретно видите, что я имею в виду.

ПРИМЕЧАНИЕ Во время последовательного обновления трафик приложений будет обрабатываться различными версиями одного и того же приложения.

Содержание вышеупомянутого примечания действительно интересно. Вспомните одну из наших основных предпосылок: набор независимых экземпляров приложения должен работать сообща, как единое целое. Результат вызова приложения должен быть одинаковым независимо от того, какой экземпляр приложения отвечает на запрос вызова. Задумайтесь об этом на мгновение. Это означает, что вы как разработчик приложения несете ответственность за то, чтобы дизайн вашего приложения поддерживал этот шаблон развертывания. Теперь позвольте мне прояснить кое-что: это не всегда возможно, но если вы сможете реализовать эту особенность в своем дизайне, тогда и только тогда шаблон развертывания с последовательными обновлениями станет доступным.

7.2.3. Параллельное развертывание

Без сомнения, требуется больше усилий для создания программного обеспечения, которое позволяет проводить последовательные обновления (и вскоре я расскажу вам о конкретном примере на базе кода). Поэтому вам, наверное, интересно, что вам даст это дополнительное усилие, если в итоге сине-зеленое развертывание

и последовательное обновление дают одинаковый результат – все экземпляры переключаются на новую версию. Короткий ответ: «много чего», но позвольте мне обратить ваше внимание на два момента.

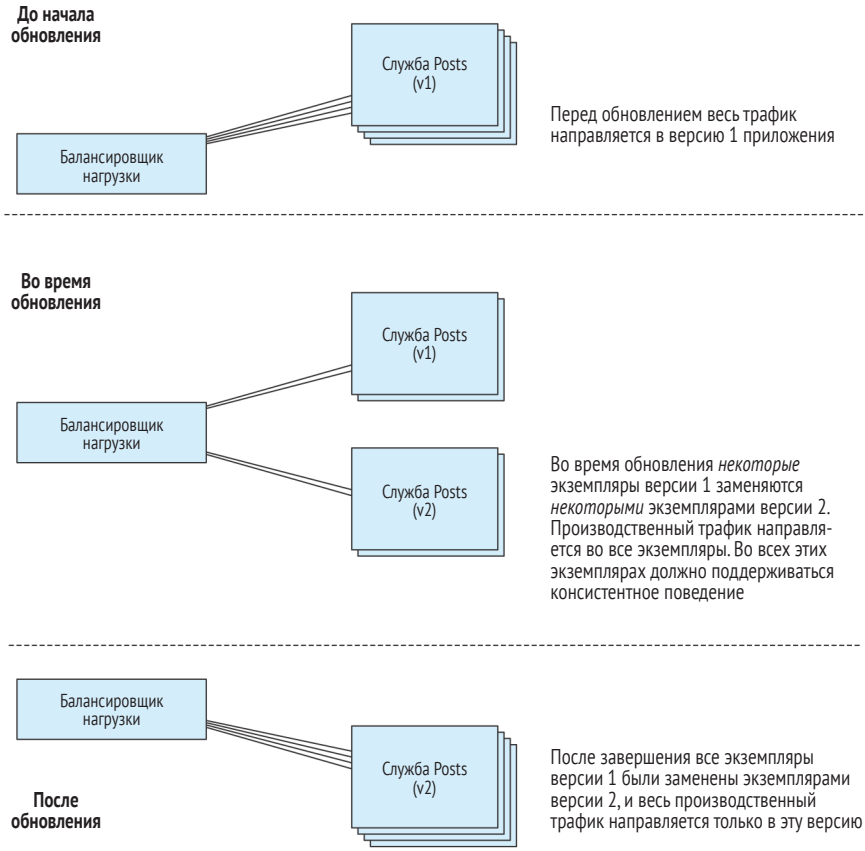


Рис. 7.6 ❖ Во время последовательных обновлений производственный трафик направляется более чем в одну версию приложения. Приложение должно функционировать как единое логическое целое, даже если запросы распределяются по разным версиям

Во-первых, сине-зеленое развертывание требует больше ресурсов, чем последовательное обновление. Во время обновления, чтобы поддерживать стабильную емкость приложения, вам потребуется столько же экземпляров зеленой версии, сколько и синей. Только после того, как вы щелкнете переключателем и весь трафик будет перенаправлен в новую версию, ресурсы, используемые старой версией, могут быть освобождены. При последовательном обновлении вы выбираете размер партии – количество экземпляров, которые будут заменены за цикл, – и можете контролировать ресурсы, необходимые для процесса обновления. Можно увидеть это на рис. 7.7.

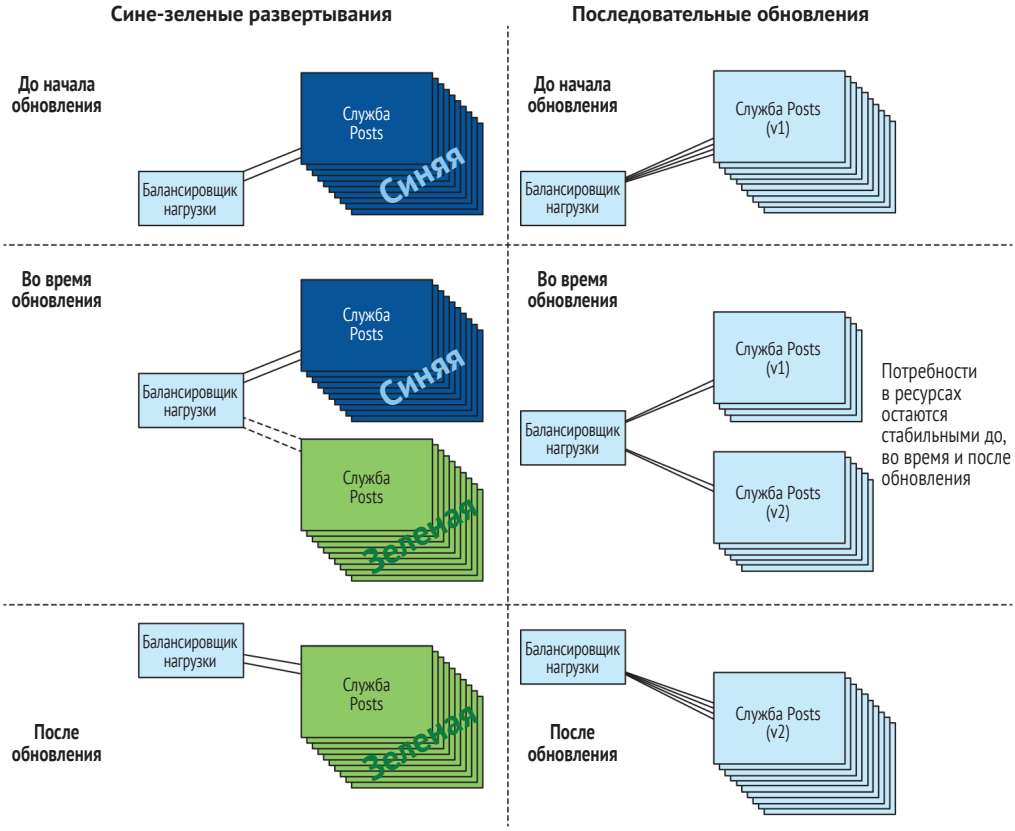


Рис. 7.7 ❖ Одним из соображений при выборе между сине-зелеными и последовательными обновлениями является потребность в ресурсах. Обратите внимание, что при сине-зеленом развертывании для API Posts необходимо удвоение ресурсов, тогда как при последовательном обновлении потребности в ресурсах увеличиваются незначительно

Во-вторых, дизайн приложения, позволяющий запускать несколько версий одновременно, дает больше преимуществ, чем последовательное обновление. Одним из них является ловкость.

Ваше программное обеспечение для облачной среды состоит из множества приложений, причем в каждый момент времени запускается несколько экземпляров каждого приложения, и основной целью в современную эпоху является частая эволюция этого программного обеспечения. Из наших предыдущих бесед вы знаете, что требование, согласно которому все различные микросервисы, из которых состоит программное обеспечение, должны обновляться в режиме реального времени, фактически снижает эту скорость. Подобные подходы приводили к появлению кошмарных диаграмм Гантта, которые отслеживали десятки или сотни зависимостей между различными компонентами и группами и при-

нуждали к выравниванию, прежде чем можно было вносить производственные изменения[1].

Теперь рассмотрим вариант, когда ваше приложение может быть использовано из многих других компонентов программного обеспечения. Если вам нужно обновить свое приложение, хотите ли вы, чтобы все ваши потребители одновременно адаптировались к новой версии? Конечно, ответ будет отрицательным, и если вы создали свое приложение таким образом, что несколько версий может работать бок о бок, одни пользователи могут использовать старые версии, а другие – новые. Мы называем это *параллельными развертываниями*.

Еще один вариант использования для параллельных развертываний – поддержка экспериментов. Примером, который я всегда использую для объяснения этой концепции, является рекомендательная система для электронной коммерции. Хотя у меня нет достоверных данных по этому поводу, готова поспорить, что у Amazon одновременно работает несколько версий такой системы. Алгоритмы, которые предлагают соответствующие товары покупателю, сложны и почти наверняка основаны на моделях, полученных с помощью машинного обучения. Незначительные изменения в алгоритме или даже только различные конфигурации в модели могут давать различные показатели кликабельности и объемы покупок. Чтобы оптимизировать результаты, розничный продавец может запускать несколько версий одновременно, анализировать результаты, а затем оставлять версию, которая работает более эффективно.

Параллельные развертывания являются мощными, но они придают повышенное значение версионированию вашего программного обеспечения. Когда трафик направляется более чем в одну версию приложения, версия должна быть идентифицируемой, а версия работающего программного обеспечения в конечном итоге состоит из версии развертываемого артефакта и версии конфигурации, примененной к работающим экземплярам.

> **ПРИМЕЧАНИЕ** Работающая версия приложения = версия развертываемого артефакта плюс версия конфигурации приложения.

Управление версией вашего развертываемого артефакта должно выполняться конвейерами сборки. Управление версиями конфигурации приложения осуществляется посредством системы управления версиями. Например, если вы используете Git, оператор приложения может фиксировать алгоритм SHA (Secure Hash Algorithm) в качестве версии.

Подведем итог основным пунктам этого раздела:

- необходимо, чтобы несколько экземпляров приложения действовало сообща даже во время события обновления жизненного цикла приложения;
- сборка приложений таким образом, чтобы несколько версий работало бок о бок, позволяет использовать последовательные обновления (а также предоставляет другие преимущества);
- последовательные обновления имеют преимущества по сравнению с синезелеными развертываниями.

Наконец, стоит отметить, что любое приложение, которое допускает использо-

[1] Дополнительную информацию о диаграммах Гантта можно найти на Википедии (**http://mng.bz/O2za**).

вание последовательных обновлений, также может быть развернуто в сине-зеленой манере. Но не наоборот. Чтобы использовать последовательное обновление, ваше приложение должно быть спроектировано таким образом, чтобы одновременно разрешалось запускать разные версии и конфигурации экземпляров; при использовании сине-зеленого развёртывания только одна версия/конфигурация обслуживает трафик сразу, поэтому специальной обработки не требуется.

Надлежащее поведение приложения в нескольких экземплярах во время события жизненного цикла приложения, такого как обновление, – это только половина картины. Также важно учитывать, как обновление приложения влияет на клиентов этого приложения. Как жизненный цикл одного приложения влияет жизненный цикл другого? Давайте рассмотрим это.

7.3. Координация между различными жизненными циклами приложения

Возвращаясь к нашей службе Posts, на рис. 7.3 было ясно, что два экземпляра приложения не будут обеспечивать согласованное поведение, что является очевидной проблемой. И теперь, когда вы только что переварили содержимое предыдущего раздела, вы видите, что это приложение нельзя обновить в последовательной манере. Допустим, вы обновили приложение, используя сине-зеленое развертывание. У службы Posts теперь есть новый секрет.

Это приводит нас к другой проблеме: события жизненного цикла приложения влияют не только на приложение, к которому они применяются, но и на другие зависимые приложения. Например, поскольку служба Connections' Posts является клиентом службы Posts, вы видите, что события жизненного цикла приложения одного приложения влияют на зависимое приложение. В этом конкретном сценарии, когда вы периодически меняете реквизиты доступа в службе Posts, также необходимо проделать то же самое и в службе Connections' Posts. Эта зависимость изображена на рис. 7.8.

Вопрос в том, как скоординировать эти обновления. Ясно, что это не относится ни к какому типу «транзакции»[1]. Ваше программное обеспечение – это распределенная система, и, как стало совершенно очевидно, сохранение автономности является важной характеристикой приложений, из которых состоит ваше ПО для облачной среды. Если вы можете обновлять клиентов и службы только в режиме реального времени, вы потеряете значительную часть этой автономности.

Вместо этого вы будете проектировать свои приложения таким образом, чтобы события жизненного цикла приложений в разных приложениях могли происходить независимо друг от друга, сохраняя при этом программное обеспечение полностью функциональным и без простоев. Ни один шаблон не дает решения для всех случаев. Поскольку вы являетесь разработчиком приложений, ваша задача – разработать правильный алгоритм.

---

[1] Время простоя для вашего программного обеспечения при обновлении всех зависимых компонентов – это один из типов транзакции, который явно не является нулевым временем простоя.

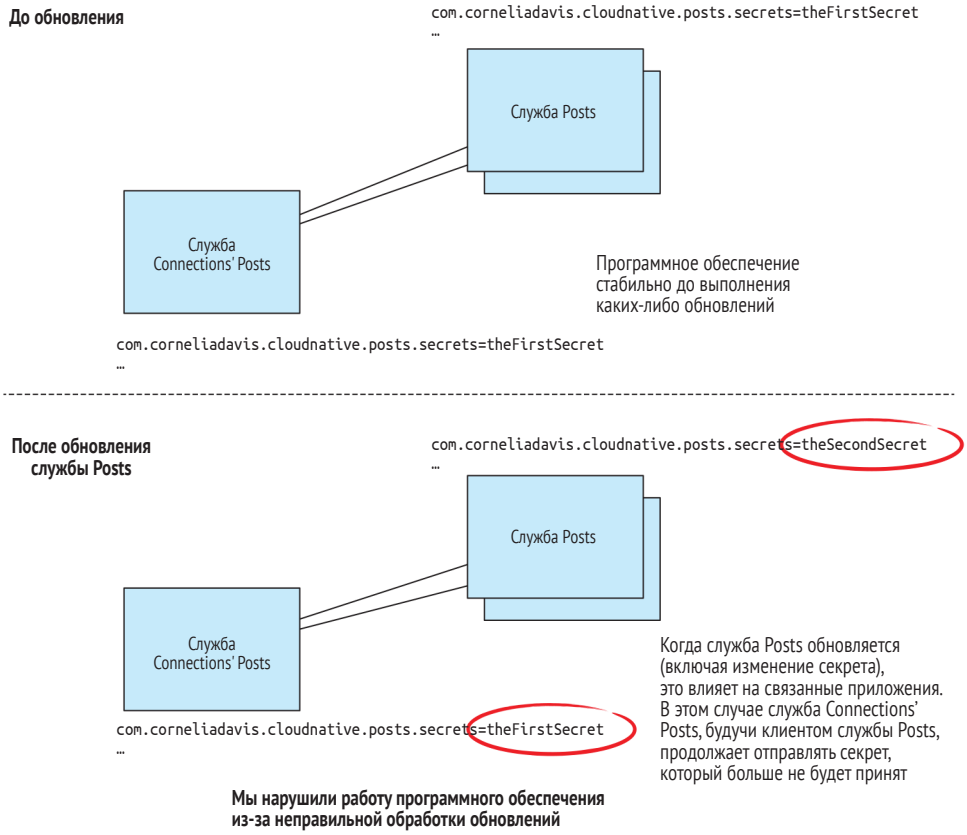


Рис. 7.8 ❖ События жизненного цикла приложения должны быть скоординированы между различными компонентами вашего программного обеспечения

ПРИМЕЧАНИЕ Вы должны проектировать свои приложения и документировать API таким образом, чтобы любые события жизненного цикла, которые влияют на зависимые службы, были исключены, минимизированы или могли быть адаптированы для этих клиентов.

Хорошо, я признаю, это все несколько абстрактно. Давайте вернемся к нашему примеру, чтобы было более понятно. Напомню, что ранее вы скомпрометировали секрет службы Posts, и в результате потребовалось обновить конфигурацию вашего работающего приложения. Это делается путем перезапуска всех экземпляров приложения с применением новой конфигурации. На данный момент не беспокойтесь о том, используете вы сине-зеленое развертывание или последовательное обновление. Первый, простой подход ставит следующую задачу:

- если сначала вы обновите службу Posts, запросы, поступающие от службы Connections' Posts, которые отправляют старый секрет, не будут выполнены;
- если сначала вы обновите службу Connections' Posts, она начнет отправлять новые учетные данные, и запросы к службе Posts, у которой по-прежнему старые данные, будут отклоняться;
- мы уже установили, что нельзя обновлять их одновременно без простоя.

Таким образом, нужно быть умнее, когда речь идет о дизайне вашего приложения. Для периодической смены реквизитов доступа существует общепринятый шаблон. Ключом к этому методу является тот факт, что вы будете реализовывать поэтапный подход к обновлению секрета; на одном этапе клиентская служба принимает более одного секрета. На рис. 7.9 изображен следующий порядок действий:

- перед началом обновления у вас есть один и тот же секрет, настроенный как на клиенте, так и на сервере;

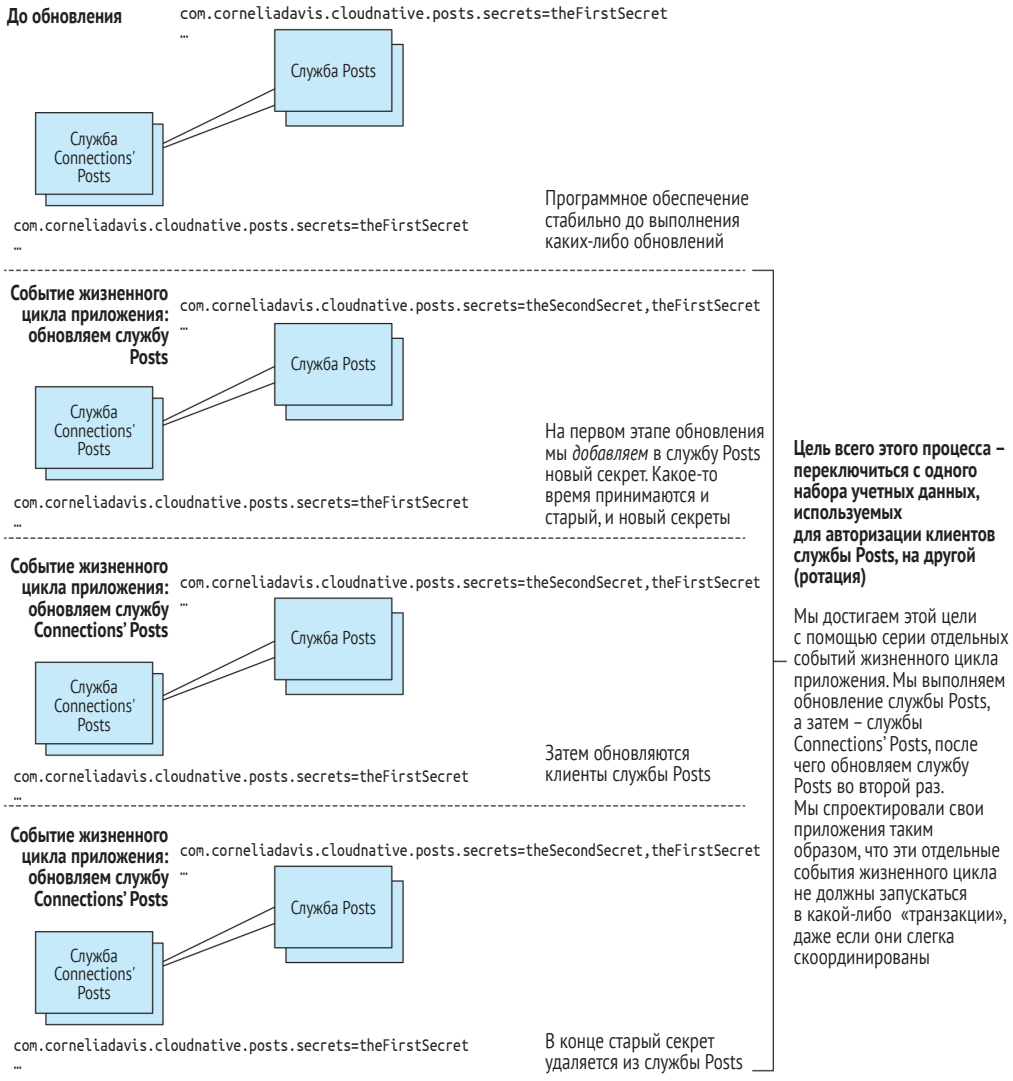


Рис. 7.9 ❖ Данный шаблон является примером рассмотрения, на которое должен обратить внимание разработчик программного обеспечения. Его цель состоит в том, чтобы обеспечить нулевое время простоя программного обеспечения во время событий жизненного цикла приложения, которые влияют на многочисленные приложения

- вы обновляете приложение Posts, *добавляя* новые учетные данные в *список* принятых учетных данных. Семантика авторизации заключается в том, что любые учетные данные в списке будут разрешать доступ (в большинстве случаев список ограничен двумя учетными данными). Обратите внимание, что клиент по-прежнему использует старые данные, но, поскольку этот секрет все еще находится в списке служб, запросы будут выполнены успешно;
- затем вы обновляете приложение Connections' Posts, заменяя старые учетные данные новыми. Поскольку новый секрет уже настроен в приложении Posts, которое теперь поддерживает список учетных данных, запросы от новых экземпляров клиента будут успешными;
- наконец, после завершения обновления клиента приложение Posts можно обновить, чтобы удалить старый секрет.

Если вы, как и я, в течение некоторого времени работали в секторе производства программного обеспечения, у вас изначально может возникнуть негативная реакция на этот процесс. Он включает в себя множество повторных развертываний, каждое из которых проходит через удаление и создание новых экземпляров приложения. Об этих старых инстинктах нужно забыть. Приложения для облачной среды предназначены для данного рода эфемерности – помните, что изменение – это правило, а не исключение, – и позволяют нам использовать ее, чтобы сделать наше программное обеспечение более надежным и управляемым.

Наконец, я хочу отметить, что этот дизайн также позволяет вам обновлять каждое из приложений в последовательном стиле. На первом этапе обновления служба Connections' Posts использует старый секрет, и старый, и новый экземпляры службы Posts принимают эти учетные данные, даже если новые экземпляры уже добавили обновленный секрет. На следующем этапе некоторые экземпляры службы Connections' Posts будут отправлять старый секрет, а другие будут отправлять новый; и снова служба Posts принимает оба. После полного обновления службы Connections' Posts она будет отправлять только новые учетные данные, поэтому во время второго обновления службы Posts и старая, и новая версии будут успешно работать. Это изображено на рис. 7.10.

После обсуждения архитектурных вопросов давайте теперь получим некоторый практический опыт работы с последовательными обновлениями, а также координацией событий жизненного цикла между приложениями. Мы будем делать это с правильной периодической сменой реквизитов доступа в нашем примере – исправляя сбой, который появился в конце главы 6.

7.4. Давайте посмотрим, как это работает: периодическая смена реквизитов доступа и жизненный цикл приложения

Чтобы следить за кодом (я попрошу вас, чтобы вы запустили все сразу), убедитесь, что у вас есть основная ветвь, извлеченная из нашего репозитория, и перейдите в каталог жизненного цикла приложения:

```
git checkout master
cd cloudnative-applifecycle
```

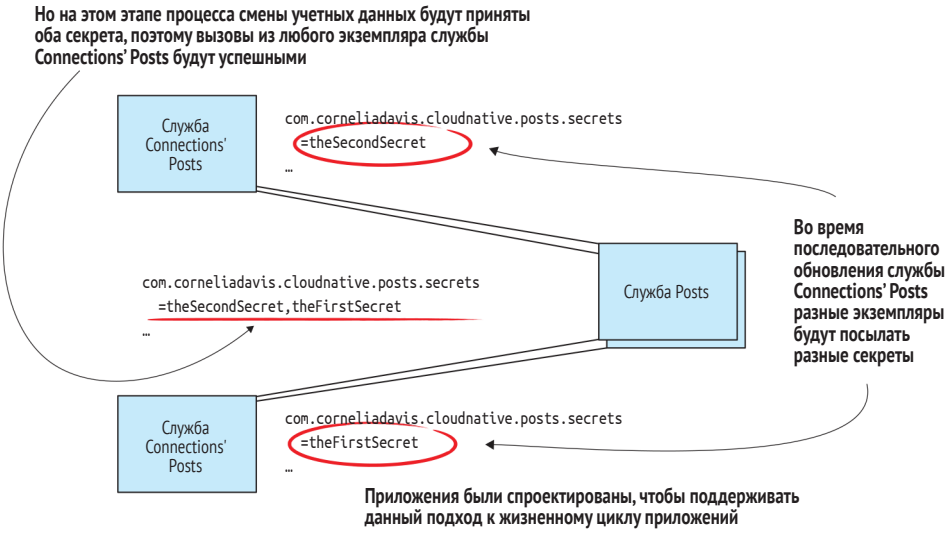


Рис. 7.10 ❖ Во время последовательного обновления службы Connections' Posts разные экземпляры будут посылать разные секреты. Приложения были спроектированы, чтобы поддерживать данный подход к жизненному циклу приложений

В этом примере мы реализуем шаблон периодической смены реквизитов доступа, показанный на рис. 7.9. Вы будете обновлять службу Posts, используя последовательное обновление, затем таким же образом обновите службу Connections' Posts и, наконец, службу Posts. Обратите внимание, что у каждого приложения есть два развернутых экземпляра; в начальном состоянии настроенные в них учетные данные совпадают для всех экземпляров приложения и координируются между службой Connections' Posts и службами Connections и Posts. Напомню, что каждое из трех приложений извлекает свою конфигурацию через сервер конфигурации из одного файла mycookbook.properties, который находится на GitHub. Обратите внимание, что это файл конфигурации для трех приложений, которые вместе образуют программное обеспечение под названием My Cookbook.

Чтобы следовать шаблону периодической смены реквизитов доступа, описанному в разделе 7.3, необходимо обновить реализации служб Posts и Connections, чтобы они могли хранить список допустимых секретов, любой из которых может использоваться для вызова службы. Хранение учетных данных осуществляется в экземпляре-одиночке класса `Utils`, ключевые фрагменты которого показаны в приведенном ниже коде.

Листинг 7.1 ❖ Utils.java

```
public class Utils implements
        ApplicationContextAware, ApplicationListener<ApplicationEvent> {

    // < Для краткости мы опустим некоторые строки >
    @Value("${com.corneliadavis.cloudnative.posts.secrets}")
    private String configuredSecretsIn;
    private Set<String> configSecrets;
```

```
    // <Для краткости мы опустим некоторые строки>

    @Override
    public void onApplicationEvent(ApplicationEvent applicationEvent) {

        if (applicationEvent instanceof ServletWebServerInitializedEvent) {
            ServletWebServerInitializedEvent
                servletWebServerInitializedEvent
                    = (ServletWebServerInitializedEvent) applicationEvent;
            this.port = servletWebServerInitializedEvent…
        } else if (applicationEvent instanceof ApplicationPreparedEvent) {
            configSecrets = new HashSet<>();
            String secrets[] = configuredSecretsIn.split(“,”);
            for (int i=0; i<secrets.length; i++)
                configSecrets.add(secrets[i].trim());
            logger.info(ipTag()
                        + “Posts Service initialized with secret(s): “
                        + configuredSecretsIn);
        }
    }

    public String ipTag() { return “[“ + ip + “:” + port +”] “; }

    public boolean isValidSecret(String secret) {
        return configSecrets.contains(secret);
    }

    // Следующий далее метод включен сюда, только чтобы облегчить процесс логирования,
    // который не существует в рабочем окружении;
    public String validSecrets() {
        String result = “”;
        for (String s : configSecrets)
            result += s + “,”;
        return result;
    }
}
```

Сперва я обращаю ваше внимание на метод `onApplicationEvent`, в особенности на тот случай, когда вы обрабатываете `ApplicationPreparedEvent`. Не вдаваясь в подробности богатого набора событий жизненного цикла приложения, реализованных с помощью Spring Framework, знайте, что `ApplicationPreparedEvent` инициируется, когда приложение было полностью инициализировано. Строка `configuredSecretsIn` была инициализирована через сервер конфигурации из свойства `com.corneliadavis.cloudnative.posts.secrets`. Здесь вы анализируете его и загружаете значения в набор (`Set`), чтобы проверка достоверности была поверхностной, как видно из определения метода `isValidSecret`.

Теперь, глядя на реализацию контроллера Posts в приведенном далее листинге, видно, что вам просто нужно проверить, действителен ли переданный секрет, прежде чем приступить к обработке. Если секрет недействителен, вы выводите и переданный секрет, и действительный, или те, что настроены в приложении. В реальном приложении вы бы не стали выводить эти значения в журнале, но это поможет вам при проведении экспериментов с данными концепциями.

Листинг 7.2 ❖ Метод из PostsController.java

```
@RequestMapping(method = RequestMethod.GET, value="/posts")
public Iterable<Post> getPostsByUserId(
    @RequestParam(value="userIds", required=false) String userIds,
    @RequestParam(value="secret", required=true) String secret,
    HttpServletResponse response) {

    Iterable<Post> posts;

    if (utils.isValidSecret(secret)) {

        logger.info(utils.ipTag()
            + "Accessing posts using secret " + secret);

        if (userIds == null) {
            logger.info(utils.ipTag() + "getting all posts");
            posts = postRepository.findAll();
            return posts;
        } else {
            ArrayList<Post> postsForUsers = new ArrayList<Post>();
            String userId[] = userIds.split(",");
            for (int i = 0; i < userId.length; i++) {
                logger.info(utils.ipTag()
                    + "getting posts for userId " + userId[i]);
                posts = postRepository.findByUserId(
                                Long.parseLong(userId[i]));
                posts.forEach(post -> postsForUsers.add(post));
            }
            return postsForUsers;
        }
    } else {
        logger.info(utils.ipTag()
           + "Attempt to access Post service with secret " + secret
           + " (expecting one of " + utils.validSecrets() + ")");
        response.setStatus(401);
        return null;
    }
}
```

Здесь показаны две ключевые части сервисной стороны шаблона: (1) секреты конфигурируются при запуске приложения и (2) для поддержки шаблона периодической смены реквизитов доступа с нулевым временем простоя служба допускает использование более одного действительного секрета за раз. Здесь я показала только код из службы Posts, но эта структура идентична и для службы Connections.

Давайте теперь посмотрим на клиентскую сторону в реализации приложения Connections' Posts в листинге 7.3. Базовая структура аналогична структуре приложений Posts и Connections.

Вы используете класс `Utils` для обработки конфигурации секретов. Затем в контроллере приложения, где выполняются вызовы к службам Posts и Connections, вы получаете доступ к значениям через объект-одиночку `utils`. Посмотрев сначала на код контроллера, вы видите, что он прост. Вы получаете доступ к сек-

рету служб Posts или Connections, который настроен в приложении, запрашивая значение у объекта utils, и отправляете его в строке запроса.

Листинг 7.3 ❖ Метод из ConnectionsPostsController.java

```
@RequestMapping(method = RequestMethod.GET, value="/Connections' Posts")
public Iterable<PostSummary> getByUsername(
    @CookieValue(value = "userToken", required=false) String token,
    HttpServletResponse response) {

    // <Для краткости мы опустим некоторые строки>

            // get connections
            String secretQueryParam
                = "?secret=" + utils.getConnectionsSecret);
            ResponseEntity<ConnectionResult[]> respConns
                = restTemplate.getForEntity(
                     connectionsUrl + username + secretQueryParam,
                     ConnectionResult[].class);
            // <Для краткости мы опустим некоторые строки>

            secretQueryParam = «&secret=» + utils.getPostsSecret();
            // get posts for those connections
            ResponseEntity<PostResult[]> respPosts
                = restTemplate.getForEntity(
                     postsUrl + ids + secretQueryParam,
                     PostResult[].class);
            // <Для краткости мы опустим некоторые строки>
}
```

Обращается к секрету, настроенному в приложении

Обращается к секрету, настроенному в приложении

Хотя большая часть класса Utils здесь похожа на тот же класс в приложении Posts, у нее есть один нюанс. Обратите внимание, что хотя свойства com.corneliadavis.cloudnative.connections.secrets и com.corneliadavis.cloudnative.posts.secrets извлечены из файла mycookbook.properties и каждое из них может содержать список секретов в службе сообщений Connections, нам нужен только самый последний. Вы установите метод эксплуатации, и это должно быть отражено в документации службы Connections' Posts, где самый свежий секрет всегда находится на первой позиции списка. Как видите, вы храните в состоянии объекта utils только один секрет для каждой из служб Posts и Connections. Чтобы прояснить это, свойство, которое настроено в приложении, является единственным секретом, даже если файл свойств содержит более одного. И снова приведенный далее код включает в себя вывод журнала, который не подходит для производственной системы, но является ценным средством для обучения.

Листинг 7.4 ❖ Метод из класса Utils службы ConnectionsPosts

```
@Override
public void onApplicationEvent(ApplicationEvent applicationEvent) {
    if (applicationEvent instanceof ServletWebServerInitializedEvent) {
            ServletWebServerInitializedEvent
                servletWebServerInitializedEvent
                    = (ServletWebServerInitializedEvent) applicationEvent;
            this.port = servletWebServerInitializedEvent...;
    } else if (applicationEvent instanceof ApplicationPreparedEvent) {
```

```
        connectionsSecret = connectionsSecretsIn.split(“,”)[0];
        postsSecret = postsSecretsIn.split(“,”)[0];
        logger.info(ipTag()
            + “Connection Posts Service initialized with Post secret: “
            + postsSecret + “ and Connections secret: “
            + connectionsSecret);
    }
}
```

Ладно, давайте запустим это в работу.

Настройка

Как и в примерах, приведенных в предыдущих главах, для запуска примеров необходимо установить следующие стандартные инструменты:

- Maven;
- Git;
- Java 1.8 (необязательно – это необходимо, только если вы планируете собирать образы контейнеров самостоятельно);
- Docker (необязательно – это необходимо, только если вы планируете собирать образы контейнеров самостоятельно);
- какой-нибудь клиент MySQL, такой как интерфейс командной строки `mysql`;
- какой-нибудь клиент Redis, например `redis-cli`;
- Minikube.

Сборка микросервисов (необязательно)

Поскольку я попрошу вас развернуть приложения в Kubernetes, а для этого требуются образы Docker, я предварительно создала эти образы и сделала их доступными в Docker Hub. Поэтому создание микросервисов из исходного кода не требуется.

Если вы еще этого не сделали, перейдите на главную ветку и замените каталог cloudnativeabundantsunshine на cloudnative-applifecycle:

```
git checkout master
cd cloudnative-applifecycle
```

Затем, чтобы собрать код (необязательно), введите приведенную ниже команду:

```
mvn clean install
```

При выполнении этой команды будут собраны три приложения с файлом с расширением JAR в целевом каталоге каждого модуля. Если вы хотите развернуть эти JAR-файлы в Kubernetes, вы также должны выполнить команды `docker build` и `docker push`, как описано в разделе «Использование Kubernetes требует создания образов Docker» в главе 5. Если вы это сделали, вам также нужно обновить YAML-файлы развертывания для Kubernetes, чтобы они указывали на ваши образы, а не на мои. Здесь я не повторяю эти шаги. Вместо этого в манифесте развертывания я указываю на образы, хранящиеся в моем репозитории на Docker Hub.

Запуск приложений

Запустите Minikube, если вы еще этого не сделали, как описано в разделе 5.2.2 главы 5. Чтобы начать с чистого листа, удалите все развертывания, которые могли остаться от предыдущей работы. Для этого я даю вам скрипт: `deleteDeploymentComp-`

lete.sh. Этот простой bash-скрипт позволяет поддерживать службы MySQL и Redis в работающем состоянии.

При вызове скрипта без параметров будут удалены только три развертывания микросервисов; при вызове сценария с аргументом all также будут удалены MySQL и Redis. Службы Kubernetes не удаляются – это наверняка избавит вас от настройки URL-адресов в каждом из манифестов развертывания приложения.

Убедитесь, что у вас чистая среда, с помощью этой команды:

```
$ kubectl get all
NAME                          READY STATUS    RESTARTS  AGE
pod/mysql-75d7b44cd6-s8zcr    1/1   Running   0         70m
pod/redis-6bb75866cd-kf99k    1/1   Running   0         72m
pod/sccs-787888bfc-x9p2m      1/1   Running   0         73m

NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP PORT(S)
service/connections-svc       NodePort    10.103.148.230  <none>      80:30955/TCP
service/connectionsposts-svc NodePort    10.104.253.33   <none>      80:31742/TCP
service/kubernetes            ClusterIP   10.96.0.1       <none>      443/TCP
service/mysql-svc             NodePort    10.107.78.72    <none>      3306:30917/TCP
service/posts-svc             NodePort    10.110.192.11   <none>      80:32119/TCP
service/redis-svc             NodePort    10.108.83.115   <none>      6379:31537/TCP
service/sccs-svc              NodePort    10.107.16.107   <none>      8888:30455/TCP

NAME                       READY UP-TO-DATE AVAILABLE AGE
deployment.apps/mysql      1/1   1          1         70m
deployment.apps/redis      1/1   1          1         72m
deployment.apps/sccs       1/1   1          1         73m

NAME                                  DESIRED CURRENT READY AGE
replicaset.apps/mysql-75d7b44cd6      1       1       1     70m
replicaset.apps/redis-6bb75866cd      1       1       1     72m
replicaset.apps/sccs-787888bfc        1       1       1     73m
```

Обратите внимание, что MySQL и Redis по-прежнему работают. Если вы очистили Redis и MySQL, разверните каждую из них с помощью приведенных ниже команд и создайте базу данных поваренной книги с помощью этих двух команд:

```
kubectl create -f mysql-deployment.yaml
kubectl create -f redis-deployment.yaml
mysql -h $(minikube service mysql-svc --format “{{.IP}}”) \
    -P $(minikube service mysql-svc --format “{{.Port}}”) -u root -p
mysql> create database cookbook;
```

После выполнения нижеизложенных шагов развертывание будет выглядеть так, как показано в первом этапе на рис. 7.9. У вас будет по две службы Connections и Posts и два экземпляра службы Connections’ Posts. Для достижения этой топологии сейчас вам еще нужно отредактировать манифесты развертывания. Эти шаги, кратко изложенные здесь, подробно описаны в главе 5:

1) настройте службы Connections и Posts с этими значениями:

URL-адрес службы MySQL	`minikube service mysql-svc --format «jdbc:mysql://{{.IP}}:{{.Port}}/cookbook»`
URL-адрес SCCS	`Minikube service sccs-svc --format «http://{{.IP}}:{{.Port}}»`

2) разверните службу Connections:

```
kubectl apply -f cookbook-deployment-connections.yaml
```

3) разверните службу Posts:

```
kubectl apply -f cookbook-deployment-posts.yaml
```

4) настройте службу сообщений Connections таким образом, чтобы она указывала на службы Posts, Connections и Users, а также на службу Redis. Эти значения можно найти с помощью приведенных ниже команд:

URL-адрес службы Posts	`minikube service posts-svc --format «http://{{.IP}}:{{.Port}}/posts?userIds=» --url`
URL-адрес службы Connections	`minikube service connections-svc --format «http://{{.IP}}:{{.Port}}/connections/» --url`
URL-адрес службы Users	`minikube service connections-svc --format «http://{{.IP}}:{{.Port}}/users/» --url`
IP-адрес Redis	`minikube service redis-svc --format «{{.Port}}»`
Порт Redis	`minikube service redis-svc --format «{{.Port}}»`
URL-адрес SCCS	`Minikube service sccs-svc --format «http://{{.IP}}:{{.Port}}»`

5) разверните службу Connections' Posts:

```
kubectl apply -f cookbook-deployment-connectionsposts.yaml
```

Вы можете протестировать свое развертывание, выполнив команды `curl` для любого из микросервисов. Службы Posts и Connections требуют, чтобы секрет передавался в строке запроса, а служба Connections' Posts требует выполнить вход, прежде чем содержимое будет доставлено. Вот как выглядят эти команды:

```
curl -i $(minikube service --url connections-svc)/connections?secret=anyval
curl -i $(minikube service --url connections-svc)/users?secret=anyvalue
curl -i $(minikube service --url posts-svc)/posts?secret=foobar
curl -X POST -i -c cookie \
   $(minikube service --url connectionsposts-svc)/login?username=cdavisafc
curl -b cookie \
   $(minikube service --url connectionsposts-svc)/connectionsposts
```

> **ПОДСКАЗКА** Простой способ поиска секретов, сконфигурированных в программном обеспечении, – это использовать команду `curl` для служб Posts и Connections с `?secret = anyvalue`, и посмотрите журналы; помните, что вы выводите принятое значение в журнал.

Давайте теперь выполним первую часть процесса обновления. Это можно сделать, обновив файл конфигурации для развертывания программного обеспечения, выполнив после этого последовательное обновление для службы Posts. Напомню, что вы сохраняете конфигурацию в GitHub и используете SCCS (Spring Cloud Config Server), чтобы доставить эту конфигурацию экземплярам приложения. Отредактируйте файл mycookbook.properties в репозитории cloud-native-config, обновив эту строку:

```
com.corneliadavis.cloudnative.posts.secrets=originalSecret
```

Добавьте новый секрет в начало списка:

```
com.corneliadavis.cloudnative.posts.secrets=newSecret,originalSecret
```

Вы должны зафиксировать и поместить эти изменения в свой репозиторий на GitHub. Теперь нам нужно выполнить последовательное обновление для службы Posts, и для этого мы воспользуемся командой `kubectl apply`. Поскольку данное изменение конфигурации осуществляется через SCCS, и, следовательно, Kubernetes не видит изменения в свойстве, нужно сделать что-то, чтобы заставить Kubernetes использовать циклы в случае с приложениями. Хитрость, которую я использую, заключается в том, что в YAML-файле развертывания есть переменная среды, которую можно применять, когда вы хотите, чтобы Kubernetes обновлял экземпляры вашего приложения. Просто нужно изменить значение `VERSIONING_TRIGGER` на что-то новое. Это делается в файле cookbook-deploy-posts.yaml

```
- name: VERSIONING_TRIGGER
  value: «1»
```

Обновляет значение, в результате чего получается что-то новое – обычно я просто увеличиваю число

Перед выполнением команды для запуска последовательного обновления в окне терминала настройте наблюдение за модулями, которые в данный момент работают в вашей среде:

```
watch kubectl get pods
```

Теперь начните обновление с помощью этой команды:

```
kubectl apply -f cookbook-deployment-posts.yaml
```

В окне наблюдения вы увидите, как создаются новые экземпляры службы Posts, а старые отключаются, а затем в конечном итоге удаляются. Это процесс обновления, который изображен на рис. 7.6. (Не знаю, как для вас, но для меня наблюдать за автоматизацией жизненного цикла приложений такого типа в первый раз было чертовски круто!) Теперь давайте применим команду `curl`, экспериментируя с разными секретами. Начнем с плохого:

```
curl -i $(minikube service --url posts-svc)/posts?secret=aBadSecret
```

Загляните в журналы двух экземпляров модуля. В одном из них вы найдете сообщение, которое выглядит примерно так:

```
Attempt to access Post service with secret aBadSecret (expecting one of
    newSecret,oldSecret,)
```

Я еще раз напомню вам, что никогда не следует отправлять секреты в файлы журналов, но здесь вы это делаете, потому что это идет на пользу нашим упражнениям. Это сообщение показывает, что к службе Posts была применена новая конфигурация. Теперь вы можете вызвать эту службу с любым из секретов и получить ответ.

Служба Connections' Posts по-прежнему использует `oldSecret`, поэтому давайте обновим его.

Мы уже обновили конфигурацию в GitHub, поэтому нам нужно, чтобы Kubernetes выполнял только последовательное обновление. Это делается путем редактирования файла cookbook-deployconnectionsposts.yaml, используя `VERSIONING_TRIGGER` так же, как мы это сделали в случае со службой Posts:

```
- name: VERSIONING_TRIGGER
  value: «1»
```

Обновляет значение, в результате чего получается что-то новое – обычно я просто увеличиваю число

Теперь выполните приведенную ниже команду, чтобы запустить последовательное обновление:

```
kubectl apply -f cookbook-deployment-connectionsposts.yaml
```

Как после обновления, так и во время него вы можете вызвать службу Connections'o Posts. В зависимости от экземпляра старый или новый секрет будет отправлен в службу Posts. Опять же, благодаря шаблону, который вы реализовали, все будет работать так, как ожидалось.

Наконец, после того как служба Connections' Posts будет полностью обновлена, вы можете удалить старый секрет из конфигурации (не забудьте зафиксировать это и загрузить на GitHub), обновить YAML-файл развертывания, чтобы использовать VERSIONING_TRIGGER, и выполнить эту команду:

```
kubectl apply -f cookbook-deployment-posts.yaml
```

После того как Kubernetes завершит последовательное обновление службы Posts, периодическая смена реквизитов доступа будет завершена. Вы просто обновили свое программное обеспечение *без каких бы то ни было простоев* и смогли использовать последовательное обновление (автоматизация жизненного цикла приложения, встроенная в Kubernetes), потому что ваше программное обеспечение было специально разработано для его поддержки.

7.5. Работа с эфемерной средой выполнения

К настоящему времени должно быть ясно, что одной из основных характеристик приложений для облачной среды является то, что они постоянно удаляются и создаются заново. Вы только что убедились в этом в примере с периодической сменой реквизитов доступа, когда вы дважды обновили все экземпляры приложений Posts и Connections, сначала для того, чтобы добавить новый секрет, а затем – чтобы удалить старый. Опять же, я понимаю ваше возможное отвращение к такому методу; это идет вразрез с глубоко укоренившейся верой в то, что стабильность – это хорошо, а изменение – плохо, но в облаке изменения неизбежны и даже могут дать новый тип стабильности.

Однако, как и многие другие шаблоны и методы для облачной среды, использование этой новой парадигмы приносит с собой каскадные эффекты – новые проблемы для вас, человека, являющегося разработчиком приложений. Главное, о чем я хочу рассказать сейчас, – это влияние, которое эфемерность этих контекстов времени выполнения оказывает на управляемость наших приложений. В этом разделе я хочу рассказать о двух темах: (1) поиск и устранение неполадок и (2) повторяемость.

Позвольте мне сначала рассказать о последней теме лишь кратко, потому что я уже несколько раз говорила об этом. Суть состоит в том, что развертывание приложения (конкретные развертываемые биты, способ его запуска и контекст, в котором оно выполняется) должно быть воспроизводимым на 100 %. Платформы приложений для облачной среды предоставляют множество функций, которые поддерживают данную цель, поэтому важно изучить передовые методы исполь-

зования таких платформ. Но поскольку я собираюсь немного подробнее рассказать о поиске и устранении неисправностей, позвольте мне отметить, что лучшие из этих платформ при правильной настройке не позволят вам подключиться к среде выполнения по протоколу SSH. Почему? Говоря проще, чтобы убедиться, что вы не можете создать работающий экземпляр своего приложения, который не воспроизводится.

Предположим, вам разрешили подключиться к контейнеру по протоколу SSH, в котором работает ваше приложение. Находясь там, вы делаете что-то, что, по вашему мнению, предназначено только для устранения неполадок, то, что не является частью производственной конфигурации приложения. Например, вы можете открыть порты для включения инструментов мониторинга или установить дополнительные пакеты. Вы устраняете проблему и, поскольку являетесь ответственным разработчиком, вы возвращаетесь к исходному коду или конфигурации и отражаете изменения там. Вы оставляете этот контейнер работающим – почему бы и нет; он прекрасно работает. Но теперь контейнер – это снежинка; хотя вы думаете, что отразили все «исправления» в приложении и конфигурации, возможно, что только у этого единственного контейнера есть правильная конфигурация, которая заставит его работать. Позже, когда экземпляр заменяется (по ряду причин), с новым экземпляром приложения могут (или не могут) вернуться старые проблемы. Ограничение операций, позволяющих создавать снежинки, значительно повышает управляемость развертывания программного обеспечения.

Что подводит меня ко второму пункту. Если вы не можете подключиться к экземпляру по протоколу SSH, как вы узнаете, что происходит, когда ваши приложения работают не так, как ожидалось? Краткий ответ: с помощью логирования и метрик. Являясь разработчиком программного обеспечения, вы играете важную роль в обеспечении того, чтобы журналы и фиды метрик содержали информацию, достаточную для диагностики проблем. И тут появляется еще один поворот в этом графике, который связан именно с жизненным циклом приложения: возможно, к тому времени, когда вы будете устранять неполадки, экземпляра приложения, с которым возникла проблема, уже не будет. Если в приложении произошел сбой, от этого контейнера, вероятно, избавились, и его место занял новый экземпляр.

Практика утилизации проблемных экземпляров приложений настолько распространена в платформах для облачной среды, что руководство по 12-факторному приложению включает в себя фактор № 11 «Рассматривайте журнал как поток событий». Цель этого совета – установить контракт или API для «публикации» данных журнала, чтобы платформы, где работают приложения для облачной среды, могли позаботиться о том, чтобы эти данные были доступны, даже если *они* поддерживают контроль над жизненным циклом приложения. В истории, которая звучит знакомо, поскольку вы видели подобные аргументы, когда я описывала использование переменных среды для данных конфигурации, потоки stdout и stderr существуют практически в каждой среде выполнения, и каждый язык программирования/фреймворк поддерживает запись в эти потоки. И это простой стандарт; не нужно искать файлы журналов, хранящиеся в специальных каталогах, или даже выполнять ротацию файлов журналов. Все данные журнала передаются в потоковом режиме и могут обрабатываться самой платформой.

ПОДСКАЗКА Итак, это подводит нас к кульминации: записывайте свои логи в stdout и stderr.

Это не значит, что вы должны просто использовать вызовы `System.out` по всему коду. Есть причина, по которой были созданы такие библиотеки журналирования, как Apache Log4j или Logback. Но их можно настроить, и часто по умолчанию, так чтобы направлять любой вывод журнала в stdout и stderr.

История метрик не так проста, в основном потому, что данные метрик по своей природе более структурированы, чем данные журналов, и сильно различаются в зависимости от приложения и платформы. Не существует единого набора данных метрик, вокруг которых можно создать платформы и инструментальные средства для предоставления всей необходимой информации. Тем не менее это важная тема, и некоторые де-факто стандартные методы и инструменты начали появляться. В главе 11 рассказывается о немногих из них.

Надеюсь, к настоящему моменту стало ясно, что хотя у меня есть целая глава, посвященная этой теме, я кратко расскажу здесь об устранении неполадок из-за влияния, которое жизненный цикл приложения оказывает на него. Как и в случае с основной логикой приложения, то, как вы обрабатываете данные наблюдаемости, также должно учитывать более динамичную природу жизненного цикла приложения. Я выделила одно слово несколькими абзацами ранее («они»), которое подчеркивает суть: в мире облачной среды люди не контролируют жизненный цикл приложения; это делают системы (а в лучшем случае интеллектуальные системы, такие как Kubernetes или Cloud Foundry). Системы требуют гораздо более сильных контрактов или API, а вы несете ответственность за обеспечение того, чтобы ваши приложения соответствовали этим контрактам.

7.6. Видимость состояния жизненного цикла приложения

В разделе 7.3 я говорила о проблемах жизненного цикла приложений в разных, но связанных приложениях. В частности, вы увидели, как удовлетворить эксплуатационные нужды с помощью серии событий жизненного цикла. Есть еще один элемент в отношениях между приложениями, о котором я хочу сейчас поговорить. Он возникает, когда одно приложение должно знать о событиях жизненного цикла другого приложения.

Например, как вам хорошо известно (и что уже вызывает сильное раздражение), когда по какой-либо причине служба Posts создается повторно (я имею в виду *службу* Kubernetes, а не модули), вам необходимо повторно развернуть службу Connections' Posts с конфигурацией, включающей в себя новый URL-адрес этой службы. Если бы вместо этого вы могли автоматически обновлять последнее приложение после такого изменения в первом, ваш опыт был бы намного лучше. (Потерпите еще чуть-чуть; до того, как мы все исправим, осталось всего несколько страниц!)

С точки зрения жизненного цикла приложения это означает, что служба Connections' Posts зависит от осведомленности относительно того, когда со службой Posts происходят события жизненного цикла. В простейшем смысле служба Posts отвечает за обеспечение доступности своего жизненного цикла. Например, на рис. 7.11 показано событие жизненного цикла приложения, которое транслируется при запуске приложения.

Делаем еще один шаг вперед. На рис. 7.12 показана зависимость от этого события. На этой диаграмме изначально служба Posts доступна для службы Connec-

tions' Posts по адресу 10.24.1.35, но когда новая служба Posts запускается по адресу 10.24.1.128, она отвечает за передачу этой информации, и службу Connections' Posts нужно обновить, используя этот IP-адрес.

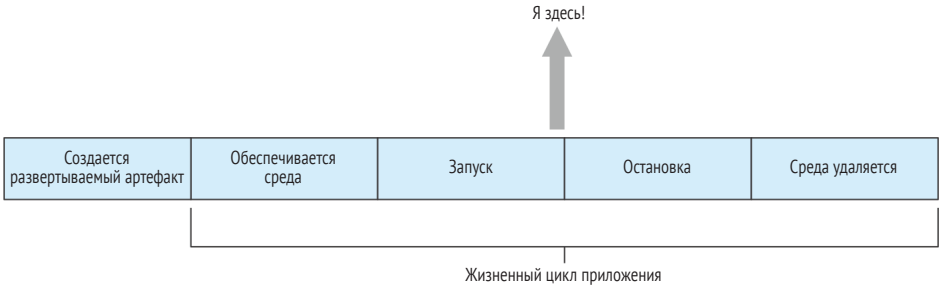


Рис. 7.11 ❖ Запуск приложения – это важное событие, в котором могут быть заинтересованы многие другие компоненты ПО для облачной среды

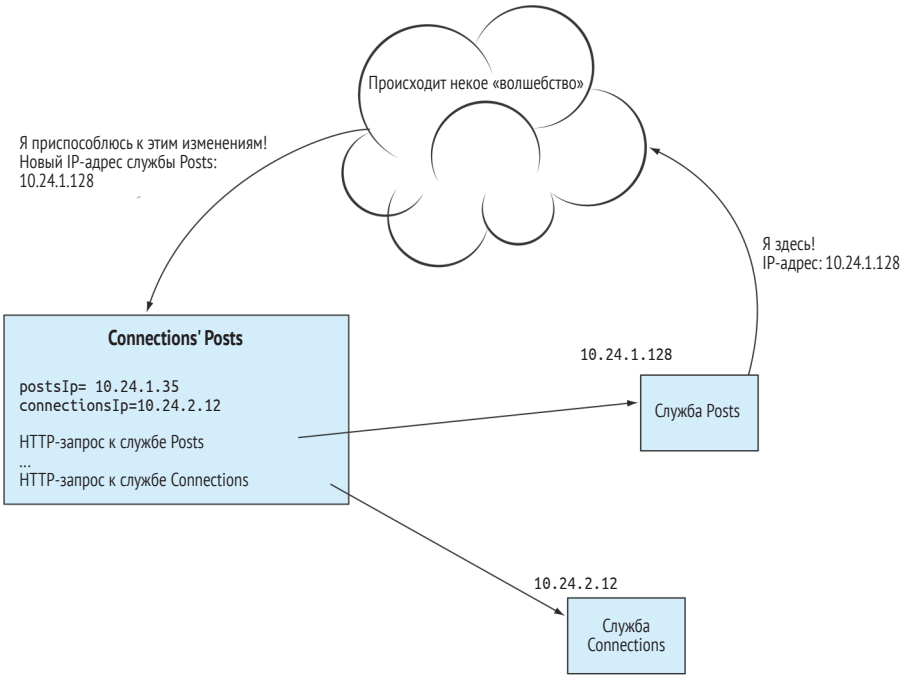


Рис. 7.12 ❖ Приложения несут ответственность за передачу событий жизненного цикла, поскольку будут затронуты другие компоненты. Эти компоненты также несут ответственность за адаптацию к этим изменениям

Точный механизм того, как обрабатывается эта передача из службы Posts, где она публикуется, и как заинтересованные стороны могут собирать соответствующую информацию, не имеет значения на данном этапе нашего обсуждения. На

рис. 7.12 это изображено в виде «волшебного» облака, и мы вернемся к этому позже. До этого момента мы занимались реализацией всего этого протокола – искали новый IP-адрес и порт с помощью команды `minikube service list` и редактировали YAML-файлы. Но это нужно автоматизировать, и суть должна быть ясна: события жизненного цикла приложения в службе Posts влияют на другие части вашего программного обеспечения, и вы должны особо учитывать это.

Говоря об обязанностях по обе стороны отношений, мое изложение было несколько расплывчатым. Я признаю, что подразумевала, что вы, разработчик, несете ответственность за трансляцию или потребление событий. Хотя полный ответ немного сложноват, и вы будете изучать его более детально по мере прочтения текста, короткий ответ выглядит так: если вы используете платформу приложений для облачной среды, она, как правило, позаботится об этих проблемах вместо вас. На данный момент я просто хочу, чтобы вы оценили эту зависимость.

Хотя вы можете понять эту проблему, я надеюсь, что вы читаете то, что я писала до сих пор, со здоровой долей скептицизма. На рис. 7.12 вы видите координацию между множеством компонентов, которые сильно распределены по ненадежной сети, но я предположила, что надлежащие операции зависят от транслируемых событий жизненного цикла, доходящих до всех заинтересованных сторон.

Вот где истина: транслируемое событие, как я его здесь описала, следует рассматривать как оптимизацию. Причина, по которой я начала с оптимизации, заключается в том, что она прекрасно иллюстрирует проблему. У вас есть целая куча фрагментов, которые должны действовать согласованно, чтобы заставить современное программное обеспечение работать. В абстрактном смысле это просто: событие жизненного цикла приложения должно быть получено теми, на кого повлияет изменение состояния. Но реальность сложнее: все это должно происходить в условиях множества сценариев сбоев. События жизненного цикла приложения не всегда генерируются; когда это происходит, они иногда теряются; и даже если это не так, иногда они не распознаются или не используются компонентами, которые должны это делать.

Чтобы понять это, позвольте мне показать на рис. 7.13 гораздо менее упрощенный жизненный цикл приложения, чем тот, что был на рис. 7.1 и 7.11. В дополнение к моделированию возможных сценариев сбоя на рис. 7.13 также показан более широкий набор возможных переходов между состояниями жизненного цикла приложения. После развертывания приложения запуск может быть успешным (запущено и отвечает) или может произойти сбой – приложение может работать, но не так, как нужно (запущено и не отвечает). Приложение, которое успешно запущено и было работоспособно в течение некоторого времени, также может выйти из строя или, что еще хуже, может продолжать работать, но не отвечать. Независимо от того, изящно оно было закрыто или не очень, в какой-то момент приложение и его среда исчезнут.

Немного расширим наш сценарий. Представим, что службе Connections' Posts необходимо знать не только, когда запускаются новые службы Posts, но и когда они отключаются. Эта диаграмма позволяет рассмотреть оба случая. Обратите внимание, что я добавила несколько толстых стрелок с аннотациями. Они эквивалентны стрелке на рис. 7.11, но обновлены до более сложной модели жизненного цикла. Аннотации в верхней части показывают трансляцию событий «счастливого пути»: когда приложение успешно запускается, оно объявляет о своем сущест-

вовании, а когда оно изящно закрывается, оно может сначала сообщить всем, что оно завершает работу. Но что происходит, когда, как показано стрелками, направленными вниз, запуск не проходит так замечательно, или когда приложение аварийно завершает работу, лишая себя возможности сообщить миру о своей надвигающейся гибели?

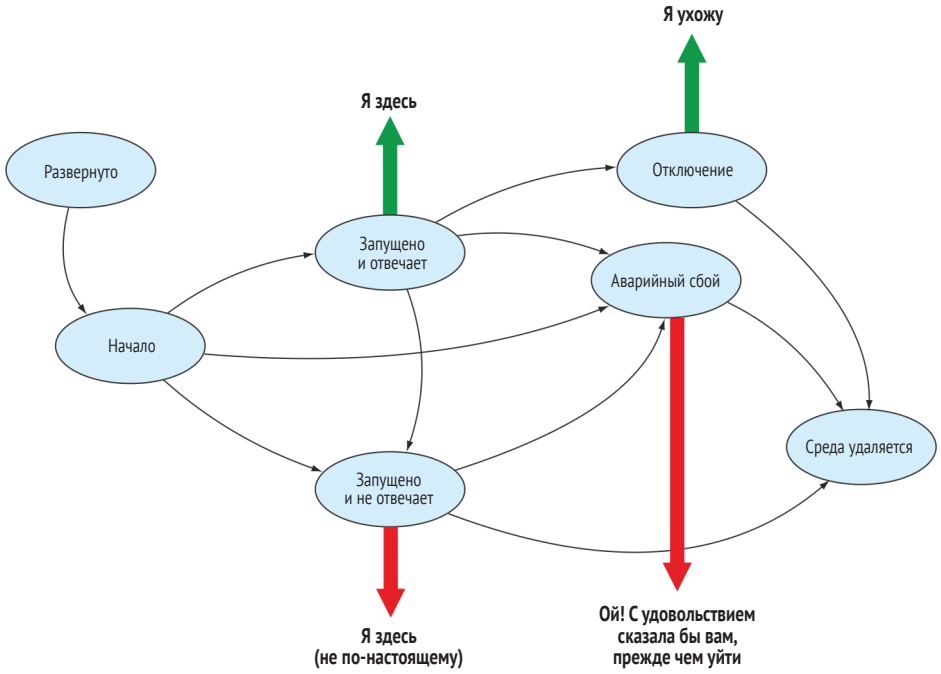


Рис. 7.13 ❖ Состояния жизненного цикла приложения и переходы между ними. При использовании подхода «счастливый путь», когда приложения запускаются и останавливаются, есть возможность передавать события изменения жизненного цикла приложения. Но когда что-то идет не так, передачи с изменением состояния будут некорректными или даже не будут сгенерированы вообще

Не будем слишком драматичными. Это часть магии облачной среды. Приложения созданы не только для того, чтобы работать по счастливому пути, но также для того, чтобы продолжать работать или самоисцеляться, когда дела идут плохо. Чтобы справиться с этим конкретным обстоятельством, ответ заключается в конечных точках работоспособности в сочетании с проверками работоспособности / респондерами, и вы как разработчик играете важную роль в координации между этими вещами. Концепция проста: конечная точка работоспособности представляет данные, обозначающие состояние приложения, а проверки работоспособности / респондеры реализуют некий тип цикла управления, который опрашивает и воздействует на этот статус, что устраняет вышеупомянутые недостатки потерянных событий жизненного цикла с избыточностью. Цикл управления делает это непрерывно (скажем, каждые 10 секунд), поэтому если один или даже пара попыток обмена данными не увенчались успехом из-за кратковременных сбо-

ев, следующая будет успешной, и система продолжит функционировать. Это еще один пример того, что я уже называла *согласованностью в конечном счете*.

На рис. 7.14 представлены две диаграммы последовательности, изображающие этот базовый шаблон. Всякий раз во время запроса приложение отвечает своим текущим состоянием жизненного цикла, а цикл управления регулярно спрашивает и отвечает соответствующим образом. Первая диаграмма последовательности показывает, что происходит, когда система функционирует должным образом: цикл управления проверяет конечную точку работоспособности приложения, и после получения ответа, указывающего на то, что все в порядке, просто ждет следующего интервала и снова спрашивает. Вторая диаграмма последовательности показывает, что происходит, когда приложение работает, но не отвечает или если конечная точка работоспособности возвращает ошибку. Один сбой не обязательно приведет к действиям по ликвидации этих последствий, но при повторных сбоях это произойдет (вы увидите, как это работает, через мгновение).

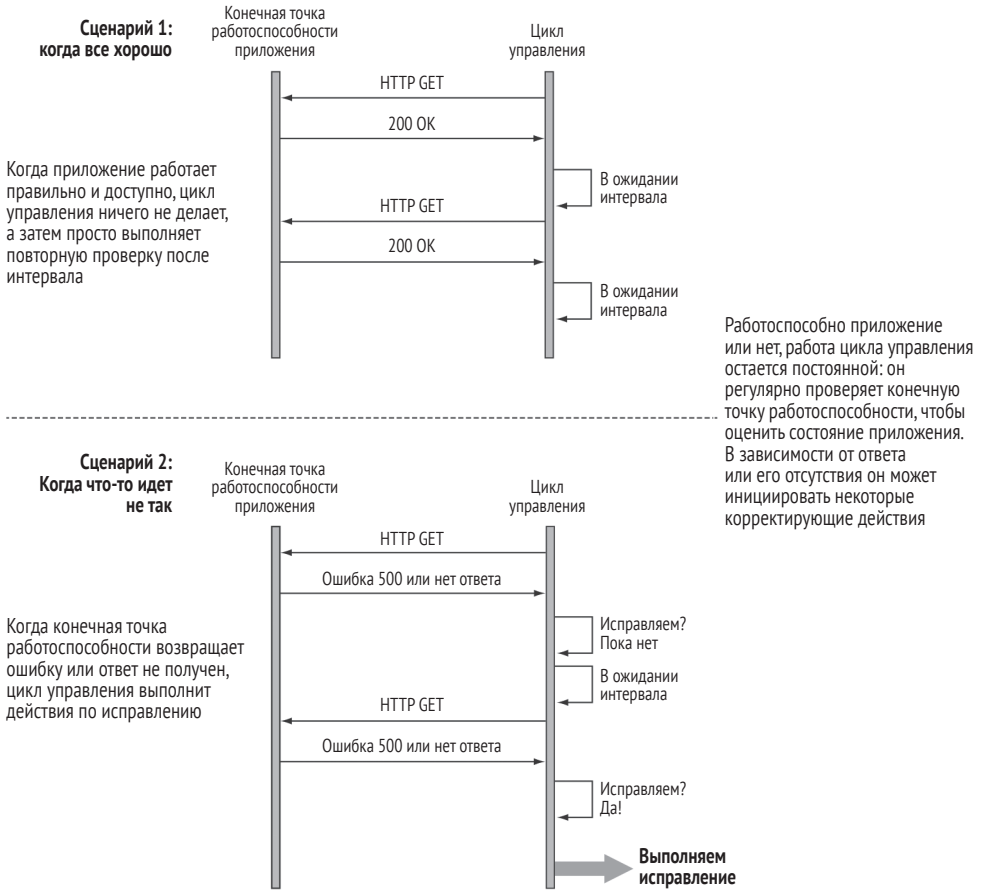


Рис. 7.14 ❖ Циклы управления играют важную роль в системах для облачной среды, обеспечивая избыточность, которая компенсирует «сбои» в системе

Я подчеркнула, что, будучи разработчиками, вы играете важную роль в том, чтобы заставить все это работать, но я хочу пояснить, что не считаю вас ответственными за обе стороны этого протокола. Цикл управления будет идти из платформы для облачной среды – Kubernetes, Cloud Foundry и чего-то похожего[1]. Вы несете ответственность за конечные точки работоспособности приложений и за создание приложений, которые можно исправить, – создав новые экземпляры, – когда они испытывают проблемы. Прямо сейчас мы сосредоточимся на конечной точке. На том, что связано с ликвидацией неприятных последствий, – вот о чем эта книга. Вы несете ответственность за реализацию конечной точки работоспособности, которая точно отражает состояние приложения. Возможно, вы захотите убедиться, например, что любое подключение к постоянным службам, таким как базы данных, функционирует.

Система в целом построена так, чтобы быть устойчивой к сбоям – перебоям в работе сети, сбоям приложений и т. д., а цикл управления обеспечивает необходимую избыточность. И теперь понятно, почему более ранний дизайн, основанный на трансляции, следует рассматривать как оптимизацию. Вместо того чтобы ждать, когда следующий цикл управления транслирует состояние, событие изменения состояния жизненного цикла тотчас же инициирует трансляцию. Если по какой-либо причине событие потеряно, при следующем запуске цикла управления появится другое.

7.6.1. Давайте посмотрим, как это работает: конечные точки работоспособности и проверки

Ладно, все это достаточно абстрактно. Давайте сделаем это на примере. Если вы запустили образцы, о которых шла речь ранее в этой главе, у вас уже все готово. Код, который вы использовали, уже включает в себя то, что я хочу здесь продемонстрировать.

Реализуя подход с использованием опроса, который я описывала ранее, вы добавите конечную точку `/healthz` в каждую из служб. Слегка вымышленный код просто проверяет булев член класса. Когда для него установлено значение `true` (по умолчанию), он возвращает код состояния успеха, а когда для него установлено значение `false`, приложение долгое время бездействует, что фактически делает его безответным. У вас есть приложение, которое работает и находится в безответном состоянии, как показано ниже.

Листинг 7.5 ❖ Метод из Posts_Controller.java

```
    @RequestMapping(method = RequestMethod.GET, value="/healthz")
    public void healthCheck(HttpServletResponse response)
                                          throws InterruptedException {

        if (this.isHealthy) response.setStatus(200);
        else Thread.sleep(400000);

}
```

Во второй половине показан цикл управления, который непрерывно опрашивает конечную точку работоспособности, извлекая данные и, при необходимости,

[1] Если у вашей платформы нет циклов управления данного типа, это не платформа для облачной среды.

воздействуя на них, и происходит из Kubernetes. В манифесте развертывания Kubernetes есть несколько новых строк:

Листинг 7.6 ❖ Выдержка из файла cookbook-deploy-posts.yaml

```
        livenessProbe:
          httpGet:
            path: /healthz
            port: 8080
          initialDelaySeconds: 60
          periodSeconds: 5
```

Эти строки конфигурируют данный цикл управления, проверку живости, поэтому каждые 5 секунд Kubernetes будет отправлять запрос с помощью метода GET по протоколу HTTP к конечной точке `/healthz` этого модуля (Kubernetes также поддерживает проверки живости по протоколу TCP). При запуске нового модуля Kubernetes будет ждать 60 секунд, прежде чем будет инициирован цикл управления, предлагая службе время выполнить инициализацию до того, как она ожидает, что конечная точка работоспособности будет точно отражать состояние приложения. Если Kubernetes получит код состояния ошибки или не получит ответа, он перезапустит контейнер. Давайте посмотрим, как это работает.

Смотрим, как это работает

Если у вас не запущено программное обеспечение, выполните действия, описанные в разделе 7.5 «Запуск приложений». Пожалуйста, передавайте логи с каждого из двух ваших модулей службы Posts в двух окнах терминала. В случае с двумя моими текущими модулями я использовала приведенные ниже команды в этих двух отдельных окнах:

```
$ kubectl logs -f posts-439493379-0w7hx
$ kubectl logs -f posts-439493379-hfzt1
```

Чтобы убедиться, что все в порядке, вы можете использовать команду `curl` для любой из конечных точек Posts, включая конечную точку `/healthz`. Вы, конечно же, увидите активность в журналах:

```
$ curl $(minikube service --url posts-svc)/posts?secret=newSecret
[
  {
    “id”: 7,
    “date”: “2019-02-17T05:42:51.000+0000”,
    “userId”: 2,
    “title”: “Chicken Pho”,
    “body”: “This is my attempt to re-create what I ate in Vietnam...”
  },
  {
    “id”: 9,
    “date”: “2019-02-17T05:42:51.000+0000”,
    “userId”: 1,
    “title”: “Whole Orange Cake”,
    “body”: “That’s right, you blend up whole oranges, rind and all...”
  },
  {
```

```
    "id": 10,
    "date": "2019-02-17T05:42:51.000+0000",
        "userId": 1,
    "title": "German Dumplings (Kloesse)",
    "body": "Russet potatoes, flour (gluten free!) and more..."
  },
  {
    "id": 11,
    "date": "2019-02-17T05:42:51.000+0000",
    "userId": 3,
    "title": "French Press Lattes",
    "body": "We've figured out how to make these dairy free, but just as good!..."
  }
]
$ curl -i $(minikube service --url posts-svc)/healthz
HTTP/1.1 200
X-Application-Context: mycookbook
Content-Length: 0
Date: S un, 17 Feb 2019 06:13:34 GMT
```

Теперь, чтобы перевести один из ваших экземпляров службы Posts в состояние «Запущено и не отвечает», выполните приведенную ниже команду `curl`:

```
$ curl -i -X POST $(minikube service --url posts-svc)/infect
```

Следите за потоками журналов. В течение 5–10 секунд вы увидите строки, подобные приведенным ниже, в одном из двух потоков журналов, и сеанс потоковой передачи журналов прекратится; потоковая передача прекращается, когда отключается контейнер, к которому она подключена. Вот что делает конечная точка `infect` с булевым значением `isHealthy` в службе Posts:

```
... ConfigServletWebServerApplicationContext : Closing
➥ org.springframework.boot.web.servlet.context.AnnotationConfigServletWeb
➥ ServerApplicationContext@27c20538: startup date [Sun Feb 17 06:03:15 GMT
➥ 2019]; parent: org.springframework.context.annotation.AnnotationConfig
➥ ApplicationContext@2fc14f68
... o.s.j.e.a.AnnotationMBeanExporter : Unregistering JMX-exposed beans on shutdown
... o.s.j.e.a.AnnotationMBeanExporter : Unregistering JMX-exposed beans
... j.LocalContainerEntityManagerFactoryBean : Closing JPA
➥ EntityManagerFactory for persistence unit 'default'
```

Но отказ от старого контейнера – это еще не все, что сделал цикл управления в Kubernetes. Цикл также запустил новый экземпляр приложения в новом контейнере. Если вы начинаете потоковую передачу журналов из этого нового контейнера, что можно сделать снова, выполнив команду `kubectl logs` (обратите внимание, что поскольку Kubernetes перезапускает только контейнер, а не модуль, имя модуля не изменится), вы увидите, что приложение снова запущено и работает:

```
$ kubectl logs -f posts-5876ffd568-gr5bf
... s.c.a.AnnotationConfigApplicationContext : Refreshing org.
➥ springframework.context.annotation.AnnotationConfigApplicationContext
➥ @2fc14f68: startup date [Sun Feb 17 06:15:30 GMT 2019]; root of context
➥ hierarchy
... trationDelegate$BeanPostProcessorChecker : Bean 'configuration
```

```
➥ PropertiesRebinderAutoConfiguration' of type [org.springframework
➥ .cloud.autoconfigure.ConfigurationPropertiesRebinderAutoConfiguration
➥ $$EnhancerBySpringCGLIB$$3cd10333] is not eligible for getting
➥ processed by all BeanPostProcessors (for example: not eligible for
➥ auto-proxying)
. ____ _ __ _ _
/\\ / ___'_ __ _ _(_)_ __ __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
\\/ ___)| |_)| | | | | || (_| | ) ) ) )
' |____| .__|_| |_|_| |_\__, | / / / /
=========|_|==============|___/=/_/_/_/
:: Spring Boot :: (v2.0.6.RELEASE)
...
... o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s):
➥ 8080 (http) with context path ''
... c.c.c.config.CloudnativeApplication : Started
➥ CloudnativeApplication in 15.74 seconds (JVM running for 16.605)
```

Этот пример демонстрирует, что когда состояние приложения становится доступным, это позволяет такой системе, как Kubernetes, соответствующим образом реагировать, если с приложением происходят неизбежные изменения. Будучи разработчиком приложений для облачной среды, вы несете ответственность за создание соответствующей реализации модели, используемой в среде выполнения вашего приложения.

То, что я описала в этом разделе, представляет собой подход с использованием опроса, но я хотела бы отметить, что циклы управления, которые транслируют тактовые импульсы, являются еще одним способом реализации этого шаблона. Используя эту технику, компоненты постоянно транслируют состояние жизненного цикла с помощью цикла управления, а объекты, которых интересует состояние этого приложения, будут слушать эти события и реагировать соответствующим образом. Я напомню вам обсуждение в главе 4: то, что я здесь описываю, является событийно-ориентированным шаблоном. Будучи разработчиком, вы должны разбираться в архитектурных шаблонах программного обеспечения в целом и проектировать/внедрять их соответствующим образом. Ключом любого подхода является то, что циклы управления обеспечивают избыточность, которая компенсирует неопределенность, присущую распределенным системам.

7.7. Внесерверная обработка данных

Это не книга или даже не глава, посвященная внесерверной обработке данных. Но именно здесь, в этой главе, в которой рассказывается о жизненном цикле приложения, краткий обзор позволяет вам глубже понять некоторые элементы программного обеспечения для облачной среды. Как обычно подчеркивается в течение первых мгновений обсуждения этой темы, данное название не совсем правильное, потому что функции, выполняемые в этом стиле, абсолютно спокойно работают на серверах. Просто разработчик совершенно не заботится об этих деталях. Это делает система для внесерверной обработки данных.

Производительность разработчиков, безусловно, является одной из целей данного стиля, равно как эксплуатационная эффективность и финансовые аспекты,

поскольку большинство систем внесерверной обработки данных взимает плату за пользование только в течение времени, пока выполняется функция. Независимо от конкретных целей, я хочу сосредоточиться на платформе для внесерверной обработки данных; эта платформа очень облачная. Давайте начнем с рассмотрения модели на самом базовом уровне.

На самом базовом уровне внесерверная обработка данных имеет приложение, работающее только тогда, когда оно активно обрабатывает данные, чтобы дать ответ на событие. Если посмотреть на это с точки зрения жизненного цикла приложения: только когда поступает запрос, предоставляется среда выполнения, приложение развертывается и запускается, и обработка запроса завершается. И после запуска среда выполнения удаляется; см. рис. 7.15.

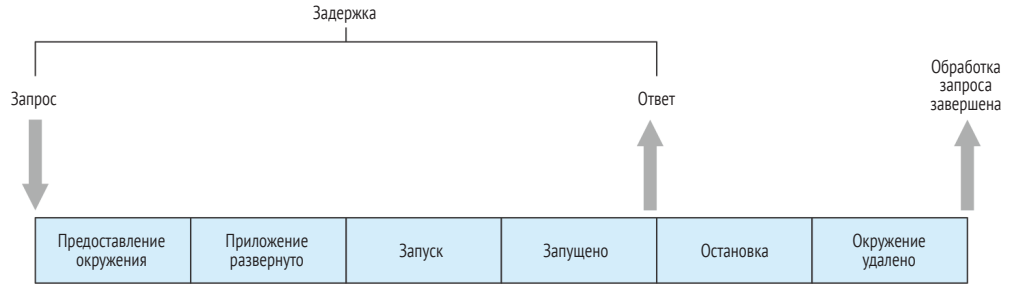


Рис. 7.15 ❖ Для вызова одной функции пройдены все этапы жизненного цикла

Данный жизненный цикл приложения не выглядит незнакомым. Нетрадиционным является тот факт, что все этапы, от инициализации до утилизации, происходят при каждом вызове. Что мне больше всего нравится во внесерверной обработке данных, так это то, что эта крайность служит для расширения шаблонов, используемых при работе с программным обеспечением для облачной среды. Например, если среда выполнения воссоздается полностью для каждого вызова, приложения ни при каких обстоятельствах не могут зависеть от внутреннего состояния предыдущих вызовов. До свидания, «липкие» сессии.

Но есть еще кое-что, на что я хочу обратить ваше внимание, когда мы говорим о приложениях для облачной среды, работающих в данном режиме. Это связано с эффективностью и задержкой. Глядя на рис. 7.15, становится очевидным, что между запросом на обработку и его завершением должно много чего произойти. Как все это может случиться при соблюдении требований к отзывчивости? Говоря коротко: посредством оптимизации, за которую вы частично несете ответственность.

На рис. 7.16 снова показаны этапы жизненного цикла внесерверной обработки данных, но на этот раз здесь присутствует ряд обозначений. Ранние этапы жизненного цикла приложения полностью обрабатываются системой, а платформы для внесерверной обработки данных специально ориентированы, среди прочего, на то, чтобы ускорить подготовку среды и развертывание приложений. Большинство построено на контейнерах и использует форматы развертываемых артефактов, которые обеспечивают быстрое развертывание. Будучи разработчиком, вы

несете ответственность и контролируете только запуск приложения и его фактическое выполнение. Вы должны сосредоточиться на том, чтобы это произошло настолько быстро, насколько это необходимо.

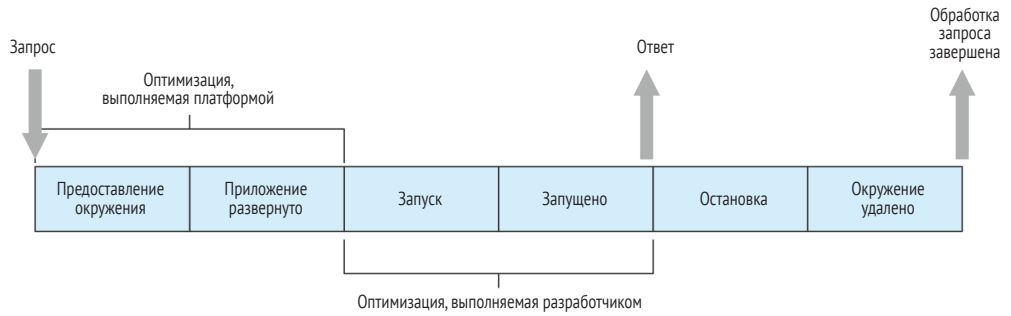


Рис. 7.16 ❖ Внесерверная обработка данных требует наличия платформы, которая оптимизирует ранние этапы, а разработчик отвечает за оптимизацию запуска и выполнения приложения

Скажу прямо: не вся ваша обработка выполняется лучшим образом в таких условиях. Если у вас есть рабочая нагрузка, у которой уходит мало времени на запуск, и она постоянно запрашивается, особенно если «расходы» на запуск превышают расходы на выполнение функции, вам, вероятно, лучше запустить один или несколько экземпляров приложения, чтобы запросы обслуживались с уже запущенных экземпляров. Если, с другой стороны, у вас есть обработка, которая запускается реже и требует гораздо больше времени, чем подготовка, развертывание приложения и его запуск, тогда парадигма внесерверной обработки данных может быть идеальной.

Если вы создаете приложение, которое будет работать в безсерверном контексте, вы должны уделить особое внимание затратам на запуск и убедиться, что они не превышают затраты на выполнение функций. У вас есть несколько рычагов, чтобы контролировать это. Во-первых, используемый вами язык программирования оказывает непосредственное влияние. Запуск виртуальной машины Java может занять десятки секунд, что было бы ужасно, если бы время выполнения кода составляло миллисекунды. И во-вторых, даже после того, как вы выберете язык, обязательно минимизируйте то, что вам нужно загрузить в среду выполнения для поддержки функциональности вашего приложения. Например, не включайте в свой код зависимости, которые никогда не используются. Вам придется заплатить цену за медленный запуск безо всякой причины.

Теперь я хочу отметить, что большинство платформ для внесерверной обработки данных на сегодняшнем рынке реализует оптимизацию, чтобы уменьшить влияние жизненного цикла приложения в его чистейшем виде. Например, окружения приложений часто сохраняются и используются повторно для запросов, которые приходят относительно близко друг к другу во времени, но здесь становится более очевидным, чем в других платформах, что ни одно приложение не должно использовать такие функции. Опять же, это одна из вещей, которые мне больше всего нравятся во внесерверной обработке данных: она проясняет необходимость использования шаблонов для облачной среды.

Резюме

- В облачной среде вы должны думать о жизненном цикле приложения и рассматривать его как единый логический объект, даже если каждый экземпляр приложения имеет свой собственный независимый жизненный цикл.
- Вы также должны внимательно следить за тем, как события жизненного цикла приложения влияют на другие приложения, которые образуют более широкую часть программного обеспечения.
- Последовательное обновление можно использовать только в том случае, если несколько экземпляров приложения может одновременно работать с разными конфигурациями. В противном случае необходимо использовать сине-зеленое развертывание. И то, и другое можно делать с нулевым временем простоя.
- Тщательно сконструированный шаблон периодической смены реквизитов доступа можно создать с помощью последовательных обновлений.
- Преднамеренная замена экземпляров приложения может помочь таким шаблонам, поэтому делайте это. Избавьтесь от предубеждений прошлого.
- Журналы приложений следует отправлять в потоки stdout и stderr, где большинство платформ для облачной среды будет их обрабатывать.
- Состояние приложения должно быть доступно, чтобы поддерживать работоспособность системы, а зависимые приложения могли соответствующим образом адаптироваться к изменениям.
- Внесерверная обработка данных – это крайняя форма обработки в облачной среде, в которой используется большинство описанных здесь шаблонов.

Глава 8

Доступ к приложениям: сервисы, маршрутизация и обнаружение сервисов

О чем идет речь в этой главе:

- отдельные сервисы, представляющие несколько экземпляров приложения;
- балансировка нагрузки на стороне сервера;
- балансировка нагрузки на стороне клиента;
- динамическая маршрутизация к экземплярам служб;
- обнаружение служб.

Я уже немного рассказывала о приложениях, развертываемых как несколько экземпляров, которым нужно было вести себя как единый логический объект. Вы узнали, что приложения не должны фиксировать состояние, чтобы один запрос к приложению не зависел от предыдущих запросов к тому же экземпляру. Вы видели, что необходимо тщательно управлять конфигурациями во всех экземплярах, чтобы обеспечить одинаковый результат независимо от того, какой экземпляр обслуживает определенный запрос, даже во время событий жизненного цикла приложения. На данный момент я хочу формализовать эту *единую логическую сущность*, и при этом вы сможете избавиться от хрупкой связи между тремя микросервисами в нашем приложении.

Потерпите немного, пока я начну с освещения первых важных принципов. Вы знаете, что ваши приложения будут развернуты в виде нескольких экземпляров, что даст вам эффективный способ масштабирования развертываний для удовлетворения спроса и создания более отказоустойчивой системы. На рис. 8.1 показано несколько экземпляров каждого из трех приложений, образующих наш работающий пример.

В предыдущих главах вы подключали эти приложения друг к другу, не вдаваясь в детали того, как это делается. Займемся этим сейчас. Для каждого набора экземпляров давайте введем поле, представляющее абстракцию *единого логического приложения*. На рис. 8.2 я более точно обозначила каждый фрагмент. Я пометила логические сущности именем приложения и более точно отметила экземпляры, как они есть, то есть в качестве экземпляров.

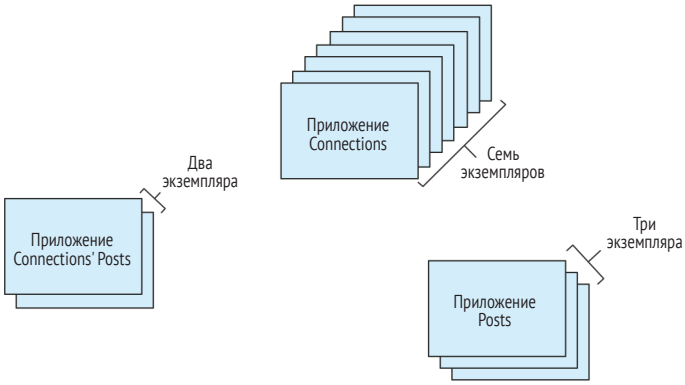


Рис. 8.1 ❖ В программном обеспечении для облачной среды приложения развертываются в виде нескольких экземпляров. Чтобы программное обеспечение функционировало предсказуемо, вам нужно, чтобы каждый набор экземпляров приложения работал как единая логическая сущность

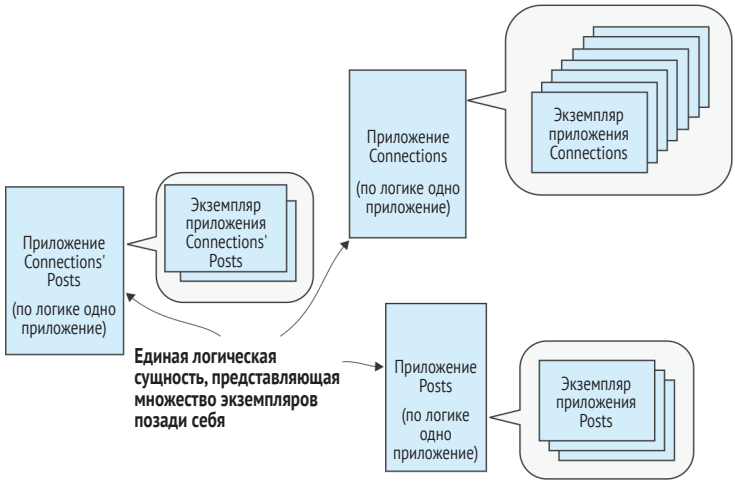


Рис. 8.2 ❖ Каждый набор экземпляров приложения представлен логической сущностью, которая определяет поведение приложения. Ожидается, что так будут вести себя все экземпляры

Затем вы можете перенести детали реализации каждого приложения в фоновый режим, как я это сделала на рис. 8.3 (всего лишь на мгновение – не волнуйтесь, мы скоро вернемся к деталям). Еще одна вещь, которую я сделала на этой диаграмме, – я пометила логические объекты как «Сервисы». Я уже применяла этот термин в другом контексте, но то, как я использую его здесь, полностью соответствует тому, как я использовала его до этого. Ранее я подразумевала под словом «сервис» компонент, который использовался кодом вашего приложения (например, база данных или шина передачи сообщений). Но помните, что приложения чаще всего являются программными компонентами, используемыми другими приложениями; таким образом, по сути, они являются сервисами.

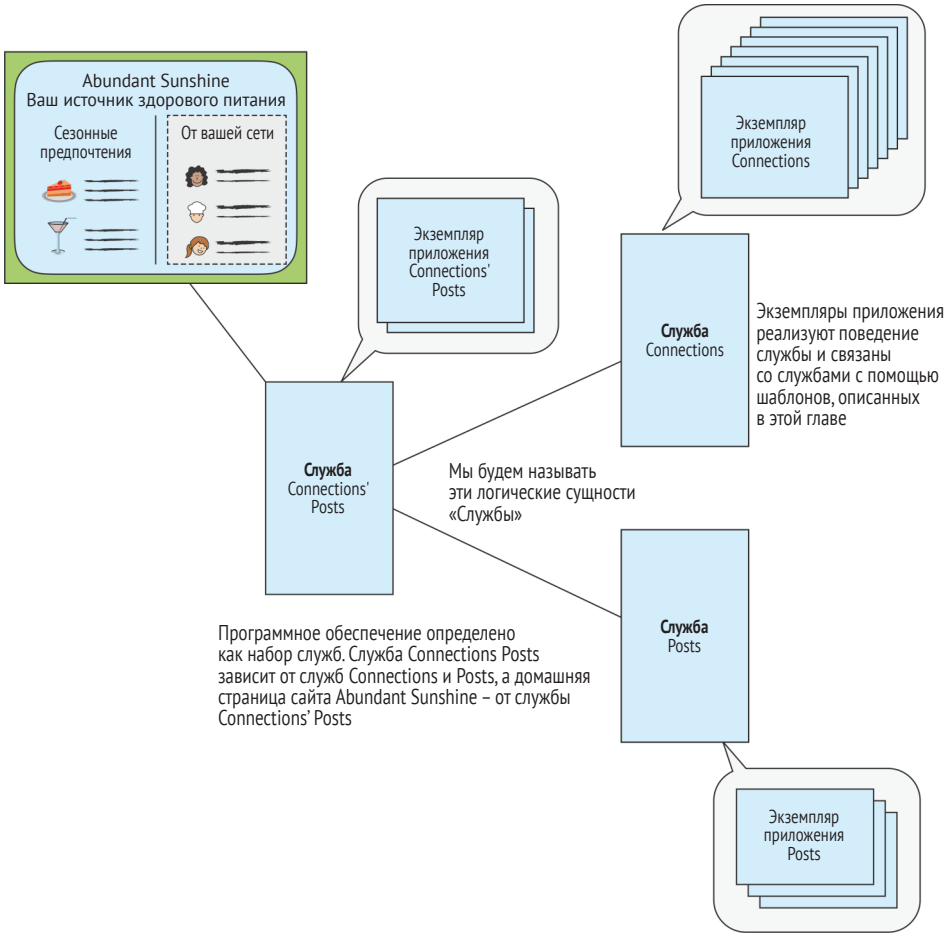


Рис. 8.3 ❖ Ваше программное обеспечение определено как набор служб. Каждая служба реализована как набор экземпляров служб

Если вы теперь сосредоточитесь на (логических приложениях) / службах, то можете определить свое программное обеспечение как набор связанных сервисов. Как вы знаете, служба Connections' Posts зависит и от службы Connections, и от службы Posts, как показано на рис. 8.3. Клиентом службы Connections' Posts в данном развертывании является веб-страница Abundant Sunshine.

Эта глава в основном посвящена данным сервисам, в частности двум аспектам, касающимся их. Во-первых, речь идет о том, как эти сервисы связаны с экземплярами приложений, которые они представляют (маршрутизация), а во-вторых, о том, как сервисы находят и обрабатывают их клиенты (обнаружение сервисов). Маршрутизация и обнаружение сервисов можно реализовать различными способами, и вы, будучи разработчиками программного обеспечения, должны разбираться в них, чтобы иметь возможность лучшим образом разрабатывать свое программное обеспечение.

На данный момент все это, возможно, выглядит несколько абстрактно, поэтому я начну главу со знакомых и конкретных примеров. Затем я подробно рассмотрю

тему маршрутизации. В случае с программным обеспечением для облачной среды это должна быть *динамическая* маршрутизация, средство, с помощью которого входящие запросы будут достигать постоянно меняющегося набора экземпляров приложения в сервисной абстракции. Мы рассмотрим как традиционную балансировку нагрузки, так и балансировку нагрузки на стороне клиента. В первом случае клиентские вызовы проходят через централизованный балансировщик нагрузки, который направляет запросы к экземплярам, а во втором случае балансировка нагрузки встроена в клиент. Затем я перейду к тому, как клиент службы находит и решит эту проблему, что, естественно, приведет нас к разговору о серверах имен и DNS. И тогда вы, наконец, сделаете это – исправите хрупкие конфигурации сервисов в нашем работающем приложении.

8.1. Сервисная абстракция

Легко говорить о сервисах на высоком уровне. Я делала это на протяжении более чем половины данной книги, но говорила об этом несколько расплывчато и хочу сейчас это исправить. Я хочу начать с простой ментальной модели, показанной на рис. 8.4.

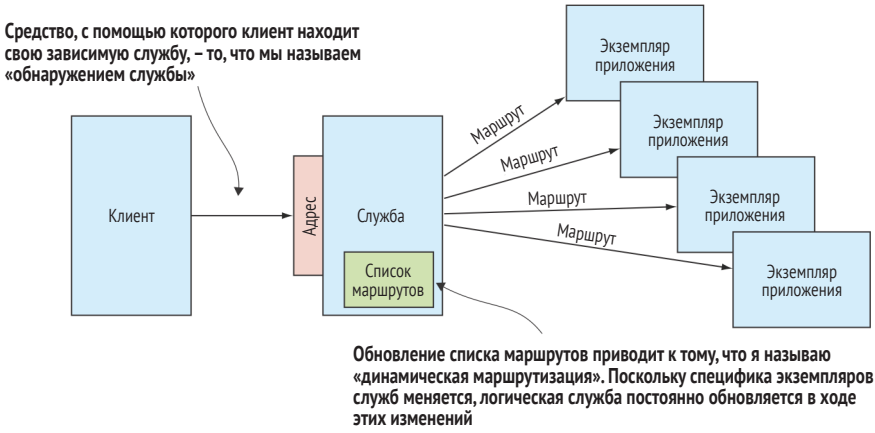


Рис. 8.4 ❖ Единственное логическое приложение, представленное набором экземпляров приложения, – это служба. Протокол, который позволяет клиенту находить и получать доступ к своим зависимым службам, – это обнаружение служб. Средства распределения входящих запросов через набор экземпляров приложения – это динамическая маршрутизация

На нем видно четыре *экземпляра приложения* и сервис, представляющий одно логическое приложение, которое реализовано этими экземплярами. С левой стороны сервиса находится клиент, который обращается к нему с помощью адреса сервиса. Средство, с помощью которого клиент находит этот сервис, называется *обнаружением сервисов*. Когда запрос поступает в сервис, он направляется в один из экземпляров. Как вы знаете, экземпляры сервисов будут постоянно меняться – иногда их будет два, иногда десять. Их IP-адреса также меняются. Для того чтобы

экземпляр точно представлял реализацию в данный момент, сервис должен быть в курсе списка экземпляров, куда он будет направлен. Это то, что я называю *динамической маршрутизацией*. В этой главе я неоднократно ссылаюсь на эту диаграмму, поэтому можете добавить данную страницу в закладки.

> **ПРИМЕЧАНИЕ** Некоторые конструктивные решения, касающиеся адресации, маршрутизации и обнаружения сервисов, будут приниматься во время развертывания программного обеспечения, а другие будут приниматься во время разработки программного обеспечения.

Как и практически все, о чем мы говорили до сих пор, шаблоны для работы с сервисами для облачной среды реализованы как в программном обеспечении, которое вы будете создавать, так и в платформах, на которых ваше программное обеспечение будет работать. В конечном счете ваша задача как разработчика заключается в том, чтобы гарантировать, что ваша реализация допускает различные варианты развертывания, или если вы встраиваете шаблон в саму реализацию, вам необходимо четко понимать последствия таких действий. Эта глава призвана помочь вам понять методы и компромиссы, чтобы вы могли сделать правильный выбор касательно работы с сервисами.

Чтобы начать разбираться в этих вариантах, давайте рассмотрим несколько примеров.

8.1.1. Пример сервиса: поиск в Google

Начнем с того, с чем вы полностью знакомы, хотя вы, вероятно, не думали об этом в контексте сервисов и их обнаружения. Давайте посмотрим, что происходит, когда вы что-то ищете в Google. Когда вы вводите **www.google.com** в адресную строку браузера, в данном случае клиента сервиса, то обращаетесь к службе поиска Google по имени. Однако запрос для главной страницы сайта Google отправляется на определенный IP-адрес, и преобразование имени в этот адрес осуществляется с помощью системы доменных имен (DNS).

Детали DNS, обеспечивающие работу интернета, сложны, и их реализация представляет собой сильно распределенную иерархическую систему с правилами распространения данных через нее. Для простоты воспользуемся командой `ping`, чтобы получить IP-адрес для имени **www.google.com**:

```
$ ping www.google.com
PING www.google.com (216.58.193.68): 56 data bytes
64 bytes from 216.58.193.68: icmp_seq=0 ttl=53 time=19.189 ms
```

На самом деле это имя может быть преобразовано в большое количество разных IP-адресов, но нам нужен только один. На рис. 8.5, который является более подробной версией рис. 8.4, теперь виден этот IP-адрес службы поиска. Когда клиент ссылается на службу с помощью имени, процесс обнаружения сервисов обращается к DNS для отображения имени в IP-адрес, а затем обращается к сервису по этому адресу.

Справа от абстракции службы находятся маршруты, используемые для направления трафика к экземплярам, которые реализуют этот сервис. Несмотря на то что я никогда не работала в команде Google Site Reliability Engineering (SRE), разумно предположить, что на этом IP-адресе находится балансировщик нагрузки, кото-

рый распределяет входящий трафик по ряду экземпляров приложения главной страницы Google. Экземпляры этого приложения все время меняются, поэтому список маршрутов, которые содержит служба, должен постоянно обновляться, и это делает сама платформа Google[1]. На рис. 8.5 отмечена роль, которую играет Borg в процессе динамической маршрутизации.

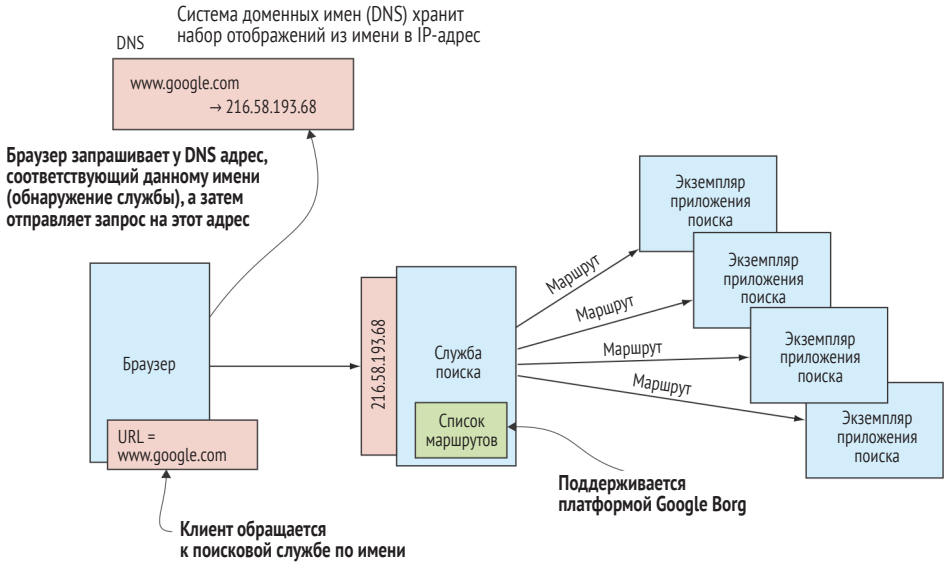


Рис. 8.5 ❖ Обращение к службе поиска Google выполняется через имя www.google.com, которое сопоставляется с конкретным IP-адресом через DNS. Платформа Google постоянно обновляет список маршрутов к экземплярам службы поиска и запросы на балансировку нагрузки между ними

В игру вступает базовый шаблон сервисов, чтобы продемонстрировать простую операцию, которую вы, вероятно, выполняете много раз в день. Вы используете имя, **www.google.com**, чтобы обратиться к службе. DNS используется как часть процесса обнаружения сервисов, отображая это имя в IP-адрес. Балансировщик нагрузки, список маршрутов которого обновляется самой платформой Google, маршрутизирует трафик к экземплярам приложения, которые реализуют этот сервис.

Давайте посмотрим на второй пример, который иллюстрирует две вещи. Во-первых, он показывает, что происходит, когда процесс обнаружения сервисов не используется, а во-вторых, он дает более полное представление об обслуживании динамических таблиц маршрутизации.

8.1.2. Пример сервиса: наш агрегатор блогов

Давайте рассмотрим наш работающий пример агрегатора блогов. В частности, мы будем рассматривать службу Posts, потому что у нас работает несколько экземп-

[1] Google написал об этой платформе в 2015 году: **http://mng.bz/pgv5**.

ляров приложения и, следовательно, нам нужна динамическая маршрутизация. Вы также можете посмотреть на клиента сервиса, в частности на приложение Connections'Posts. Как и в примере с Google, вы увидите правую часть (динамическая маршрутизация) и левую (адресация и обнаружение сервисов) абстракции службы. На рис. 8.6 снова дается более подробная версия рис. 8.4.

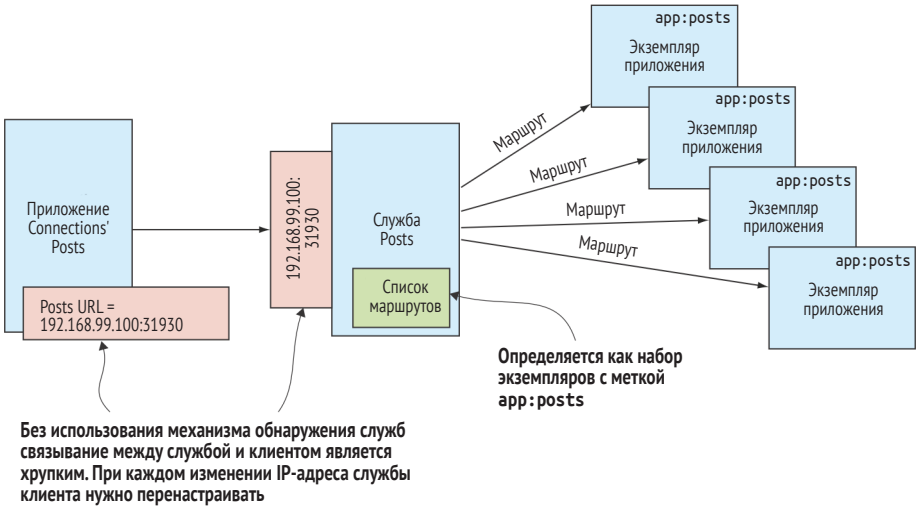


Рис. 8.6 ❖ До сих пор в конфигурации вашего программного обеспечения для агрегации блогов не использовался какой-либо протокол обнаружения службы, и служба Connections 'Posts связывалась со службой Posts через IP-адрес. Это приводит к хрупкому развертыванию

Служба Connections' Posts, которая является клиентом службы Posts, имеет IP-адрес для службы Posts. Посмотрев на переменные среды, определенные в файле cookbook-deploy-connectionposts.yaml, можно увидеть, что URL-адрес службы Posts выглядит примерно так: `http://192.168.99.100:31930/posts?userIds=`. Вы, наверное, помните, что каждый раз, когда вы заново создавали *службу* Posts, вам приходилось перенастраивать приложение Connections' Posts, указывая в этом URL-адресе новую комбинацию IP-адреса и порта. Вы были вынуждены делать это, потому что не использовали обнаружение сервисов. В конце этой главы мы это исправим. На рис. 8.6 видно, что службе назначен этот IP-адрес, и это значение также жестко закодировано в клиенте сервиса.

Справа от сервиса находятся маршруты. У Kubernetes есть простой и изящный способ определить, какие экземпляры приложений используются в сервисе: теги и селекторы. Каждый экземпляр службы Posts помечается парой типа «ключ/значение» `app:posts`. Сервис определяется с помощью селектора, устанавливающего, что эта служба представляет список экземпляров приложения с тегом `app:posts`. Это изображено в правой части рис. 8.6. Сам Kubernetes реализует процессы, которые постоянно обновляют список маршрутов, поэтому при каждом обращении к службе запрос будет перенаправляться в один из текущих экземпляров.

Хотя я надеюсь, что два этих примера помогли объяснить ключевые фрагменты вашей ментальной модели сервисов, я также надеюсь, что вы по-прежнему ищете большего. Пока я еще не рассматривала никаких конкретных реализаций и мало что говорила о компромиссах, на которые ссылалась. Давайте копнем глубже, начав с правой стороны нашей абстракции сервисов.

8.2. Динамическая маршрутизация

Поговорим о двух элементах в правой части абстракции: это средства, с помощью которых обновляется список экземпляров приложения, и маршрутизация трафика к этим экземплярам службы. Я называю последнее *балансировкой нагрузки* и хочу рассказать о двух подходах: балансировке нагрузки на стороне сервера и балансировке нагрузки на стороне клиента.

8.2.1. Балансировка нагрузки на стороне сервера

Наверняка, поскольку это наиболее распространенная реализация, балансировка нагрузки на стороне сервера вам знакома. В развертывании, где реализуется этот шаблон, есть компонент, который выполняет операцию приема входящих запросов и отправляет эти запросы одному из соответствующих экземпляров. В своей клиентской базе я обычно вижу как аппаратные, так и программные балансировщики нагрузки (включая F5, nginx и продукты от уже зарекомендовавших себя сетевых компаний, таких как Cisco и Citrix), а также сервисы балансировки нагрузки от всех крупных облачных провайдеров (таких как Google, Amazon и Microsoft).

Балансировка нагрузки обычно выполняется на уровне протокола TCP/UDP или на уровне протокола HTTP. Способ, с помощью которого балансировщик нагрузки выбирает экземпляр для маршрутизации, будет варьироваться; например, круговой или случайный. Детали этих алгоритмов выбора не должны вас беспокоить. Я намеренно не буду описывать здесь подробности, потому что в случае с ПО для облачной среды вы абсолютно не должны зависеть ни от одной из этих особенностей. Равно как ваши приложения не должны создаваться таким образом, чтобы обработка одного запроса зависела от предыдущего запроса, достигшего такого же экземпляра ранее, так же и вы не должны зависеть от циклирования экземпляров в каком-либо конкретном порядке. А поскольку балансировщики нагрузки часто позволяют вам включать привязку сессий, также известную как *липкие сессии*, должна повторить, что вы не должны этого делать. Приложения, которые зависят от «липких» сессий, не предназначены для облачной среды.

Централизованная балансировка нагрузки имеет ряд преимуществ:

- это зрелая технология. Реализации балансировки нагрузки, о которых я упоминала здесь, развивались в течение нескольких десятилетий и являются надежными;
- о централизованной реализации часто легче рассуждать, чем о сильно распределенной;
- конфигурировать один централизованный объект часто проще, чем сильно распределенный.

С другой стороны, один объект может представлять единственную точку отказа в системе, но в действительности балансировщики нагрузки на стороне сервера

почти всегда развертываются как кластер, как для масштабирования, так и для обеспечения устойчивости. На рис. 8.7 показан клиентский запрос, проходящий через балансировщик нагрузки на стороне сервера (изображенный в виде кластера), который распределяет нагрузку по всем экземплярам службы.

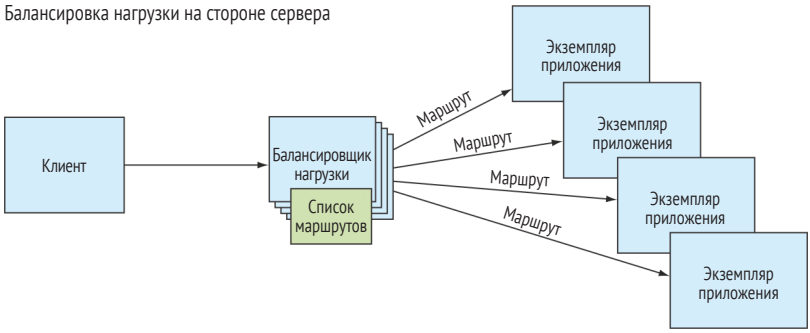


Рис. 8.7 ❖ При централизованной балансировке нагрузки или нагрузке на стороне сервера клиентский запрос обрабатывается кластером балансировщиков нагрузки, у которого есть список маршрутов к экземплярам службы. Балансировщик нагрузки распределяет клиентские запросы по разным экземплярам приложения

8.2.2. Балансировка нагрузки на стороне клиента

Если вы доведете идею кластера балансировщиков нагрузки до предела, то можете распределить компоненты балансировки нагрузки настолько широко, что они будут включены в самих клиентов, как показано на рис. 8.8. Обратите внимание, что каждый экземпляр клиента имеет собственную встроенную функцию балансировки нагрузки.

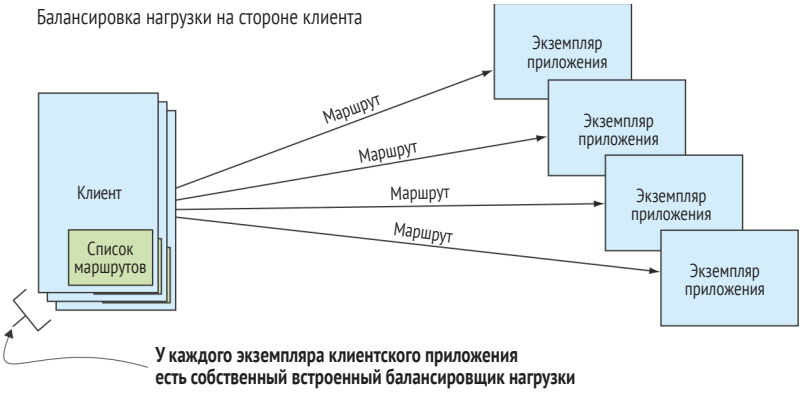


Рис. 8.8 ❖ При балансировке нагрузки на стороне клиента клиент отправляет запросы непосредственно экземплярам службы и выполняет задачу по распределению запросов по всем экземплярам. Список маршрутов к экземплярам служб поддерживается внутри самого клиента

Балансировка нагрузки на стороне клиента приобрела популярность, потому что количество микросервисов, составляющих наши программные реализации, резко возросло, и, соответственно, увеличилось количество сетевых запросов, проходящих через систему. Использование балансировщика нагрузки в клиенте эффективно устраняет один скачок во всей сети, и в широких масштабах это может существенно повлиять на производительность. Сравните рис. 8.7 и 8.8, и вы увидите, что в последнем клиент обращается к экземплярам службы напрямую.

Для балансировки нагрузки на стороне клиента вы почти всегда используете фреймворк, который либо помещает библиотеки в двоичный файл вашего приложения (как Netflix Ribbon, **https://github.com/Netflix/ribbon**), либо использует другие методы, такие как сайдкары (как Istio, **https://istio.io/**). В любом случае, прежде чем прийти к выводу, что вам всегда нужно оптимизировать производительность и, следовательно, вы всегда будете использовать балансировку нагрузки на стороне клиента, рассмотрим следующие последствия:

- если вы встраиваете библиотеки в свой код, для обновления фреймворка потребуется выполнить повторную сборку своих приложений;
- настройка функциональности балансировки нагрузки может быть более сложной;
- вам нужно будет узнать детали того, как использовать конкретную возможность балансировки нагрузки на стороне клиента, которую вы выбираете. Скорее всего, вы уже хорошо разбираетесь в том, как делать запросы по протоколу TCP или HTTP от своего клиента, используя хорошо протестированные и широко распространенные библиотеки. Теперь вы изучаете новый протокол;
- самое главное, вы будете ограничивать варианты развертывания своих приложений. Например, выбрав Ribbon, вы значительно усложняете использование балансировщика нагрузки на стороне сервера, в котором могут применяться корпоративные политики.

Будет ли ваше программное обеспечение использовать балансировку нагрузки на стороне клиента или на стороне сервера – это, вероятно, решение, на которое влияют корпоративные стандарты, и, безусловно, это будет сильно зависеть от архитектурных принципов, согласованных в командах разработчиков. Независимо от того, используете ли вы балансировку нагрузки на стороне клиента или на стороне сервера, вам нужно изучить еще один шаблон, даже если ваша платформа реализует его за вас: как обновляется список экземпляров службы для функции маршрутизации.

8.2.3. Свежесть маршрутов

На первый взгляд, поддержание этого списка маршрутов в актуальном состоянии кажется простым: когда создается новый экземпляр, вам нужно добавить его адрес в список, а когда экземпляр удаляется, вы должны удалить его из списка. Но нечто настолько простое по своей концепции становится сложным в сильно распределенной, постоянно меняющейся среде. Оглядываясь назад на обсуждение в предыдущей главе, посвященной жизненному циклу приложения, напомню, что мы уже рассматривали некоторые крайние случаи, например тот, в котором приложению отказано в возможности объявить о своем предстоящем выходе из системы до того, как произойдет внезапный аварийный сбой. Если бы точность

таблицы маршрутизации зависела от таких крайних случаев, то ваша система работала бы плохо. В основе правильно функционирующей системы лежит цикл управления (да, еще один цикл управления!), задачей которого является постоянная оценка фактического состояния развертывания и обеспечение отражения этой реальности в таблицах маршрутизации. Ваша платформа для облачной среды обеспечит основные возможности, от которых это зависит, а ваша задача – гарантировать, что ваше приложение предоставляет информацию, необходимую для правильной работы платформы. Когда речь идет о свежести маршрута, это означает две вещи: (1) предоставление информации, чтобы платформа могла построить точную модель фактического состояния системы, и (2) предоставление средства, с помощью которого идентифицируются экземпляры, реализующие сервис.

Первая часть этого уже была рассмотрена в главе 7. Вы несете ответственность за реализацию конечных точек, которые платформа может использовать для оценки работоспособности приложения. Напомню, что мы настроили Kubernetes, свою облачную платформу, для реализации проверок этих конечных точек работоспособности, и они используются для построения этой модели состояния системы.

Вторая часть поддержания свежести маршрутов – это средство, с помощью которого можно идентифицировать набор экземпляров приложения, которые должны быть в списке. Опять же, способ добиться этого зависит от платформы, а в Kubernetes это делается с помощью тегов и селекторов. Хотя я ранее не рассказывала об этом подробно, это все время использовалось при развертывании нашего программного обеспечения На рис. 8.9 показана часть манифеста развертывания для Posts, cookbook-deploy-posts.yaml.

Здесь видно, что Kubernetes надлежащим образом называет абстракцию сервиса `Service`, а часть определения – это селектор с тегом `app:posts`. Далее в определении экземпляра приложения вы видите, что экземпляры будут помечены метаданными, включая тег `app:posts`. Цикл управления, который поддерживает список маршрутов в актуальном состоянии, выполнит соответствующий запрос к модели фактического состояния системы и соответствующим образом обновит таблицы маршрутизации.

Хорошо, теперь у вас есть достаточно информации о динамической маршрутизации и балансировке нагрузки, чтобы я могла познакомить вас с конкретной реализацией правой части службы Posts. То, что я показываю здесь, относится к развертыванию на базе утилиты Minikube, которую мы уже использовали.

Если вы вернетесь обратно к рис. 8.6, заманчиво будет рассматривать изображенный там сервис как балансировщик нагрузки, но это всего лишь абстракция. В нашем развертывании на базе Minikube балансировщик нагрузки реализован с одним экземпляром компонента Kubernetes под названием Kube Proxy (Minikube – это развертывание, не предназначенное для рабочего окружения, поэтому допустимы отдельные точки отказа). Как видно из названия, Kube Proxy – это всего лишь прокси-сервер, который принимает входящие запросы и направляет их в соответствующие бэкенды. В Kubernetes каждому экземпляру приложения назначается собственный IP-адрес и, как вы только что видели мгновение назад, цикл управления постоянно запрашивает набор экземпляров для тех, у кого есть тег `app:posts`. Получающая в результате система показана на рис. 8.10. Чтобы было ясно, это реализация балансировки нагрузки на стороне сервера.

```
kind: Service
apiVersion: v1
metadata:
  name: posts-svc
spec:
  selector:
    app: posts
  ports:
  - protocol: "TCP"
    port: 80
    targetPort: 8080
  type: NodePort
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: posts
  labels:
    app: posts
spec:
  replicas: 2
  selector:
    matchLabels:
      app: posts
  template:
    metadata:
      labels:
        app: posts
    spec:
      containers:
      - name: posts
        image: cdavisafc/cloudnative-applifecycle-posts
```

Определяет (абстрактную) службу – сущность, которая представляет единую логическую службу Posts

Список экземпляров, которые представляет эта служба, задается с помощью селектора. Этот селектор обозначает все экземпляры приложения с тегом `app:posts`

Экземпляры приложения помечены тегом `app:posts`. Когда Kubernetes запускает экземпляр приложения, он прикрепляет к нему эти метаданные

Рис. 8.9 ❖ Манифест (абстрактной) службы Posts и экземпляров службы, которые ее реализуют. Список маршрутов экземпляров сервисов обновляется с помощью цикла управления, использующего селектор службы для поиска всех экземпляров приложений, которые соответствуют определенным критериям (в этом случае `app: posts`)

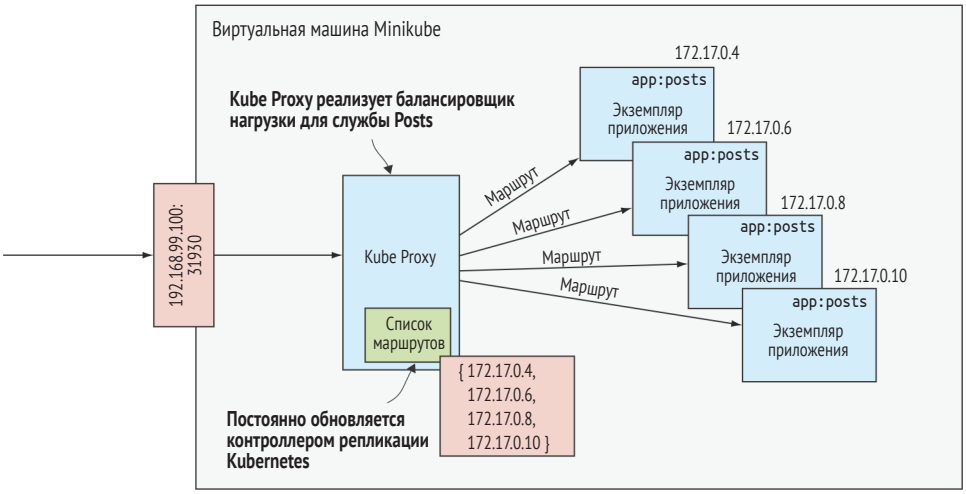


Рис. 8.10 ❖ Конкретная реализация службы Posts, работающей в Minikube. Kube Proxy является реализацией балансировки нагрузки и включает в себя список IP-адресов для всех экземпляров службы Posts

Рассмотрев правую часть абстракции сервиса, изображенной на рис. 8.4 (и многих последующих рисунках), давайте теперь обратимся к левой стороне, где показано, как осуществляется доступ к сервису. Когда вы создали службу Posts, Minikube динамически назначал порт, на котором прокси Kube слушает; IP-адрес виртуальной машины Minikube наряду с этим портом – это то, что вы использовали для доступа к сервису. Но это не надежно. Когда адрес службы изменяется, клиента нужно перенастроить, потому что вы не использовали процесс обнаружения служб в своем примере. Клиент может обращаться к сервису с помощью более подходящего способа посредством процесса обнаружения служб, который мы сейчас подробно рассмотрим.

8.3. Обнаружение служб

По сути, вам нужна простая абстракция, которая слабо связывает клиента из (изменяющегося) адреса службы, от которой он зависит. Это не сложно. Вам просто нужна служба именования. На рис. 8.11 изображен простой протокол, который позволяет клиенту обращаться к службе по имени; выполняется поиск адреса по этому имени, и устанавливается возможность подключения.

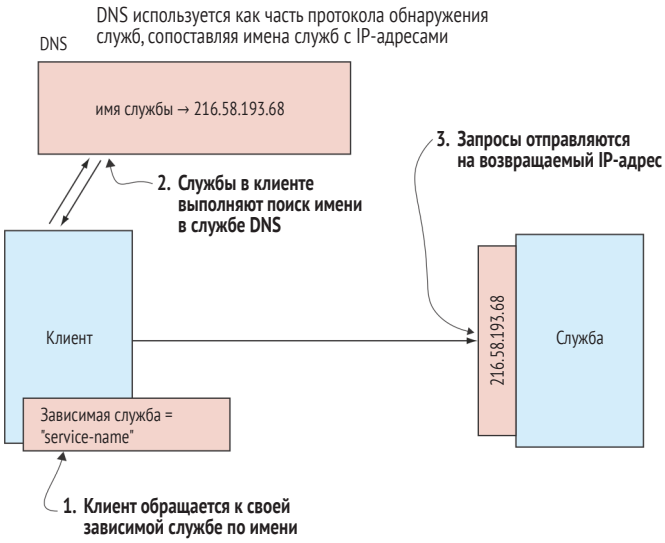


Рис. 8.11 ❖ Во время доступа к службе протокол обнаружения служб позволяет клиенту обращаться к службе по имени, что приводит к более устойчивой связи между ними

Есть два момента, чтобы заставить этот шаблон работать. Во-первых, должен быть способ размещения записей в службе имен. Во-вторых, должен быть способ получения адреса с указанием имени. Опять же, это звучит просто. Но как только вы начнете делать это в контексте распределенной системы, это будет не так просто. Хорошая новость состоит в том, что вам не нужно решать эту задачу самостоятельно. Систем именования предостаточно, и ваша задача – просто использовать их эффективно.

Но, прежде чем перейти к их использованию, позвольте мне рассказать о характеристиках системы именования в контексте ваших приложений для облачной среды. Вы уже достаточно хорошо понимаете тот факт, что в сильно распределенной топологии программного обеспечения наличие реплицированных фрагментов, которые работают независимо, имеет решающее значение. Имя службы само по себе – это реплицированная, распределенная система. Не вдаваясь в подробности теоремы CAP[1], скажу, что службы именования обычно настраиваются для обеспечения доступности по сравнению с согласованностью, и данный вариант подходит для этого конкретного случая использования. Чтобы понять, почему, давайте рассмотрим, что происходит, когда клиент обращается к сервису.

Преимущество доступности по сравнению с согласованностью означает, что когда клиент запрашивает адрес у службы именования, он всегда получает ответ, но этот ответ может быть устаревшим. Неправильные ответы будут даваться только тогда, когда сервис доступен по новому адресу или больше не доступен по старому адресу, но свежая информация еще не распространилась по всей системе; узел службы именования, отвечающий на вопрос клиента, несколько устарел. Но хорошо сконструированная система именования минимизирует окна, в которых может возникнуть эта несогласованность. Поскольку клиент ничего не может сделать для того, чтобы добраться до службы без преобразования имен, и поскольку большую часть времени ответ службы именования будет точным, полезно отдавать предпочтение доступности по сравнению с согласованностью. Но, так как могут возникнуть несоответствия, хотя это бывает редко, клиент системы разрешения имен должен учитывать эту возможность.

Будучи разработчиком клиентского приложения, вы несете ответственность за реализацию необходимого корректирующего поведения. Это не единственный случай, когда вам нужно создать реализацию, которая адаптируется к определенным несоответствиям, но хорошая новость заключается в том, что некоторые основные шаблоны, такие как повторная отправка запроса, которые вы подробно будете изучать в следующей главе, во многих случаях могут оказаться полезными вам. Итак, давайте добавим несколько основных попыток к вашему протоколу обнаружения сервисов.

Предположим, у вас есть служба, которая недавно пережила события жизненного цикла, в результате которых экземпляр был удален и создан новый. Когда клиент обращается к этой службе, он консультируется с DNS, чтобы получить IP-адрес, по которому доступна служба, но поскольку в данном случае мы отдаем предпочтение доступности по сравнению с согласованностью, DNS предоставляет в ответ IP-адрес уже не существующего экземпляра службы. Клиент пытается получить доступ к этой службе и, разумеется, не получает никакого ответа. Он может повторить запрос еще раз или два раза (и я подробнее расскажу о повторных попытках в следующей главе), но в итоге все равно потерпит неудачу. Это поведение показано в верхней части рис. 8.12.

---

[1] Теорема CAP, доказанная профессором Эриком Брюером, утверждает, что из трех свойств: согласованность, доступность и устойчивость к разделению (CAP) – только два могут быть реализованы в распределенном хранилище данных. См. **http://mng.bz/DVlV**.

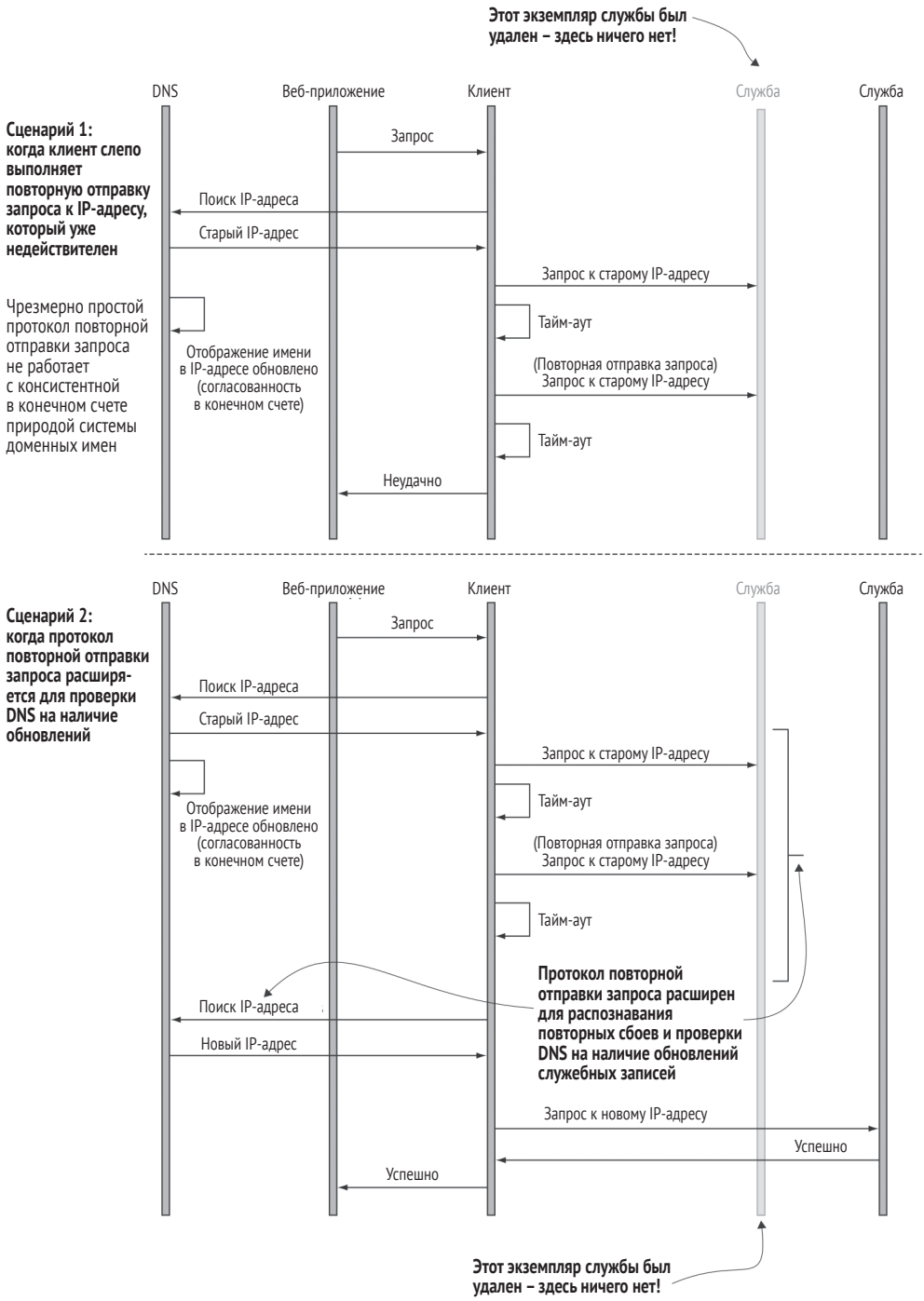


Рис. 8.12 ❖ Протокол обнаружения служб должен учитывать консистентный в конечном счете характер системы доменных имен

Зная, что DNS – это консистентная в конечном счете система, если вы немного отрегулируете это поведение, то получите гораздо лучшие результаты. После пары неудачных попыток вы можете снова запросить у DNS IP-адрес, и поскольку DNS была обновлена и теперь работает согласованно, вы получаете новый IP-адрес. Ваша попытка получить доступ к службе по новому адресу увенчалась успехом, и теперь ваш клиент может завершить свою работу. Этот протокол показан в нижней части рис. 8.12. В зависимости от фреймворка, который вы используете в коде клиента, вы можете или не можете нести явную ответственность за эту реализацию; фреймворк может прозрачно реализовать этот протокол за вас. Конечно, важно, чтобы вы четко понимали, несете ли вы личную ответственность за реализацию.

Но что произойдет, если на старом IP-адресе фактически есть служба, но это другая служба? Это гораздо более опасно. Короткий ответ на эту загадку заключается в том, что службы именования ни при каких обстоятельствах не должны использоваться для реализации безопасности. Реализация службы и/или развертывание должны использовать механизмы контроля доступа, чтобы запрещать неавторизованный доступ. При такой реализации клиентский доступ к устаревшему IP-адресу будет натыкаться на сообщение об ошибке, указывающее на то, что доступ запрещен, и клиент может ответить соответствующим образом. Знание того, что проблема управления доступом может быть результатом устаревшего IP-адреса, означает, что, будучи разработчиком клиента, вы можете выполнить проверку с помощью службы имен, чтобы убедиться, доступен ли обновленный IP-адрес, и, если это так, выполнить повторную попытку.

Это обсуждение крайних случаев, связанных с обнаружением сервисов, предвещает более глубокое обсуждение механизмов компенсации, которое появится в главах 9 и 10. Давайте пока оставим это и рассмотрим конкретные реализации шаблона обнаружения основных сервисов.

8.3.1. Обнаружение служб в сети

Мы уже рассматривали этот сценарий ранее в данной главе: что происходит, когда вы заходите на сайт **www.google.com** с помощью веб-браузера? Браузер реализует протокол доступа к службе имен (в данном случае DNS) на стороне клиента и затем отправляет запрос на этот адрес. Но как соответствующие записи попали в DNS? Вы знаете, что это делается путем явного внесения записей в реестр.

На рис. 8.13 показана консоль Cloud DNS, интерфейса DNS, который компания Google предоставляет в рамках Google Cloud Platform (GCP). Там вы можете увидеть записи, отображающие имена в IP-адреса. В этом случае они были созданы как часть установки Cloud Foundry на GCP.

В этом сценарии DNS-служба в интернете обеспечивает реализацию службы именования. Записи были помещены в DNS с помощью процесса развертывания программного обеспечения, и при доступе к pcf.kerman.cf-app.com ваш веб-браузер запрашивает DNS, чтобы получить IP-адрес 35.184.74.187 и приложение Cloud Foundry Operations Manager.

Record sets

Add record set | Delete record sets

DNS name	Type	TTL (seconds)	Data
kerman.cf-app.com.	NS	21600	ns-cloud-d1.googledomains.com. ns-cloud-d2.googledomains.com. ns-cloud-d3.googledomains.com. ns-cloud-d4.googledomains.com.
kerman.cf-app.com.	SOA	21600	ns-cloud-d1.googledomains.com.
*.apps.kerman.cf-app.com.	A	300	35.190.29.206
*.dev-k8s.kerman.cf-app.com.	A	300	35.202.105.107
pcf.kerman.cf-app.com.	A	300	35.184.74.187
*.pks.kerman.cf-app.com.	A	300	35.193.27.67
*.sys.kerman.cf-app.com.	A	300	35.190.29.206
doppler.sys.kerman.cf-app.com.	A	300	35.224.193.77
loggregator.sys.kerman.cf-app.com.	A	300	35.224.193.77
ssh.sys.kerman.cf-app.com.	A	300	35.202.74.34
tcp.kerman.cf-app.com.	A	300	35.225.64.210
*.ws.kerman.cf-app.com.	A	300	35.224.193.77

Рис. 8.13 ❖ DNS-записи,
которые отображают доменные имена в IP-адреса

8.3.2. Обнаружение сервисов с балансировкой нагрузки на стороне клиента

Как вы уже видели, обнаружение сервисов касается разрешения найти службу по определенному адресу без тесной связи этого адреса с реализацией клиента. Этот протокол используется как при балансировке нагрузки на стороне сервера, так и на стороне клиента, но существуют некоторые различия. Рисунок 8.14 проясняет это. Разница, если хотите, временная.

При балансировке нагрузки на стороне сервера преобразование имен обычно выполняется как часть вызова службы – после обращения к DNS запрос отправляется в службу. С другой стороны, при балансировке нагрузки на стороне клиента протокол обнаружения сервисов используется для обновления списка маршрутов, которые являются частью балансировщика нагрузки на стороне клиента. В некоторой степени мы смешали балансировку нагрузки и обнаружение сервисов, поэтому нужно рассмотреть это детально.

Я уже упоминала Netflix Ribbon и реализацию балансировщика нагрузки на стороне клиента. Этот фреймворк поддерживает модель программирования, в которой клиентский код может ссылаться на свой зависимый сервис через имя. Используя Spring Framework, это можно сделать с помощью аннотации класса, например:

```
@RibbonClient(name = «posts-service»)
```

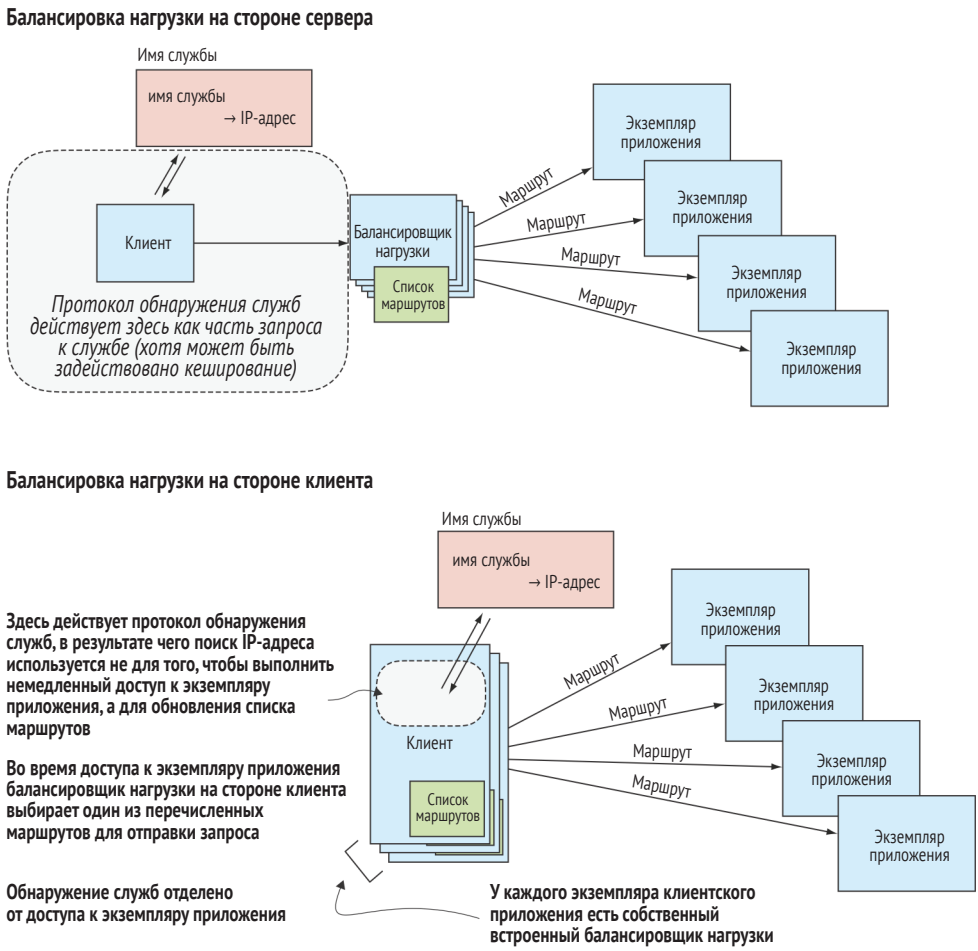


Рис. 8.14 ❖ Обнаружение служб зависит от того, используется балансировка нагрузки на стороне клиента или балансировка нагрузки на стороне сервера

Позже вы можете использовать это имя, например с помощью `restTemplate`, чтобы связаться со службой:

```
String posts
   = restTemplate.getForObject("http://posts-service/posts", String.class);
```

Поиск адреса и отправка запроса осуществляются с помощью комбинации Spring Framework и клиента Ribbon. Но обнаружение сервисов также зависит от регистрации пар типа «имя/адрес» в службе имен. Здесь, когда вы используете балансировку нагрузки на стороне клиента, вам нужен специальный сервис, чтобы облегчить этот процесс. Netflix Ribbon почти всегда используется в сочетании с другим сервисом, Netflix Eureka (**https://github.com/Netflix/eureka**), который применяется для обнаружения сервисов.

В этом случае нужно запустить службу Eureka; это служба именования, которая разрешает IP-адреса из заданного имени. Самый простой способ регистрации экземпляра службы в Eureka – это снова использовать Spring Framework. Любое приложение, которое включает в себя стартер Spring Boot для Eureka в пути к классам и имеет настроенные координаты в службе Eureka, будет автоматически зарегистрировано Spring Framework в Eureka. Это часть жизненного цикла приложения, которым за вас управляет Spring Framework.

8.3.3. Обнаружение сервисов в Kubernetes

Последний пример, который я хочу рассмотреть, устанавливает этап для добавления обнаружения сервисов в нашу реализацию, тем самым устраняя хрупкую конфигурацию, которая докучала нам в первых главах книги. Шаблон, конечно, такой же, как и в последних двух примерах: здесь есть некий тип службы имен, процесс для размещения записей в этой службе и еще один протокол для получения IP-адресов из имени.

Kubernetes предоставляет реализацию DNS-службы, которая называется – подождите – *CoreDNS*. (Ладно, говорить об этом было немного веселее, когда использовалось название Kube-DNS. В конце 2018 года Kube-DNS был официально заменен на DNS-сервер следующего поколения CoreDNS.) Хоть это и необязательный компонент, мне еще предстоит найти развертывание Kubernetes, которое не может установить его по умолчанию. Он развернут как приложение (в модуле) в вашем работающем кластере Kubernetes. Другие части платформы Kubernetes, а также элементы кода вашего приложения будут взаимодействовать с ней для выполнения регистрации и поиска, что образует протокол обнаружения сервисов. Можно убедиться, что CoreDNS работает, выполнив эту команду:

```
$ kubectl get pods --namespace=kube-system
NAME                               READY    STATUS    RESTARTS   AGE
coredns-86c58d9df4-8mfq8           1/1      Running   0          6d19h
coredns-86c58d9df4-sfqjm           1/1      Running   0          6d19h
etcd-minikube                      1/1      Running   0          6d19h
kube-addon-manager-minikube        1/1      Running   0          6d19h
kube-apiserver-minikube            1/1      Running   0          6d19h
kube-controller-manager-minikube   1/1      Running   0          6d19h
kube-proxy-jwcmg                   1/1      Running   0          16h
kube-scheduler-minikube            1/1      Running   0          6d19h
storage-provisioner                1/1      Running   0          6d19h
```

В этом выводе вы, вероятно, заметили, что CoreDNS работает как кластер из двух модулей.

Службы именования являются критически важным компонентом системы, составляющим ваше программное обеспечение, и поэтому должны быть развернуты с высокой степенью отказоустойчивости. Несколько экземпляров способствует этому.

Начиная с одной части протокола обнаружения сервисов, регистрация DNS выполняется Kubernetes автоматически при создании служб. Имена, которые Kubernetes регистрирует для конкретной службы, можно задать явно в манифесте развертывания, или значения по умолчанию могут быть получены из стандартных полей, таких как имя службы.

Для другой части протокола – поиска – подход прост. CoreDNS действует так же, как и любая другая DNS. Любая обработка, которая обычно взаимодействует со службой DNS (например, с помощью `restTemplate` для выполнения HTTP-запроса), выполняет такой поиск в CoreDNS. Kubernetes гарантирует, что адрес для CoreDNS настроен в работающие модули.

На рис. 8.15 все это собрано воедино:

1) в кластере Kubernetes находится служба DNS, которая носит название CoreDNS;
2) при запуске имя и адрес службы добавляются в службу CoreDNS;
3) у всех модулей (приложений), работающих в среде Kubernetes, есть адрес службы CoreDNS;
4) любые операции доступа к DNS, такие как отправка HTTP-запроса к URL-адресу, содержащему имя, осуществляют доступ к службе CoreDNS для разрешения адреса.

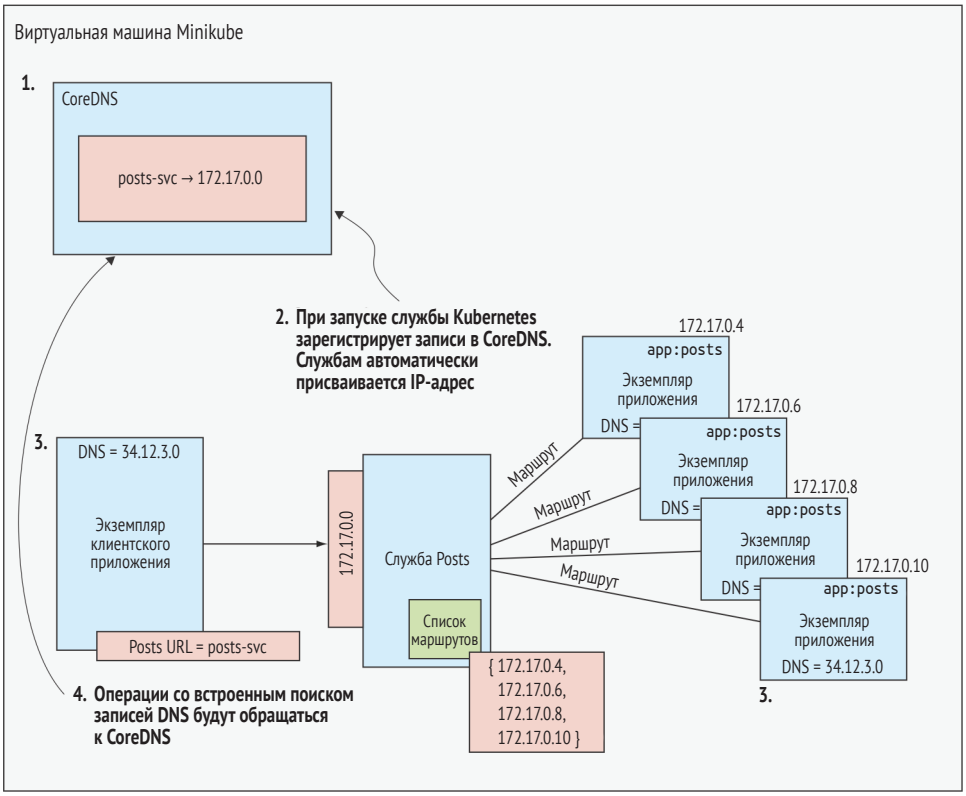


Рис. 8.15 ❖ Kubernetes обеспечивает реализацию протокола обнаружения служб с включением службы доменных имен и процессов, которые автоматически создают записи и получают доступ к ним в этом реестре

Теперь мы готовы применить эти новые знания в нашем примере агрегатора блогов.

8.3.4. Давайте посмотрим, как это работает: использование обнаружения сервисов

Пора! Теперь вы избавитесь от хрупкой конфигурации между различными службами, из которых состоит наше приложение. Когда вы закончите, службы больше не будут обращаться друг к другу через IP-адрес, и больше не нужно будет обновлять ранее хрупкую конфигурацию при изменении IP-адреса службы. Вместо этого вы будете использовать службу DNS для реализации протокола обнаружения сервисов. Или, лучше сказать, платформа, на которой вы развертываете приложения, в данном случае Kubernetes, реализует этот протокол за вас.

Настройка

На этом этапе я отсылаю вас к инструкциям по настройке для запуска примеров, приведенных в предыдущих главах книги. В этой главе нет новых требований для запуска образца.

Вы будете получать доступ к файлам в каталоге cloudnative-servicediscovery, поэтому в окне терминала перейдите в этот каталог.

Запуск приложений

Поскольку я внесла изменения в манифесты развертывания и больше не нужно выполнять какие-либо хрупкие шаги по настройке компонентов развертывания, я предлагаю вам удалить все развертывание своего приложения, используемого в качестве примера, целиком, включая базы данных и компоненты сервера конфигурации, а также все службы Kubernetes. Это позволит вам ясно увидеть, насколько проще становится развертывание благодаря добавлению автоматического обнаружения сервисов (вместо того чтобы внедрять протокол обнаружения сервисов вручную). Это можно сделать, запустив приведенный ниже скрипт:

```
$ ./ deleteDeploymentComplete.sh all
```

Для начала я отмечу, что этот каталог скуден. Он содержит только пару служебных скриптов и манифестов развертывания для нашего приложения. Там нет вообще никакого исходного кода, и это говорит о многом. Помните момент, на который я указала в начале главы? Я сказала, что проектные решения столь же вероятно будут проблемой времени развертывания, как и разработка, поскольку код, который осуществлял вызовы зависимых служб, например из приложения Connections' Posts в службу Posts, уже использовал методы, которые использовали службы DNS, заменяя нестабильные IP-адреса именами в манифестах развертывания приложения, где изменений кода не требуется. Манифесты развертывания указывают на образы Docker, которые я создала для главы 7.

Давайте начнем с развертывания двух служб баз данных и сервера конфигурации Spring Cloud; это делается с помощью этих трех команд:

```
kubectl apply -f mysql-deployment.yaml
kubectl apply -f redis-deployment.yaml
kubectl apply -f spring-cloud-config-server-deployment-kubernetes.yaml
```

Не забудьте заново создать базу данных поваренной книги:

```
$ mysql -h $(minikube service mysql-svc --format “{{.IP}}”) \
      -P $(minikube service mysql-svc --format “{{.Port}}”) -u root -p
mysql> create database cookbook;
```

Выполнив уже привычную команду `kubectl get all`, можно увидеть развертывание, службу и созданные в результате модули.

Теперь давайте посмотрим на изменения, которые я внесла в манифесты развертывания службы Connections:

- я обновила унифицированный идентификатор ресурса (URI) для службы MySQL, чтобы ссылаться на нее по имени; определение соответствующей переменной среды теперь выглядит так:

```
- name: SPRING_APPLICATION_JSON
  value: '{"spring":{"datasource":{"url": "jdbc:mysql://mysql-svc/cookbook"}}}'
```

- также видно, что к серверу Spring Cloud также обращаются по имени:

```
- name: SPRING_CLOUD_CONFIG_URI
  value: "http://sccs-svc:8888"
```

Теперь вы готовы запустить службу Connections с помощью этой команды:

```
kubectl apply -f cookbook-deployment-connections.yaml
```

Видны те же конфигурации для службы Posts, которую теперь можно запустить с помощью этой команды:

```
kubectl apply -f cookbook-deployment-posts.yaml
```

Наконец, в приведенном ниже листинге видно, что в манифесте развертывания Connections' Posts вы теперь обращаетесь к Redis, SCCS и к службам Posts и Connections по имени.

Листинг 8.1 ❖ Выдержка из файла cookbook-deploy-connectionsposts.yaml

```
- name: CONNECTIONPOSTSCONTROLLER_POSTSURL
  value: "http://posts-svc/posts?userIds="
- name: CONNECTIONPOSTSCONTROLLER_CONNECTIONSURL
  value: "http://connections-svc/connections/"
- name: CONNECTIONPOSTSCONTROLLER_USERSURL
  value: "http://connections-svc/users/"
- name: REDIS_HOSTNAME
  value: "redis-svc"
- name: REDIS_PORT
  value: "6379"
- name: SPRING_APPLICATION_NAME
  value: "mycookbook"
- name: SPRING_CLOUD_CONFIG_URI
  value: «http://sccs-svc:8888»
```

Вы можете запустить этот сервис с помощью такой команды:

```
kubectl apply -f cookbook-deployment-connectionsposts.yaml
```

Вы заметили, что вам не нужно было редактировать ни один из манифестов развертывания? Ах, прелесть слабой связанности посредством обнаружения сервисов.

Хочу обратить ваше внимание еще на два момента в конфигурации службы Connections' Posts.

Во-первых, имена, используемые для обращения к службам Posts и Connections, не сопровождаются каким-либо номером порта. Вы, возможно, заметили, что

предыдущие конфигурации показали, что обе службы слушали на одном и том же IP-адресе (это адрес вашей виртуальной машины Minikube), но на разных портах. Когда вы заменили URI (унифицированный идентификатор ресурса) на имя службы, вы не только устранили хрупкую привязку к IP-адресу, но также изменили что-то в способе маршрутизации трафика. Когда использовался IP-адрес, трафик шел к службам Posts или Connections по пути север/юг. Ваш запрос вышел за пределы окружения Kubernetes и повторно поступил через IP-адрес виртуальной машины Minikube. При замене IP-адреса и порта на имя службы маршрутизация из Connections' Posts к Posts оставалась в окружении Kubernetes, используя маршрутизацию восток/запад. Кроме того, когда Kubernetes создает объект службы, он назначает внутренний IP-адрес этому объекту, и это тот самый IP-адрес, который ассоциирован с именем в CoreDNS.

Второе, на что я хочу обратить ваше внимание, связано с первым пунктом. Обратите внимание, что номер порта службы Redis теперь установлен на `6379`. В предыдущих конфигурациях доступ к службе Redis осуществлялся через маршрутизацию север/юг, как и для служб Posts and Connections. Но при изменении имени хоста Redis на `redis-svc`, зарегистрированное в DNS, используется маршрутизация восток/запад, и трафик будет отправляться напрямую к `redis-svc`. Посмотрев на определение службы Redis, вы увидите, что она настроена на прослушивание на порту 6379, и запросы, поступающие на этот порт, будут передаваться на `targetPort`, на котором слушает модуль, где работает фактическая служба Redis.

Листинг 8.2 ❖ Выдержка из файла redis-deploy.yam

```
kind: Service
apiVersion: v1
metadata:
  name: redis-svc
spec:
  selector:
    app: redis
  ports:
  - protocol: “TCP”
    port: 6379
    targetPort: 6379
  type: NodePort
```

Теперь у вас есть полностью работоспособный образец приложения, с одним важным отличием от предыдущего: вы можете удалить службы Posts или Connections и воссоздать их, а клиент этих служб, служба Connections' Posts, не требует повторной конфигурации или даже развертывания. Поскольку доступ к зависимым службам становится проще с помощью протокола обнаружения сервисов, развертывание программного обеспечения для облачной среды терпимо к таким изменениям.

Адаптация к постоянным изменениям при развертывании ПО для облачной среды была бы неразрешимой без помощи платформы, которая обеспечивает такие вещи, как проверка работоспособности и свежесть маршрутов. Обнаружение сервисов является столь же важным протоколом для использования. Ваша работа заключается в том, чтобы создавать код своего приложения и развертывания таким образом, чтобы эти платформы могли предоставлять такие сервисы вашему программному обеспечению.

Резюме

- Простая абстракция может использоваться для более слабого связывания клиентов с зависимыми службами.
- Доступно два основных подхода использования балансировки нагрузки – централизованный (или на стороне сервера) и на стороне клиента. У каждого из них есть свои преимущества и недостатки.
- Конфигурация балансировщиков нагрузки должна быть динамичной и в высокой степени автоматизированной, поскольку в облачной среде экземпляры, к которым направляется трафик, меняются гораздо чаще, чем это было в прошлом.
- Службы именования, такие как DNS, играют важную роль в протоколе обнаружения сервисов, который позволяет клиентам находить зависимые службы даже в постоянно меняющейся топологии.
- При использовании службы доменных имен вы как разработчик должны учитывать тот факт, что таблицы преобразования имен в IP-адреса консистентны в конечном счете и должны учитывать записи, которые могут быть устаревшими.
- Использование протокола обнаружения сервисов обеспечивает гораздо более устойчивое развертывание программного обеспечения.

Глава 9

Избыточность взаимодействия: повторная отправка запроса и другие циклы управления

О чем идет речь в этой главе:

- повторная отправка запроса: повторение попыток доступа по тайм-ауту;
- шквал повторных отправок;
- безопасные и идемпотентные службы;
- возврат к предыдущей рабочей версии;
- циклы управления.

Что вы делаете во время просмотра сайтов, если страница, к которой вы пытаетесь получить доступ, не загружается? Вы нажимаете кнопку **Обновить**, верно? Я много говорила об избыточных экземплярах сервисов, но теперь хочу обратиться к другому месту, где избыточность используется в программном обеспечении для облачной среды: при выполнении запросов. Точно так же, как нельзя зависеть от одного экземпляра приложения, чтобы всегда быть работоспособным, невозможно зависеть от каждого запроса, чтобы никогда не испытывать никаких проблем. Вместо этого ваше программное обеспечение будет повторять запросы, равно как и вы. Ну, может быть, не *так, как* это делаете вы. Давайте рассмотрим эту тему.

Я начну с самого простого случая: вы загружаете страницу, чтобы прочитать ее. Например, возможно, вы просматриваете домашнюю страницу Hacker News (**https://news.ycombinator.com/**), заголовок «Монахи, играющие панк-рок (2007)» привлекает ваше внимание, и вы нажимаете на ссылку, чтобы прочитать статью полностью. Статья не загружается или загружается только частично, поэтому вы нажимаете кнопку **Обновить**, и все в порядке.

Но если у вас возникла ошибка при загрузке сразу после нажатия кнопки **Разместить заказ** на сайте вашего любимого электронного магазина, вряд ли вы просто нажмете эту кнопку еще раз. Сначала вы убедитесь, прошла ли покупка, возможно, проверив корзину, выполнив доступ к списку открытых заказов или, вероятно, даже проверив, получили ли вы электронное письмо с подтверждением заказа. Если вы найдете доказательства того, что заказ был размещен, все в по-

рядке – не нужно повторять частично неудавшийся запрос. Если, с другой стороны, вы достаточно уверены в том, что заказ не был размещен, то вернетесь и повторите запрос, чтобы совершить покупку.

Позвольте мне также обратить ваше внимание еще на одну функцию, с которой вы, вероятно, сталкивались на некоторых посещаемых вами веб-сайтах: флажок «Я не робот» или капчу. Данный тип виджета обычно используется, чтобы наложить ограничения на определенные взаимодействия с веб-сайтом, дабы не дать ботам создавать учетные записи (в большом количестве) или для предотвращения взлома паролей, например. Эта функция в основном относится к повторным отправкам запроса и выделяет еще один аспект избыточности запросов, о котором я расскажу здесь: при переходе от обычных пользователей к клиентским машинам вам необходимо знать об увеличении порядков величин как объема, так и частоты запросов.

Эти знакомые сценарии являются хорошим местом, чтобы приступить к изучению избыточных взаимодействий, но важно понимать, что и действующие лица, и контекст – разные. В архитектуре программного обеспечения для облачной среды клиент и служба взаимодействия – это программы. Программно-реализуемый клиент должен принимать решения, аналогично тому, как это делаете вы, будучи человеком, – например, сколько нужно ждать, прежде чем отказаться от запроса. Он также должен понимать, когда не следует предпринимать повторную отправку запроса, и должен осознавать свою силу.

Возможно, вы заметили, что я использую слово «взаимодействие», и, надеюсь, это возвращает вас к ментальной модели программного обеспечения для облачной среды, которую я создала в первой главе. Взаимодействие является одной из основных сущностей, с которой я вас познакомила ранее. И хотя глава 8 затронула эту тему, речь шла прежде всего о том, что необходимо, прежде чем установить взаимодействие, например как клиент находит зависимую службу. Теперь мы приступаем к серьезному рассмотрению взаимодействия (см. рис. 9.1), вначале делая акцент на клиентской стороне этого взаимодействия. Глава 10 посвящена завершению службы взаимодействия.

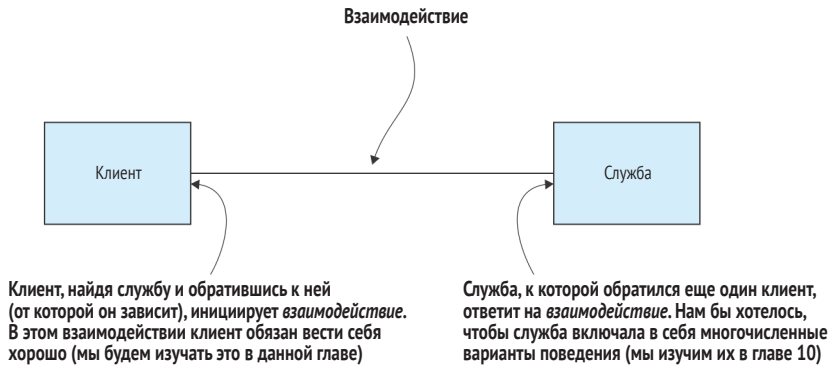


Рис. 9.1 ❖ Определенные шаблоны проектирования, применяемые с обеих сторон взаимодействия, приведут к созданию гораздо более прочных и надежных систем. В этой главе мы будем рассматривать шаблоны на стороне клиента, а в следующей – шаблоны на стороне сервиса

Я начинаю главу с добавления простой реализации повторной отправки запроса для нашего приложения, используемого в качестве примера, и мы проведем несколько экспериментов, чтобы продемонстрировать ценность этого шаблона. Затем я доведу эти эксперименты до предела, чтобы вы увидели, что происходит, когда большие объемы повторных отправок оказывают негативные последствия, которые распространяются по всей системе. Потом мы рассмотрим методы защиты от шквалов повторных отправок. В конце концов, повторная отправка запроса – всего лишь один из примеров повторяющихся действий, и в завершение главы я расскажу о распространенном шаблоне циклов управления и их важной роли в программном обеспечении для облачной среды.

9.1. Повторная отправка запроса

Программное обеспечение для облачной среды, по определению, является распределенной системой. В прошлом вызов функции из другой части вашего кода был просто вызовом метода, и все выполнялось в одном и том же процессе. Сегодня ваши реализации заполнены запросами, которые передаются по сети – сети, которая не всегда надежна. И даже когда с сетью все в порядке, нет никаких гарантий, что пока ваш процесс запущен и работает, служба, которую вы вызываете, в равной степени исправна. Именно эти атрибуты распределенных систем определяют необходимость обеспечения устойчивости запросов.

Позвольте мне прояснить кое-что в самом начале. Существует несколько способов определения, или обеспечения, *устойчивости запросов*. Более традиционный подход может сосредоточиться на *долговечности запросов* – это значит придумать способы гарантировать, что запросы никогда не будут теряться. Но это аналог традиционного подхода к усилению защиты серверов и устройств хранения: делать их все сильнее и сильнее во избежание неприятностей. Вместо этого более современная парадигма, которая лежит в основе всего, о чем идет речь в этой книге, признает, что компоненты *все же* не сработают как надо и что мы достигнем устойчивости, адаптируясь к этому неизбежному нарушению. Вот о чем эта глава. В ней исследуется, как достичь устойчивости за счет избыточности запросов, вместо того чтобы рассматривать каждый запрос как нечто, что просто невозможно потерять. Итак, в следующих главах представлены подходы, позволяющие сохранять запросы, но с одной изюминкой – я оставлю это на потом.

9.1.1. Основной шаблон

Основной шаблон прост: ваше приложение отправит запрос удаленному сервису, и если оно не получит ответ в течение разумного периода времени, то попытается сделать это снова. К настоящему времени вы знакомы с нашим примером – агрегатором блогов, – где служба Connections' Posts выполняет вызовы к службам Connections и Posts, а затем возвращает агрегированный результат.

Демонстрация, которую мы сейчас рассмотрим, фокусируется на службе Connections' Posts в качестве клиента, который отправляет HTTP-запрос в службу Posts (она по-прежнему выполняет запросы к службе Connections, но в этом упражнении мы сосредоточимся на взаимодействии между Connections' Posts и Posts). В данном примере мы реализовали повторную отправку этого запроса (рис. 9.2),

поэтому теперь, когда служба Connections' Posts выполняет вызов к службе Posts и не получает ответ, она просто повторяет запрос.

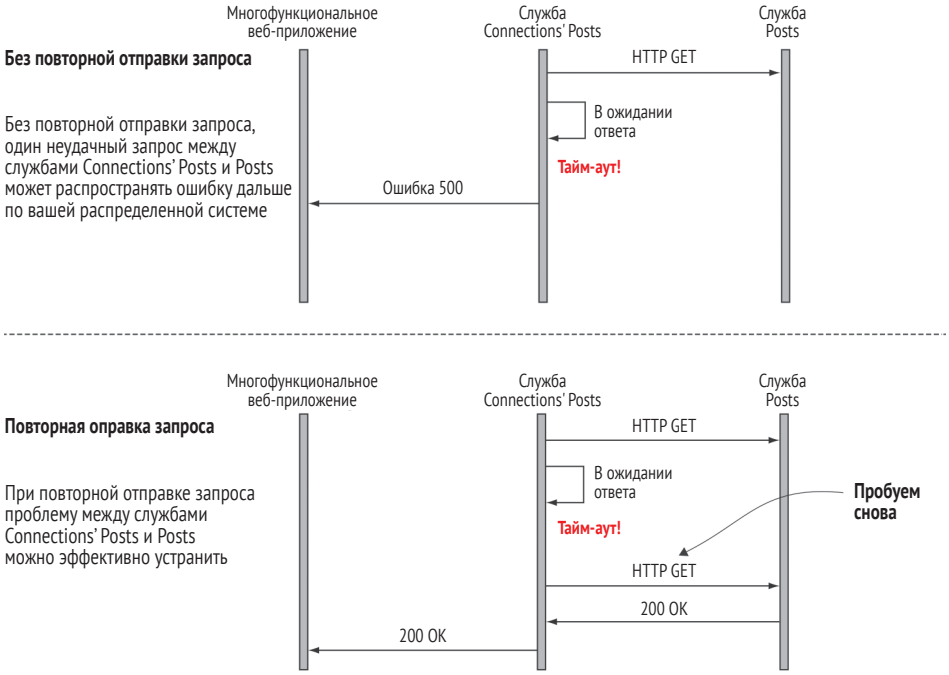


Рис. 9.2 ❖ Повторные отправки запросов могут изолировать части вашей распределенной системы от ошибок в других частях

Эта простая повторная отправка сделает всю нашу систему более терпимой к сбоям, возвращая результаты в тех случаях, когда в противном случае службе Connections' Posts не удалось бы создать агрегированный набор сообщений в блоге.

9.1.2. Давайте посмотрим, как это работает: простая повторная отправка запроса

В этой и следующей главах мы проведем серию экспериментов, в которых изучается влияние применения различных шаблонов для облачной среды на взаимодействие между службами. Этот первый пример закладывает основу, а каждая последующая реализация будет строиться на последнем. Мы начнем с осуществления простой повторной отправки запроса.

Настройка

На этом этапе я отсылаю вас к инструкциям по настройке для запуска примеров из предыдущих глав. В этой главе нет новых требований для запуска образца.

Вы будете получать доступ к файлам в каталоге cloudnative-requestresilience, поэтому в окне терминала перейдите в этот каталог.

Как и в предыдущих главах, я уже предварительно собрала образы Docker и сделала их доступными в Docker Hub. Если вы хотите собрать исходный код Java и об-

разы Docker и перенести их в собственный репозиторий образов, я отсылаю вас к предыдущим главам (самые подробные инструкции приведены в главе 5).

Запуск приложений

По мере продвижения вы будете использовать разные версии шаблона повторной отправки запроса, поэтому для начала нужно извлечь правильный тег в репозитории GitHub:

```
git checkout requestretries/0.0.1
```

Вам понадобится кластер Kubernetes, и для этого начального примера можно использовать Minikube. См. раздел 5.2.2 в главе 5, чтобы найти инструкции по настройке и запуску Minikube. Начнем с чистого листа. Удалите все развертывания, которые могут остаться после вашей предыдущей работы. Для этого я предоставлю вам скрипт: `deleteDeploymentComplete.sh`. Этот простой bash-скрипт позволяет поддерживать работу служб MySQL, Redis и Spring Cloud. При вызове скрипта без параметров удаляются только три развертывания микросервисов; при вызове скрипта с аргументом `all` службы MySQL, Redis и SCSS также удаляются. Убедитесь, что ваша среда очищена, с помощью приведенной ниже команды:

```
$ kubectl get all
NAME                            READY STATUS  RESTARTS AGE
pod/mysql-6585c56bff-hfwn5      1/1   Running 0        2m
pod/redis-846b8c56fb-wr6zx      1/1   Running 0        2m
pod/sccs-84cc988f57-d2mgm       1/1   Running 0        2m

NAME                           CLUSTER-IP        EXTERNAL-IP  PORT(S)        AGE
service/connectionsposts-svc   10.101.76.173     <none>       80:31224/TCP   44s
service/connections-svc        10.105.144.139    <none>       80:32290/TCP   44s
service/kubernetes             10.96.0.1         <none>       443/TCP        4m
service/mysql-svc              10.109.9.155      <none>       3306:32260/TCP 2m
service/posts-svc              10.98.202.179     <none>       80:32746/TCP   45s
service/redis-svc              10.109.19.150     <none>       6379:30270/TCP 2m
service/sccs-svc               10.98.94.67       <none>       8888:32640/TCP 2m

NAME                      DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
deployment.apps/mysql     1       1       1          1         2m
deployment.apps/redis     1       1       1          1         2m
deployment.apps/sccs      1       1       1          1         2m

NAME                                DESIRED CURRENT READY AGE
replicaset.apps/mysql-6585c56bff    1       1       1     2m
replicaset.apps/redis-846b8c56fb    1       1       1     2m
replicaset.apps/sccs-84cc988f57     1       1       1     2m
```

Обратите внимание, что мы оставили `mysql`, `redis` и `sccs` в работающем состоянии, так же как службы для наших трех микросервисов. Если вы удалили `redis`, `mysql` и `sccs`, разверните их, запустив скрипт `deployServices.sh`. Если вы создали службу MySQL заново, не забудьте создать базу данных поваренной книги с помощью этих команд:

```
$ mysql -h $(minikube service mysql-svc --format “{{.IP}}”) \
      -P $(minikube service mysql-svc --format “{{.Port}}”) -u root -p
mysql> create database cookbook;
```

Теперь вы можете развернуть три микросервиса, трижды выполнив команду `kubectl apply`, указав путь к YAML-файлам Connections' Posts, Connections и Posts. Я создала скрипт, который инкапсулирует все три службы, поэтому можете просто запустить его:

```
./deployApps.sh
```

Как и прежде, для вызова микросервиса Connections' Posts сначала выполните вход, а затем получите доступ к списку постов:

```
curl -i -X POST -c cookie \
   $(minikube service --url connectionsposts-svc)/login?username=cdavisafc
curl -i -b cookie \
   $(minikube service --url connectionsposts-svc)/connectionsposts
```

На этом этапе вы должны иметь возможность выполнить эту последнюю команду несколько раз с согласованными результатами.

Теперь давайте создадим ряд проблем. Напомню, что в предыдущей главе мы добавили конечную точку в службу Posts, чтобы нарушить ее работу. При отправке HTTP-запроса с помощью метода POST для конечной точки `/infect` ответы на последующие запросы к службе будут задерживаться на 400 секунд. Выглядит довольно неисправно. Обратите внимание, что в манифесте развертывания я удалила проверку живости, которую добавила в конце главы 8; в этих экспериментах я хочу, чтобы службы были в неисправном состоянии. В настоящее время у нас работают два экземпляра службы Posts, поэтому давайте выведем из строя один из них, выполнив этот запрос с помощью метода POST:

```
curl -i -X POST $(minikube service --url posts-svc)/infect
```

Прежде чем снова вызывать службу Connections' Posts, давайте начнем потоковую передачу журналов этой службы, выполнив приведенную ниже команду в другом окне терминала:

```
kubectl logs -f <name of your Connections' Posts pod>
```

Теперь зайдите в службу Connections' Posts еще несколько раз. Я хотела бы, чтобы вы обратили внимание на две момента. Во-первых, при каждом использовании команды `curl` вы получаете ответ – служба агрегации работает просто отлично. А во-вторых, просматривая журналы, можно увидеть такие записи:

```
... : [172.17.0.10:8080] getting posts for user network cdavisafc
... : [172.17.0.10:8080] connections = 2,3
... : [172.17.0.10:8080] On (0) request to unhealthy posts service I/O
➥ error on GET request for "http://posts-svc/posts": Read timed out;
➥ nested exception is java.net.SocketTimeoutException: Read timed out
... : [172.17.0.10:8080] On (1) request to unhealthy posts service I/O
➥ error on GET request for "http://posts-svc/posts": Read timed out;
➥ nested exception is java.net.SocketTimeoutException: Read timed out
... : [172.17.0.10:8080] On (2) request to unhealthy posts service I/O
➥ error on GET request for "http://posts-svc/posts": Read timed out;
➥ nested exception is java.net.SocketTimeoutException: Read timed out
... : [172.17.0.10:8080] Retrieved results from database
```

Это показывает, что запросы были отправлены в службу Posts (время ожидания истекло), но, вместо того чтобы эта ошибка распространялась обратно к вам, кли-

енту, служба Connections' Posts автоматически восстанавливается при повторной отправке запроса. Даже если потребовалось несколько отправок (в предыдущем примере их было три), в конечном итоге непораженная служба Posts оказалась доступна, и результат был возвращен.

Давайте посмотрим на реализацию из файла ConnectionsPostsController.java.

Листинг 9.1 ❖ Выдержка из файла ConnectionsPostsController.java

```
int retryCount = 0;
while (implementRetries || retryCount == 0) {
  try {
    RestTemplate restTemp = restTemplateBuilder
                            .setConnectTimeout(connectTimeout)
                            .setReadTimeout(readTimeout)
                            .build();
    ResponseEntity<PostResult[]> respPosts
      = restTemp.getForEntity(postsUrl + ids + secretQueryParam,
                            PostResult[].class);
    if (respPosts.getStatusCode().is5xxServerError()) {
      response.setStatus(500);
      return null;
    } else {
      logger.info(utils.ipTag() + "Retrieved results from database");
      PostResult[] posts = respPosts.getBody();
      for (int i = 0; i < posts.length; i++)
        postSummaries.add(
          new PostSummary(getUsersname(posts[i].getUserId()),
          posts[i].getTitle(), posts[i].getDate()));
      return postSummaries;
    }
  } catch (Exception e) {
    // Произойдет при тайм-ауте соединения.
    // Поскольку это простая реализация, мы просто
    // попробуем снова;
    logger.info(utils.ipTag() +
      "On (" + retryCount + ") request to unhealthy posts service " +
      e.getMessage());
    if (implementRetries)
      retryCount++;
    else {
      logger.info(utils.ipTag() +
        "Not implementing retries - returning with a 500");
      response.setStatus(500);
      return null;
    }
  }
}
```

Как видите, это простая реализация. Если вы реализуете повторные отправки запроса, которые контролируются с помощью нового свойства приложения, вы отправите запрос в службу Posts. Если время истекло, вы остаетесь в цикле `while` и пробуете снова. Вы можете увидеть строки, генерирующие те самые сообщения журнала, которые вы просматривали при запуске образца.

Хотя это и действительно просто, здесь мы уже переходим к первому нюансу: сколько ваша реализация будет ждать, прежде чем будет выброшено исключение тайм-аута? (В конечном счете это решение, которое, скорее всего, примет оператор приложения – и это можете быть и вы.) Слишком долго, и тогда ваши вышестоящие клиенты (веб-страница, вызывающая службу Connections' Posts) могут оставаться в состоянии ожидания в течение длительного периода времени, что может привести к тайм-ауту. Слишком мало, и тогда служба Connections' Posts может лишиться абсолютно достоверных результатов (и стать причиной последствий, описанных в следующем разделе). В текущей реализации я установила тайм-аут соединения на ¼ секунды, а тайм-аут на чтение на ½ секунды, как видно в этих строках:

```
RestTemplate restTemplate = restTemplateBuilder
                                    .setConnectTimeout(250)
                                    .setReadTimeout(500)
                                    .build();
```

Я хочу обратить ваше внимание на то, что во всем, что вы здесь проделали, и в случае, когда человек обновил веб-страницу, и в нашей реализации, вы не беспокоились по поводу того, *почему* не получили ответ от нижестоящей службы. Это может быть проблема с сетью, или ошибка в приложении (возможно, та, что я продемонстрировала здесь), или ряд других проблем. Когда проблема временная, причина чаще всего несущественна. Только когда вы начнете сталкиваться с постоянными проблемами, вас это будет волновать, и даже тогда это касается не ваших приложений, это, скорее, общая проблема мониторинга. Мы расскажем об устранении неполадок в следующей главе.

Итак, все это выглядит довольно прилично и легко и, похоже, работает. На какие из этих последствий и постоянных проблем я по-прежнему намекаю? Давайте перейдем к ним.

9.1.3. Повторная отправка запроса: что может пойти не так?

В предыдущем примере программное обеспечение работает нормально с ограниченной нагрузкой, которую вы помещаете в систему при помощи команды `curl`. Но когда в системе, которая в остальном хорошо настроена для конкретной нагрузки, что-то идет не так, это уже совсем другая история. Это немного похоже на уличное движение.

Возьмем автостраду, у которой имеется достаточно полос для движения, чтобы 14 000 автомобилей, движущихся со скоростью 60 миль в час, могли проехать ее за час (по моим приближенным подсчетам, это шоссе с четырьмя полосами движения). Аварий пока нет, все в порядке. Но когда две полосы становятся недоступными из-за мелкого ДТП, ситуация быстро меняется. Тому же потоку машин, который в настоящее время движется по двум остальным полосам, не только придется замедлиться, чтобы сохранить безопасную дистанцию вождения, но и автомобили, приближающиеся к ограниченному участку шоссе, при таком же объеме, что и раньше, очень быстро создадут довольно большую пробку. И как мы все, вероятно, знаем, даже когда место аварии очищено, требуется время, прежде чем очередь из автомобилей рассосется. Этот сценарий изображен на рис. 9.3.

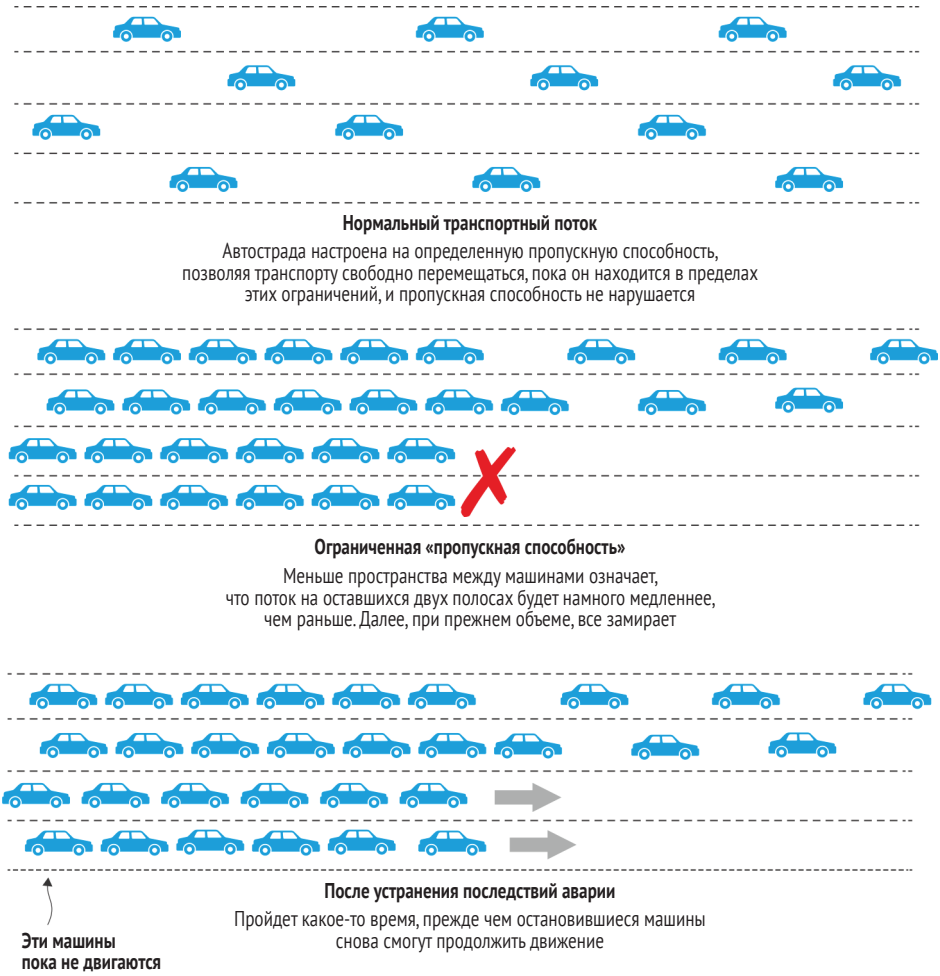


Рис. 9.3 ❖ Ограниченные сети действуют как ограниченные магистрали, резервируя запросы, которые продолжают поступать в том же объеме, что и раньше. Даже когда ограничения сняты, может пройти некоторое время, прежде чем весь трафик в очереди снова начнет двигаться

Ситуация с потоком запросов через вашу сеть экземпляров приложения выглядит точно так же. Несмотря на то что ваш специалист по надежности сайта, без сомнения, разработал топологию развертывания, которая оставляет немного места для колебаний объема запросов и незначительных сбоев, при резервном копировании значительной части вашей нагрузки эти эффекты могут распространиться по всей системе. Давайте посмотрим, как это работает.

9.1.4. Создание шквала повторных отправок запроса

В предыдущем разделе я представила базовый шаблон: служба Connections' Posts будет совершать повторную отправку запроса к службе Posts, если не получит от-

вет изначально. На высоком уровне это имеет смысл, и текущая реализация прекрасно справится с незначительными сбоями. Но когда происходит нечто более существенное (метафорическая авария на шоссе), повторные попытки могут быть не такими полезными и даже могут отрицательно повлиять на общее состояние системы. Здесь я хочу провести эксперимент с простой реализацией, которую дала в предыдущем разделе.

Напомню, что мы сосредоточены на взаимодействии между службой Connections' Posts (клиент) и службой Posts. Как видно из первого примера, мы реализовали повторную отправку для этого запроса. На рис. 9.4 показаны повторные отправки, которые вы видели ранее и увидите еще в следующем примере. Давайте снарядим вас, чтобы вы могли продолжить.

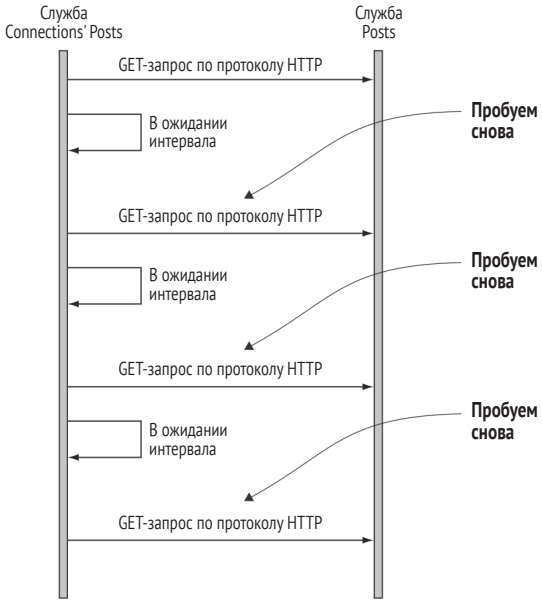


Рис. 9.4 ❖ Клиент, служба Connections' Posts, будет повторять отправку запросов к службе Posts. Она будет продолжать попытки до тех пор, пока не получит из HTTP-запроса код состояния «успешно»

9.1.5. Давайте посмотрим, как это работает: создание шквала повторных отправок запроса

Данный эксперимент не вносит никаких изменений в код предыдущего эксперимента, но мы отправим в систему значительное количество трафика, смоделируем короткий сбой и будем наблюдать за результатами.

Настройка

Вам понадобится все, что я перечислила в первом примере этой главы, с одной поправкой:

- доступ к более крупному кластеру Kubernetes;
- этот кластер должен позволять запуск привилегированных контейнеров.

Мы будем нагружать свое программное обеспечение, а затем вносить сбои в систему. Я хочу, чтобы вы исследовали, например, что происходит, когда вы теряете все полосы на нашем метафорическом шоссе, и, это не менее важно, что происходит, когда эти полосы восстанавливаются. Чтобы иметь возможность значительно увеличить нагрузку на систему, вы создадите более крупное развертывание нашего приложения, используемого в качестве примера; следовательно, вам понадобится более крупное окружение, чтобы запустить его.

Не вдаваясь в детали довольно сложной темы, скажу следующее: привилегированные контейнеры позволяют выполнять в них больше команд, по сравнению с непривилегированными контейнерами, и вам это понадобится, когда вы приступите к ограничению сетевого трафика. Хорошая новость состоит в том, что большинство облачных провайдеров будет обслуживать кластеры Kubernetes с привилегированными контейнерами, которые активированы по умолчанию.

На момент написания этих строк я обнаружила, что Google Kubernetes Engine (GKE) предоставляет простейший общедоступный облачный интерфейс при создании кластеров Kubernetes. Вам понадобится кластер с количеством памяти примерно 25–30 Гб на всех узлах. GKE также активирует привилегированные контейнеры по умолчанию, а вам это понадобится.

Чтобы запустить симуляции в данном разделе, пожалуйста, скачайте приведенный ниже тег из своего репозитория Git:

```
git checkout requestretries/0.0.2
```

Запуск приложений

Выполнение экспериментов состоит из трех частей:

1) развертывание приложения;
2) нагрузка на приложение;
3) моделирование различных сценариев, приводящих к сбоям, и наблюдение за результатами.

Если вы еще не запускали примеры в более крупном кластере Kubernetes или в кластере, размер которого можно изменить, чтобы располагать достаточной емкостью, вам, вероятно, придется развернуть все компоненты, из которых состоит наш образец, заново. Я не буду подробно описывать здесь установку, а вместо этого сошлюсь на предыдущие главы, но в заключение, после создания нового кластера Kubernetes и подключения к нему с помощью команды `kubectl`, вам нужно сделать следующее:

1) отредактируйте манифест развертывания для сервера конфигурации Spring Cloud (SCCS), spring-cloud-config-server-deploy-kubernetes.yaml, чтобы указать путь к репозиторию Git, в котором находятся конфигурации вашего приложения. Разумеется, вы можете оставить данные моего хранилища;
2) разверните службы MySQL, Redis и SCCS. Я предоставила для этого скрипт, `deployServices.sh`, поэтому можете просто запустить его;
3) создайте базу данных поваренной книги, подключившись к MySQL с помощью клиента командной строки и выполнив команду `create database cookbook;`. Обратите внимание, что манифест развертывания MySQL указывает `LoadBalancer` для типа службы, который должен был выделить публичный IP-адрес для вашей базы данных MySQL. Вы можете использовать это, чтобы подключиться к интерфейсу командной строки `mysql`;
4) разверните все три микросервиса, выполнив скрипт `deployApps.sh`.

В результате вы получите развертывание, которое выглядит примерно так:

```
$ kubectl get pods
NAME                                READY STATUS   RESTARTS  AGE
connection-posts-685c669f7b-4qvx7   1/1   Running  0         6d
connection-posts-685c669f7b-6lgmf   1/1   Running  0         6d
connection-posts-685c669f7b-6pt9p   1/1   Running  0         6d
connection-posts-685c669f7b-d8q8h   1/1   Running  0         6d
connection-posts-685c669f7b-z7gsw   1/1   Running  0         6d
connections-7cf9b5ccf9-cjnhs        1/1   Running  0         6d
connections-7cf9b5ccf9-cw4s9        1/1   Running  0         6d
connections-7cf9b5ccf9-kskqm        1/1   Running  0         6d
connections-7cf9b5ccf9-mfj8b        1/1   Running  0         6d
connections-7cf9b5ccf9-nd4nw        1/1   Running  0         6d
connections-7cf9b5ccf9-nnl8r        1/1   Running  0         6d
connections-7cf9b5ccf9-xjq8j        1/1   Running  0         6d
mysql-64bd6d89d8-96vb6              1/1   Running  0         27d
posts-7785bcf45-9tfj4               1/1   Running  0         6d
posts-7785bcf45-bsn8g               1/1   Running  0         6d
posts-7785bcf45-w5xzs               1/1   Running  0         6d
posts-7785bcf45-wtbv8               1/1   Running  0         6d
redis-846b8c56fb-bm5z9              1/1   Running  0        27d
sccs-84cc988f57-hp2z2               1/1   Running  0        27d
```

Для создания нагрузки на приложение мы будем использовать Apache JMeter. Я создала развертывание для JMeter в Kubernetes, а также файл конфигурации, содержащий спецификации нашего нагрузочного теста. Первый шаг – загрузка файла конфигурации, что вы будете делать при создании карты конфигурации Kubernetes. Выполните приведенную ниже команду:

```
kubectl create configmap jmeter-config \
 --from-file=jmeter_run.jmx=loadTesting/ConnectionsPostsLoad.jmx
```

Если вы хотите запустить нагрузочные тесты, то можете просто создать развертывание JMeter; чтобы остановить тестирование, удалите развертывание. Давайте попробуем сделать это сейчас. Выполните эту команду:

```
kubectl create -f loadTesting/jmeter-deployment.yaml
```

Чтобы увидеть вывод JMeter, выполните потоковую передачу журналов для модуля JMeter с помощью приведенной ниже команды (вставив имя своего модуля):

```
kubectl logs -f <name of your jmeter pod>
```

Вы увидите вывод журнала:

```
$ kubectl logs -f jmeter-deployment-7d747c985-kjxct
START Running Jmeter on Mon Feb 18 19:42:17 UTC 2019
JVM_ARGS=-Xmn506m -Xms2024m -Xmx2024m
jmeter args=-n -t /etc/jmeter/jmeter_run.jmx
Feb 18, 2019 7:42:19 PM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
Creating summariser <summary>
Created the tree successfully using /etc/jmeter/jmeter_run.jmx
Starting the test @ Mon Feb 18 19:42:19 UTC 2019 (1550518939413)
```

```
Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445
summary +    530 in 00:00:30 =   17.7/s Err:     0 (0.00%) Active: 328
summary =    612 in 00:00:40 =   15.3/s Err:     0 (0.00%)
summary +   1027 in 00:00:30 =   34.3/s Err:     0 (0.00%) Active: 576
summary =   1639 in 00:01:10 =   23.4/s Err:     0 (0.00%)
summary +   1521 in 00:00:30 =   50.7/s Err:     0 (0.00%) Active: 823
summary =   3160 in 00:01:40 =   31.6/s Err:     0 (0.00%)
summary +   2014 in 00:00:30 =   66.6/s Err:     0 (0.00%) Active: 1073
summary =   5174 in 00:02:10 =   39.7/s Err:     0 (0.00%)
summary +   2512 in 00:00:30 =   84.4/s Err:     0 (0.00%) Active: 1319
summary =   7686 in 00:02:40 =   48.0/s Err:     0 (0.00%)
summary +   2939 in 00:00:30 =   98.0/s Err:     0 (0.00%) Active: 1500
summary =  10625 in 00:03:10 =   55.9/s Err:     0 (0.00%)
```

Тут показано, что приложение Connections' Posts обрабатывало почти 100 запросов в секунду, после того как нагрузка стала полной (я настроила тесты, чтобы они медленно нарастали), с ошибкой 0,0 % (вы увидите эту цифру по мере продвижения). Чтобы остановить нагрузочный тест, выполните эту команду:

```
kubectl delete deploy jmeter-deployment
```

Теперь, когда вы установили свое развертывание и убедились, что нагрузочное тестирование работает правильно, давайте начнем наши эксперименты.

Мы смоделируем перебои в работе сети между службой Posts и службой MySQL.

В реальных условиях такие сбои могут быть вызваны сбоями в оборудовании (например, потеря физического коммутатора) или ошибкой в конфигурации (например, неправильное изменение правила брандмауэра). Чтобы создать такой же эффект здесь, мы будем изменять правила маршрутизации в службе MySQL, чтобы либо запретить, либо потом, когда мы наладим работу сети, разрешить запросы от конкретных экземпляров службы Posts.

На рис. 9.5 показано пять экземпляров службы Connections' Posts, четыре экземпляра службы Posts и один развернутый нами экземпляр службы MySQL. Линии между каждым экземпляром обозначают различные способы обмена данными между этими службами. Обратите внимание, что каждая служба Posts снабжена IP-адресом и что для службы MySQL указано имя модуля. Чтобы запретить трафику идти по одному из этих подключений, мы создадим правило маршрутизации в экземпляре MySQL, которое отклоняет трафик с определенного IP-адреса. Для этого мы выполним команду `route` в контейнере MySQL, а делается это с помощью команды `kubectl exec`.

Чтобы разорвать сетевое соединение между службой Posts, работающей на адресе 10.36.1.13, и экземпляром Posts, работающим в модуле с именем mysql-57bdb878f5-dhlck, мы выполним приведенную ниже команду:

```
kubectl exec mysql-57bdb878f5-dhlck -- route add -host 10.36.1.13 reject
```

Это показано на рис. 9.5 с помощью символа X на линии связи. Когда служба Posts, работающая на адресе 10.36.1.13, пытается подключиться к службе MySQL, время ожидания истекает. Этот тайм-аут будет распространяться на службу Connections' Posts, и результатом будет повторная попытка. Если вам повезет, эта попытка достигнет другого экземпляра Posts и сможет получить доступ к базе данных, и запрос Connections' Posts будет выполнен успешно.

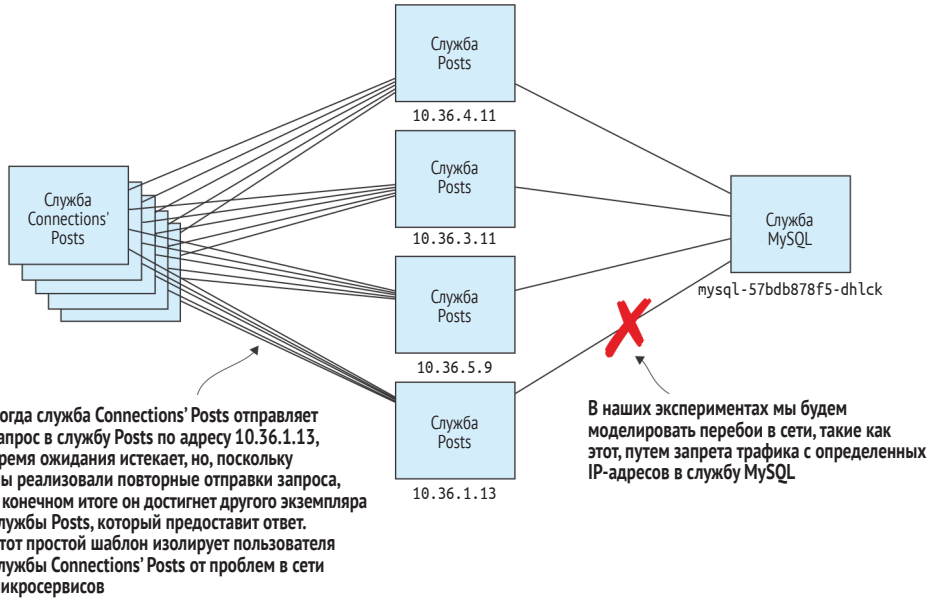


Рис. 9.5 ❖ Развертывание, содержащее пять экземпляров службы Connections' Posts и четыре экземпляра службы Posts, дает 20 способов подключения экземпляра первой службы к экземпляру второй. Между экземплярами службы Posts и единственным экземпляром службы MySQL существует четыре соединения. Повторная отправка запроса – эффективный способ найти работоспособный путь через сеть микросервисов

Восстановление соединения выполняется с помощью аналогичной команды `kubectl exec`, удаляющей правило отклонения маршрутизации из контейнера, на котором работает служба MySQL:

```
kubectl exec mysql-57bdb878f5-dhlck -- route delete -host 10.36.1.13 reject
```

После этого давайте проведем два эксперимента:

1) полностью отключите, с помощью приведенных ранее команд `route`, все экземпляры Posts от службы MySQL, при этом *логика повторной попытки должна быть активирована*. Это можно сделать, установив для переменной среды `CONNECTIONPOSTCONTROLLER_IMPLEMENTRETRIES` в манифесте развертывания для приложения Connections' Posts значение `true`;
2) полностью отключите, с помощью приведенных ранее команд `route`, все экземпляры Posts от службы MySQL, при этом *логика повторной попытки должна быть деактивирована.* Это можно сделать, установив для переменной среды `CONNECTIONPOSTCONTROLLER_IMPLEMENTRETRIES` в манифесте развертывания для приложения Connections' Posts значение `false`. По истечении времени ожидания попыток подключения к службе Posts служба Connections' Posts вернется с ошибкой и отсутствием результата.

Чтобы избавить вас от необходимости четыре раза выполнять предыдущую команду `kubectl exec` вручную, я предоставлю вам скрипт `alternetwork-db.sh`. Однако вам придется отредактировать его, чтобы отразить имя модуля MySQL и IP-адреса ваших экземпляров службы Posts.

Вы можете получить имя своей службы MySQL с помощью обычной команды kubectl:

```
kubectl get pods
```

А чтобы получить IP-адреса экземпляров Posts, используйте это:

```
kubectl get pods -l app=posts -o wide
```

Теперь вы можете запретить всем экземплярам Posts подключаться к экземпляру MySQL, выполнив эту команду:

```
./alternetwork-db.sh add
```

Ниже приведен вывод журнала одного из экземпляров службы Connections' Posts, показывающий, что он перешел от обслуживания трафика к выполнению только повторных отправок:

```
2019-02-18 04:05:55.986 ... connections = 2,3
2019-02-18 04:05:55.989 ... getting posts for user network cdavisafc
2019-02-18 04:05:55.995 ... connections = 2,3
2019-02-18 04:05:56.055 ... getting posts for user network cdavisafc
2019-02-18 04:05:56.056 ... getting posts for user network cdavisafc
2019-02-18 04:05:56.059 ... On (0) request to unhealthy posts service I/O
➥ error on GET request for "http://posts-svc/posts": Connect to posts-
➥ svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested
➥ exception is org.apache.http.conn.ConnectTimeoutException: Connect to
➥ posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out
2019-02-18 04:05:56.060 ... connections = 2,3
2019-02-18 04:05:56.060 ... connections = 2,3
2019-02-18 04:05:56.070 ... getting posts for user network cdavisafc
2019-02-18 04:05:56.074 ... connections = 2,3
2019-02-18 04:05:56.092 ... On (1) request to unhealthy posts service I/O
➥ error on GET request for "http://posts-svc/posts": Connect to posts-
➥ svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested
➥ exception is org.apache.http.conn.ConnectTimeoutException: Connect to
➥ posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out
2019-02-18 04:05:56.093 ... On (2) request to unhealthy posts service I/O
➥ error on GET request for "http://posts-svc/posts": Connect to posts-
➥ svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested
➥ exception is org.apache.http.conn.ConnectTimeoutException: Connect to
➥ posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out
2019-02-18 04:05:56.229 ... On (0) request to unhealthy posts service I/O
➥ error on GET request for "http://posts-svc/posts": Connect to posts-
➥ svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested
➥ exception is org.apache.http.conn.ConnectTimeoutException: Connect to
➥ posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out
2019-02-18 04:05:56.232 ... On (0) request to unhealthy posts service I/O
➥ error on GET request for "http://posts-svc/posts": Connect to posts-
➥ svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested
➥ exception is org.apache.http.conn.ConnectTimeoutException: Connect to
➥ posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out
2019-02-18 04:05:56.310 ... On (0) request to unhealthy posts service I/O
➥ error on GET request for "http://posts-svc/posts": Connect to posts-
➥ svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested
```

```
➥ exception is org.apache.http.conn.ConnectTimeoutException: Connect to
➥ posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out
2019-02-18 04:05:56.343 ... On (6) request to unhealthy posts service I/O
➥ error on GET request for "http://posts-svc/posts": Connect to posts-
➥ svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested
➥ exception is org.apache.http.conn.ConnectTimeoutException: Connect to
➥ posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out
```

Ниже показан вывод JMeter, аннотированный тремя временными точками. Когда начинается тест, приложение Connections' Posts возвращает результаты с ошибкой 0,0 %. Затем на метке времени 1, когда вы выполняете команду `./alter-network-db.sh add`, видно, что процент ошибок быстро достигает 100 %. Служба Connections' Posts так и не вернулась, JMeter истекает по запросу (и считает попытку как ошибку), однако приложения Connections' Posts продолжают выполнять повторную отправку запроса к службе Posts до бесконечности.

```
START Running Jmeter on Mon Feb 18 20:08:18 UTC 2019
JVM_ARGS=-Xmn402m -Xms1608m -Xmx1608m
jmeter args=-n -t /etc/jmeter/jmeter_run.jmx
Feb 18, 2019 8:08:20 PM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
Creating summariser <summary>
Created the tree successfully using /etc/jmeter/jmeter_run.jmx
Starting the test @ Mon Feb 18 20:08:21 UTC 2019 (1550520501121)
Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445
summary +     67 in 00:00:08 =    8.2/s Err:     0 (0.00%) Active: 67
summary +    501 in 00:00:30 =   16.7/s Err:     0 (0.00%) Active: 314
summary =    568 in 00:00:38 =   14.9/s Err:     0 (0.00%)
summary +    999 in 00:00:30 =   33.3/s Err:     0 (0.00%) Active: 562
summary =   1567 in 00:01:08 =   23.0/s Err:     0 (0.00%)
summary +   1493 in 00:00:30 =   49.8/s Err:     0 (0.00%) Active: 810
summary =   3060 in 00:01:38 =   31.2/s Err:     0 (0.00%)
summary +   1992 in 00:00:30 =   66.4/s Err:     0 (0.00%) Active: 1059
summary =   5052 in 00:02:08 =   39.4/s Err:     0 (0.00%)
summary +   2488 in 00:00:30 =   82.9/s Err:     0 (0.00%) Active: 1307
summary =   7540 in 00:02:38 =   47.7/s Err:     0 (0.00%)
summary +   2929 in 00:00:30 =   97.7/s Err:     0 (0.00%) Active: 1500
summary =  10469 in 00:03:08 =   55.7/s Err:     0 (0.00%)
summary +   2997 in 00:00:30 =   99.9/s Err:     0 (0.00%) Active: 1500
summary =  13466 in 00:03:38 =   61.7/s Err:     0 (0.00%)

<time marker 1 - I have broken the network between Posts and MySQL>

summary +   2515 in 00:00:30 =   83.8/s Err:  2239 (89.03%) Active: 1500
summary =  15981 in 00:04:08 =   64.4/s Err:  2239 (14.01%)
summary +   3000 in 00:00:30 =  100.0/s Err:  3000 (100.00%) Active: 1500
summary =  18981 in 00:04:38 =   68.2/s Err:  5239 (27.60%)
summary +   2961 in 00:00:30 =   98.7/s Err:  2961 (100.00%) Active: 1500
summary =  21942 in 00:05:08 =   71.2/s Err:  8200 (37.37%)
summary +   2970 in 00:00:30 =   99.0/s Err:  2970 (100.00%) Active: 1500
summary =  24912 in 00:05:38 =   73.7/s Err: 11170 (44.84%)
summary +   3007 in 00:00:30 =  100.1/s Err:  3007 (100.00%) Active: 1500
summary =  27919 in 00:06:08 =   75.8/s Err: 14177 (50.78%)
```

```
summary +   2968 in 00:00:30 =   99.0/s Err:  2968 (100.00%) Active: 1500
summary =  30887 in 00:06:38 =   77.6/s Err: 17145 (55.51%)

<time marker 2 – I have repaired the network between Posts and MySQL>

summary +   3007 in 00:00:30 =  100.2/s Err:  3007 (100.00%) Active: 1500
summary =  33894 in 00:07:08 =   79.2/s Err: 20152 (59.46%)
summary +   2995 in 00:00:30 =   99.8/s Err:  2995 (100.00%) Active: 1500
summary =  36889 in 00:07:38 =   80.5/s Err: 23147 (62.75%)
summary +   2997 in 00:00:30 =   99.9/s Err:  2997 (100.00%) Active: 1500
summary =  39886 in 00:08:08 =   81.7/s Err: 26144 (65.55%)
summary +   3000 in 00:00:30 =   99.9/s Err:  3000 (100.00%) Active: 1500
summary =  42886 in 00:08:38 =   82.8/s Err: 29144 (67.96%)

<another 6 minutes of 100% error!!>

summary +   3011 in 00:00:30 =  100.4/s Err:  3011 (100.00%) Active: 1500
summary =  78913 in 00:14:38 =   89.9/s Err: 65171 (82.59%)
summary +   2982 in 00:00:30 =   99.4/s Err:  2982 (100.00%) Active: 1500
summary =  81895 in 00:15:08 =   90.2/s Err: 68153 (83.22%)
summary +   3057 in 00:00:30 =  101.9/s Err:  2999 (98.10%) Active: 1500
summary =  84952 in 00:15:38 =   90.6/s Err: 71152 (83.76%)
summary +   3054 in 00:00:30 =  101.8/s Err:  2390 (78.26%) Active: 1500
summary =  88006 in 00:16:08 =   90.9/s Err: 73542 (83.56%)
summary +   2982 in 00:00:30 =   99.3/s Err:  2442 (81.89%) Active: 1500
summary =  90988 in 00:16:38 =   91.2/s Err: 75984 (83.51%)
summary +   3025 in 00:00:30 =  101.0/s Err:  2418 (79.93%) Active: 1500
summary =  94013 in 00:17:08 =   91.4/s Err: 78402 (83.39%)
summary +   2991 in 00:00:30 =   99.7/s Err:  2374 (79.37%) Active: 1500
summary =  97004 in 00:17:38 =   91.7/s Err: 80776 (83.27%)
summary +   3106 in 00:00:30 =  103.5/s Err:  2253 (72.54%) Active: 1500
summary = 100110 in 00:18:08 =   92.0/s Err: 83029 (82.94%)
summary +   3017 in 00:00:30 =  100.6/s Err:  1825 (60.49%) Active: 1500
summary = 103127 in 00:18:38 =   92.2/s Err: 84854 (82.28%)
summary +   2997 in 00:00:30 =   99.9/s Err:  1839 (61.36%) Active: 1500
summary = 106124 in 00:19:08 =   92.4/s Err: 86693 (81.69%)
summary +   2987 in 00:00:30 =   99.5/s Err:  1787 (59.83%) Active: 1500
summary = 109111 in 00:19:38 =   92.6/s Err: 88480 (81.09%)
summary +   3036 in 00:00:30 =  101.3/s Err:  1793 (59.06%) Active: 1500
summary = 112147 in 00:20:08 =   92.8/s Err: 90273 (80.50%)
summary +   2985 in 00:00:30 =   99.5/s Err:  1795 (60.13%) Active: 1500
summary = 115132 in 00:20:38 =   93.0/s Err: 92068 (79.97%)
summary +   2988 in 00:00:30 =   99.6/s Err:  1786 (59.77%) Active: 1500
summary = 118120 in 00:21:08 =   93.1/s Err: 93854 (79.46%)
summary +   3009 in 00:00:30 =  100.1/s Err:  1859 (61.78%) Active: 1500
summary = 121129 in 00:21:38 =   93.3/s Err: 95713 (79.02%)
summary +   3021 in 00:00:30 =  100.9/s Err:  1829 (60.54%) Active: 1500
summary = 124150 in 00:22:08 =   93.5/s Err: 97542 (78.57%)
summary +   3001 in 00:00:30 =  100.1/s Err:  1802 (60.05%) Active: 1500
summary = 127151 in 00:22:38 =   93.6/s Err: 99344 (78.13%)
summary +   3121 in 00:00:30 =  104.0/s Err:  1308 (41.91%) Active: 1500
summary = 130272 in 00:23:08 =   93.8/s Err: 100652 (77.26%)
summary +   3096 in 00:00:30 =  103.1/s Err:  1036 (33.46%) Active: 1500
summary = 133368 in 00:23:38 =   94.0/s Err: 101688 (76.25%)
```

```
summary +   2976 in 00:00:30 =   99.3/s Err:   596 (20.03%) Active: 1500
summary = 136344 in 00:24:08 =   94.2/s Err: 102284 (75.02%)
summary +   3005 in 00:00:30 =  100.1/s Err:   583 (19.40%) Active: 1500
summary = 139349 in 00:24:38 =   94.3/s Err: 102867 (73.82%)
summary +   3002 in 00:00:30 =  100.1/s Err:   634 (21.12%) Active: 1500
summary = 142351 in 00:25:08 =   94.4/s Err: 103501 (72.71%)
summary +   2999 in 00:00:30 =  100.0/s Err:   596 (19.87%) Active: 1500
summary = 145350 in 00:25:38 =   94.5/s Err: 104097 (71.62%)
summary +   3013 in 00:00:30 =  100.4/s Err:   580 (19.25%) Active: 1500
summary = 148363 in 00:26:08 =   94.6/s Err: 104677 (70.55%)
summary +   3016 in 00:00:30 =  100.5/s Err:   579 (19.20%) Active: 1500
summary = 151379 in 00:26:38 =   94.7/s Err: 105256 (69.53%)
summary +   2999 in 00:00:30 =  100.0/s Err:   600 (20.01%) Active: 1500
summary = 154378 in 00:27:08 =   94.8/s Err: 105856 (68.57%)
summary +   2999 in 00:00:30 =  100.0/s Err:   571 (19.04%) Active: 1500
summary = 157377 in 00:27:38 =   94.9/s Err: 106427 (67.63%)
summary +   2988 in 00:00:30 =   99.6/s Err:   600 (20.08%) Active: 1500
summary = 160365 in 00:28:08 =   95.0/s Err: 107027 (66.74%)
summary +   3107 in 00:00:30 =  103.6/s Err:    58 (1.87%) Active: 1500
summary = 163472 in 00:28:38 =   95.1/s Err: 107085 (65.51%)
summary +   2995 in 00:00:30 =   99.8/s Err:     0 (0.00%) Active: 1500
summary = 166467 in 00:29:08 =   95.2/s Err: 107085 (64.33%)
summary +   3007 in 00:00:30 =  100.2/s Err:     0 (0.00%) Active: 1500
summary = 169474 in 00:29:38 =   95.3/s Err: 107085 (63.19%)
```

На метке времени 2 в предыдущем выводе, после 3 минут отключения службы MySQL, мы восстанавливаем работу сети, выполняя эту команду:

```
./alternetwork-db.sh delete
```

То, что вы сейчас ищете, – это то, сколько времени потребуется, чтобы система вернулась в устойчивое состояние – состояние, где у службы Connections' Posts 0,0 % ошибок.

Как видите, вывод довольно длинный. Примерно через 9 минут после восстановления сети появляются первые признаки восстановления. Затем потребуется еще 12–13 минут, чтобы система полностью восстановилась. Это и есть *шквал повторных отправок запросов*. Система была настолько перегружена повторными отправками в очереди, что ей потребовалось более четверти часа, чтобы восстановиться. Представьте, что Amazon не сможет совершать сделки купли-продажи в течение такого периода времени. Ему бы это дорого обошлось!

Каким бы плохим это ни казалось, то, что я продемонстрировала здесь, – лишь маленький пример.

В системе с сотнями подключенных экземпляров служб короткий скачок в сети может привести к сбоям сроком на несколько часов, которые даже могут вызвать аварийный сбой экземпляров приложения. Помните историю, с которой началась эта книга? Отключение, случившееся в Amazon, в конечном итоге было вызвано шквалом повторных отправок запросов, который произошел после короткого отключения сети.

> **ПРЕДУПРЕЖДЕНИЕ** Шквал повторных отправок запроса может иметь катастрофические последствия для сложных распределенных систем.

Прежде чем перейти к рассмотрению смягчительных мер, я бы хотела, чтобы вы выполнили тот же тест, но с отключенными повторными отправками. На этот раз при тайм-ауте попытки доступа к службе Posts служба Connections' Posts вернет ошибку, без результата, но все же вернет. Чтобы отключить повторную отправку запроса, измените значение переменной среды `CONNECTIONPOSTCONTROLLER_IMPLEMENTRETRIES` в файле cookbook-deploykubernetes-connectionposts.yaml на `false` и обновите развертывание с помощью этой команды:

```
kubectl apply -f cookbook-deployment-kubernetes-connectionposts.yaml
```

Затем вы можете создать модуль JMeter, как делали это ранее с помощью команды `kubectl create` (если вы еще не удалили предыдущее развертывание, сделайте это, используя команду `kubectl delete deploy`). Ниже приводится вывод JMeter с двумя метками времени:

```
START Running Jmeter on Mon Feb 18 20:58:54 UTC 2019
JVM_ARGS=-Xmn528m -Xms2112m -Xmx2112m
jmeter args=-n -t /etc/jmeter/jmeter_run.jmx
Feb 18, 2019 8:58:56 PM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
Creating summariser <summary>
Created the tree successfully using /etc/jmeter/jmeter_run.jmx
Starting the test @ Mon Feb 18 20:58:56 UTC 2019 (1550523536966)
Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445
summary +     18 in 00:00:02 =    7.9/s Err:     0 (0.00%) Active: 18
summary +    401 in 00:00:30 =   13.4/s Err:     0 (0.00%) Active: 263
summary =    419 in 00:00:32 =   13.0/s Err:     0 (0.00%)
summary +    890 in 00:00:30 =   29.7/s Err:     0 (0.00%) Active: 506
summary =   1309 in 00:01:02 =   21.0/s Err:     0 (0.00%)
summary +   1378 in 00:00:30 =   46.0/s Err:     0 (0.00%) Active: 752
summary =   2687 in 00:01:32 =   29.1/s Err:     0 (0.00%)
summary +   1877 in 00:00:30 =   62.6/s Err:     0 (0.00%) Active: 1000
summary =   4564 in 00:02:02 =   37.3/s Err:     0 (0.00%)
summary +   2369 in 00:00:30 =   79.0/s Err:     0 (0.00%) Active: 1249
summary =   6933 in 00:02:32 =   45.5/s Err:     0 (0.00%)
summary +   2869 in 00:00:30 =   95.6/s Err:     0 (0.00%) Active: 1498
summary =   9802 in 00:03:02 =   53.8/s Err:     0 (0.00%)
summary +   3004 in 00:00:30 =  100.2/s Err:     0 (0.00%) Active: 1500
summary =  12806 in 00:03:32 =   60.3/s Err:     0 (0.00%)
summary +   2998 in 00:00:30 =   99.9/s Err:     0 (0.00%) Active: 1500
summary =  15804 in 00:04:02 =   65.2/s Err:     0 (0.00%)
summary +   3001 in 00:00:30 =  100.0/s Err:     0 (0.00%) Active: 1500
summary =  18805 in 00:04:32 =   69.1/s Err:     0 (0.00%)

<time marker 1 - I have broken the network between Posts and MySQL>

summary +   2951 in 00:00:30 =   98.4/s Err:  2662 (90.21%) Active: 1500
summary =  21756 in 00:05:02 =   72.0/s Err:  2662 (12.24%)
summary +   2999 in 00:00:30 =  100.0/s Err:  2999 (100.00%) Active: 1500
summary =  24755 in 00:05:32 =   74.5/s Err:  5661 (22.87%)
summary +   3001 in 00:00:30 =  100.0/s Err:  3001 (100.00%) Active: 1500
summary =  27756 in 00:06:02 =   76.6/s Err:  8662 (31.21%)
summary +   3000 in 00:00:30 =  100.0/s Err:  3000 (100.00%) Active: 1500
```

```
summary =  30756 in 00:06:32 =   78.4/s Err: 11662 (37.92%)
summary +   3001 in 00:00:30 =  100.0/s Err:  3001 (100.00%) Active: 1500
summary =  33757 in 00:07:02 =   80.0/s Err: 14663 (43.44%)
summary +   3000 in 00:00:30 =  100.0/s Err:  3000 (100.00%) Active: 1500
summary =  36757 in 00:07:32 =   81.3/s Err: 17663 (48.05%)
summary +   2999 in 00:00:30 =  100.0/s Err:  2999 (100.00%) Active: 1500
summary =  39756 in 00:08:02 =   82.4/s Err: 20662 (51.97%)

<time marker 2 - I have repaired the network between Posts and MySQL>

summary +   3051 in 00:00:30 =  101.7/s Err:  1473 (48.28%) Active: 1500
summary =  42807 in 00:08:32 =   83.6/s Err: 22135 (51.71%)
summary +   2999 in 00:00:30 =  100.0/s Err:     0 (0.00%) Active: 1500
summary =  45806 in 00:09:02 =   84.5/s Err: 22135 (48.32%)
```

Как видите, пока работа сети нарушена, служба Connections' Posts сообщает о 100%-ной ошибке. Но самое главное, как только сеть восстанавливается, на метке времени 2 система тотчас же возвращается в стабильное состояние с ошибкой 0,0 %. Нет очереди из повторных отправок, которые обрушиваются на систему.

> **ПРИМЕЧАНИЕ** При использовании повторных отправок запроса системе потребовалось 15 минут для восстановления после трехминутного сбоя в работе сети. Когда повторные отправки *не* использовались, восстановление после трехминутного сбоя было мгновенным.

Итак, мы столкнулись с парадоксом. Повторные отправки запроса могут привести к катастрофическим последствиям, но также могут дать большие преимущества, особенно когда неудавшиеся вызовы являются лишь перемежающимися. Есть ли способ воспользоваться преимуществами повторной отправки, не боясь, что они шквалом обрушатся на систему, вызвав хаос? Да, есть. И их несколько. В этой главе я расскажу о том, как быть умнее, когда мы выполняем повторную отправку запроса, – о том, что значит быть более доброжелательным клиентом. Следующая глава посвящена тому, как поставить защиту перед службой, чтобы не дать менее доброжелательным клиентам создавать проблемы в системе.

9.1.6. Как избежать шквала повторных отправок запросов: добрые клиенты

Несмотря на крайне негативные последствия, свидетелями которых вы стали, когда речь шла о повторных отправках запроса в предыдущем разделе, их ценность остается очевидной. В частности, в случае с периодическими проблемами с подключением повторная отправка часто будет работать, тем самым подавляя ошибку, которая в противном случае могла бы распространиться по всей распределенной системе, из которой состоит наше ПО для облачной среды. Хитрость заключается в том, чтобы сбалансировать напряжение между потенциальными негативными эффектами и позитивными.

Первое наблюдение, которое можно сделать, заключается в том, что для проблем, которые возникают лишь время от времени и в течение ограниченного периода времени, редко требуется более одной или двух попыток повторной отправки запроса, чтобы обмен был успешным. Следовательно, первый элемент управления, который можно применить в нашем цикле повторных отправок, –

это ограничить общее число таких попыток. Так, например, вместо использования цикла while, который выполняется бесконечно, можно реализовать счетчик и остановить повторные отправки запросов, когда вы достигнете порогового значения.

Но что происходит, когда соединение только на мгновение недоступно, а все попытки повторной отправки запроса были исчерпаны до того, как соединение было восстановлено? Вы только что потеряли преимущества повторных отправок, потому что слишком усердно повторяли свои запросы. Введение задержки между попытками обеспечивает здесь некий баланс.

9.1.7. Давайте посмотрим, как это работает: стать более доброжелательным клиентом

Давайте применим два элемента управления, ограничивающих количество попыток повторной отправки запроса и замедляющих частоту отправок, к нашей реализации и посмотрим, как это меняет поведение нашего программного обеспечения, особенно при нагрузке.

Я не буду снова повторять все инструкции по настройке и сборке; моя презентация здесь – всего лишь продолжение предыдущего раздела. Чтобы получить доступ к новой реализации, скачайте приведенный ниже тег из репозитория Git:

```
git checkout requestretries/0.0.3
```

В этом листинге вы увидите, что там, где раньше была простая реализация повторной отправки запроса, у вас теперь идет вот такой код:

Листинг 9.2 ❖ Выдержка из файла ConnectionsPostsController.java

```
try {
    postSummaries = postsServiceClient.getPosts(ids, restTemplate);
    response.setStatus(200);
    return postSummaries;
} catch (HttpServerErrorException e) {
    logger.info(utils.ipTag() + “Call to Posts service returned 500”);
    response.setStatus(500);
    return null;
} catch (ResourceAccessException e) {
    logger.info(utils.ipTag() + “Call to Posts service timed out”);
    response.setStatus(500);
    return null;
} catch (Exception e) {
    logger.info(utils.ipTag() + “Unexpected Exception: Exception Class “
        + e.getClass() + e.getMessage());
    response.setStatus(500);
    return null;
}
```

Обратите внимание, что единственное различие в блоках catch – это сообщение, которое регистрируется, поэтому логически реализация теперь выглядит так:

```
try {
    postSummaries = postsServiceClient.getPosts(ids, restTemplate);
    response.setStatus(200);
    return postSummaries;
```

```
} catch (Exception e) {
    logger.info(utils.ipTag() + e.getMessage());
    response.setStatus(500);
    return null;
}
```

Вы также заметите, что теперь вызов службы Posts осуществляется с помощью нового класса `PostsServiceClient`, который является клиентом службы Posts. Создание этого класса обеспечивает область поверхности, к которой могут применяться аннотации Spring Retry.

С помощью предыдущего кода, если вызов службы Posts прошел успешно, вы возвращаете набор постов, полученных с помощью вызова `postsServiceClient.getPosts`. В противном случае установите статус HTTP на 500 (ошибка) и ничего не возвращайте. Давайте посмотрим на реализацию этого клиента службы Posts.

Листинг 9.3 ❖ Метод из файла PostsServiceClient.java

```
@Retryable( value = ResourceAccessException.class,
            maxAttempts = 3,
            backoff = @Backoff(delay = 500))
public ArrayList<PostSummary> getPosts(String ids,
    RestTemplate restTemplate) throws Exception {

    ArrayList<PostSummary> postSummaries = new ArrayList<PostSummary>();
    String secretQueryParam = "&secret=" + utils.getPostsSecret();
    logger.info("Trying getPosts: " + postsUrl + ids + secretQueryParam);

    ResponseEntity<ConnectionsPostsController.PostResult[]> respPosts
        = restTemplate.getForEntity(postsUrl + ids + secretQueryParam,
                        ConnectionsPostsController.PostResult[].class);
    if (respPosts.getStatusCode().is5xxServerError()) {
        throw new HttpServerErrorException(respPosts.getStatusCode(),
                                "Exception thrown in obtaining Posts");
    } else {
        ConnectionsPostsController.PostResult[] posts
            = respPosts.getBody();
        for (int i = 0; i < posts.length; i++)
            postSummaries.add(
                new PostSummary(
                    getUsersname(posts[i].getUserId(),restTemplate),
                            posts[i].getTitle(), posts[i].getDate()));
        return postSummaries;
    }
}
```

В этом коде используется проект, который является частью фреймворка Spring, Spring Retries (**https://github.com/spring-projects/spring-retry**). Мне кажется интересным, что шаблоны повторной отправки запроса, которые теперь инкапсулированы в этом проекте, изначально были встроены в проект Spring Batch. Будучи извлеченным в собственный проект, он может использоваться во многих сценариях. Например, первая строка файла README проекта Spring Retry гласит: «Он используется в Spring Batch, Spring Integration, Spring для Apache Hadoop (среди прочего)». Повторные отправки запросов настолько распространены в программ-

ном обеспечении для облачной среды, что имеет смысл обзавестись библиотекой, чтобы вы могли легко использовать их во многих случаях.

Хочу обратить ваше внимание на две части этого кода. Во-первых, обратите внимание, что аннотация `@Retryable` включает в себя атрибуты, которые в точности отражают элементы управления, о которых я говорила ранее: ограничение количества повторных отправок и предоставление времени между попытками отправки (период ожидания между попытками будет составлять полсекунды). Также обратите внимание на то, что вы можете указать, что повторные попытки должны предприниматься только для определенных исключений – в этом случае исключения доступа (тайм-аут соединения или на чтение).

Еще одна вещь, которую вы заметите при изучении этого кода, – вы больше не несете ответственности за логику циклов. Данный код – это просто реализация счастливого пути. Он отправляет HTTP-запрос к Posts, а если она возвращает код состояния ошибки HTTP, передает эту ошибку наверх. В противном случае он обрабатывает тело ответа и возвращает эти значения.

Здесь нет строк с try/catch, нет цикла. Однако если этот код должен был сгенерировать исключение `ResourceAccessException`, которое может быть выброшено `restTemplate`, реализация Spring Retry перехватит его и, основываясь на значениях аннотации, возможно, выполнит метод еще раз. С другой стороны, Spring Retry добивается этого с помощью аспектов; отсюда и включение зависимости АОП (аспектно-ориентированного программирования) наряду с зависимостью Spring Retry.

Листинг 9.4 ❖ Выдержка из файла pom.xml службы Connections' Posts

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
    <version>1.2.2.RELEASE</version>
</dependency>
```

Давайте посмотрим, что делает эта реализация в нашем сценарии с нагрузкой из предыдущего раздела. Если вы хотите продолжить, вам, конечно же, придется заново развернуть программное обеспечение. Если вы запустили примеры из предыдущего раздела, можете запустить скрипт `deployApps.sh`. Затем мы выполним тот же самый нагрузочный тест, что и в предыдущем разделе.

Ниже приводится вывод (как видно из журналов модуля JMeter), снова с двумя метками времени:

```
START Running Jmeter on Mon Feb 18 21:58:55 UTC 2019
JVM_ARGS=-Xmn502m -Xms2008m -Xmx2008m
jmeter args=-n -t /etc/jmeter/jmeter_run.jmx -l resultsconnectionsposts
Feb 18, 2019 9:58:57 PM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
Creating summariser <summary>
Created the tree successfully using /etc/jmeter/jmeter_run.jmx
Starting the test @ Mon Feb 18 21:58:57 UTC 2019 (1550527137576)
```

```
Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445
summary +     14 in 00:00:02 =    8.1/s Err:     0 (0.00%) Active: 14
summary +    394 in 00:00:30 =   13.2/s Err:     0 (0.00%) Active: 259
summary =    408 in 00:00:32 =   12.9/s Err:     0 (0.00%)
summary +    887 in 00:00:30 =   29.6/s Err:     0 (0.00%) Active: 508
summary =   1295 in 00:01:02 =   21.0/s Err:     0 (0.00%)
summary +   1388 in 00:00:30 =   46.3/s Err:     0 (0.00%) Active: 756
summary =   2683 in 00:01:32 =   29.3/s Err:     0 (0.00%)
summary +   1887 in 00:00:30 =   62.9/s Err:     0 (0.00%) Active: 1005
summary =   4570 in 00:02:02 =   37.6/s Err:     0 (0.00%)
summary +   2377 in 00:00:30 =   79.3/s Err:     0 (0.00%) Active: 1253
summary =   6947 in 00:02:32 =   45.8/s Err:     0 (0.00%)
summary +   2878 in 00:00:30 =   95.9/s Err:     0 (0.00%) Active: 1500
summary =   9825 in 00:03:02 =   54.1/s Err:     0 (0.00%)
summary +   2993 in 00:00:30 =   99.7/s Err:     0 (0.00%) Active: 1500
summary =  12818 in 00:03:32 =   60.6/s Err:     0 (0.00%)
summary +   3006 in 00:00:30 =  100.2/s Err:     0 (0.00%) Active: 1500
summary =  15824 in 00:04:02 =   65.5/s Err:     0 (0.00%)

<time marker 1 – I have broken the network between Posts and MySQL>

summary +   2645 in 00:00:30 =   88.2/s Err:  2354 (89.00%) Active: 1500
summary =  18469 in 00:04:32 =   68.0/s Err:  2354 (12.75%)
summary +   3002 in 00:00:30 =  100.0/s Err:  3002 (100.00%) Active: 1500
summary =  21471 in 00:05:02 =   71.2/s Err:  5356 (24.95%)
summary +   3000 in 00:00:30 =  100.0/s Err:  3000 (100.00%) Active: 1500
summary =  24471 in 00:05:32 =   73.8/s Err:  8356 (34.15%)
summary +   3006 in 00:00:30 =  100.2/s Err:  3006 (100.00%) Active: 1500
summary =  27477 in 00:06:02 =   76.0/s Err: 11362 (41.35%)
summary +   3015 in 00:00:30 =  100.5/s Err:  3015 (100.00%) Active: 1500
summary =  30492 in 00:06:32 =   77.9/s Err: 14377 (47.15%)
summary +   3051 in 00:00:30 =  101.7/s Err:  3051 (100.00%) Active: 1500
summary =  33543 in 00:07:02 =   79.6/s Err: 17428 (51.96%)

<time marker 2 – I have repaired the network between Posts and MySQL>

summary +   3002 in 00:00:30 =  100.0/s Err:   3002 (100.00%) Active: 1500
summary =  36545 in 00:07:32 =   80.9/s Err:  20430 (55.90%)
summary +   2942 in 00:00:30 =   98.1/s Err:   2942 (100.00%) Active: 1500
summary =  39487 in 00:08:02 =   82.0/s Err:  23372 (59.19%)
summary +   3323 in 00:00:30 =  110.8/s Err:    378 (11.38%) Active: 1500
summary =  42810 in 00:08:32 =   83.7/s Err:  23750 (55.48%)
summary +   3021 in 00:00:30 =  100.6/s Err:      2 (0.07%) Active: 1500
summary =  45831 in 00:09:02 =   84.6/s Err:  23752 (51.83%)
summary +   2998 in 00:00:30 =  100.0/s Err:      0 (0.00%) Active: 1500
summary =  48829 in 00:09:32 =   85.4/s Err:  23752 (48.64%)
summary +   3001 in 00:00:30 =  100.0/s Err:      0 (0.00%) Active: 1500
summary =  51830 in 00:10:02 =   86.1/s Err:  23752 (45.83%)
```

Когда вы начинаете тест, то видите 0,0 % ошибок, идущих из Connections' Posts. Вызовы службы Posts и все другие операции завершаются успешно.

На метке времени 1 вы отключаете службы Posts и MySQL с помощью тех же команд маршрутизации в контейнерах MySQL, используя эту команду:

```
./alternetwork-db.sh add
```

Как видите, ошибка быстро доходит до 100 %, потому что служба Connections' Posts вернет ошибку сервера, если не получит результат от службы Posts, даже если повторные попытки будут реализованы. А теперь посмотрим, что происходит после метки времени 2, когда вы восстанавливаете работу сети:

```
./alternetwork-db.sh delete
```

Вы видите первые признаки восстановления всего за минуту и полное восстановление менее чем за три минуты. Вы избежали шквала повторных отправок запроса даже в самых экстремальных условиях, когда работа сети была нарушена в течение нескольких минут.

Это может заставить вас задуматься о том, представляет ли данная реализация ценность в случаях с более неустойчивыми ошибками. Давайте смоделируем эту ситуацию, отключив только одну из служб Posts от сети. Это можно сделать, выполнив одну из команд `kubectl` из скрипта `alternetwork-db.sh`. Например:

```
kubectl exec mysql-57bdb878f5-dhlck -- route $1 -host 10.36.4.11 reject
```

Мы разорвали только одно соединение от одного из экземпляров службы Posts к службе MySQL, как показано на рис. 9.5.

Взглянув на вывод журнала JMeter, видно, что хотя у службы Posts есть проблемы из-за отсутствия подключения к MySQL (начиная с метки 1), многие из возникающих в результате неудачных попыток из сообщений Connections полностью смываются повторными отправками запроса. Мы отключили от MySQL только один экземпляр службы Posts. В среднем 25 % запросов от службы Connections' Posts к службе Posts окончатся неудачно. Но, как видно из приведенного ниже вывода, общий процент ошибок намного меньше – менее 1 %. И когда связь восстанавливается на метке 2, процент ошибок тотчас же возвращается к 0,0 %:

```
START Running Jmeter on Mon Feb 18 22:16:50 UTC 2019
JVM_ARGS=-Xmn524m -Xms2096m -Xmx2096m
jmeter args=-n -t /etc/jmeter/jmeter_run.jmx -l resultsconnectionsposts
Feb 18, 2019 10:16:52 PM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
Creating summariser <summary>
Created the tree successfully using /etc/jmeter/jmeter_run.jmx
Starting the test @ Mon Feb 18 22:16:52 UTC 2019 (1550528212234)
Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445
summary +     58 in 00:00:07 =    8.2/s Err:     0 (0.00%) Active: 58
summary +    483 in 00:00:30 =   16.1/s Err:     0 (0.00%) Active: 304
summary =    541 in 00:00:37 =   14.6/s Err:     0 (0.00%)
summary +    982 in 00:00:30 =   32.7/s Err:     0 (0.00%) Active: 553
summary =   1523 in 00:01:07 =   22.7/s Err:     0 (0.00%)
summary +   1477 in 00:00:30 =   49.3/s Err:     0 (0.00%) Active: 802
summary =   3000 in 00:01:37 =   30.9/s Err:     0 (0.00%)
summary +   1974 in 00:00:30 =   65.8/s Err:     0 (0.00%) Active: 1049
summary =   4974 in 00:02:07 =   39.2/s Err:     0 (0.00%)
summary +   2473 in 00:00:30 =   82.4/s Err:     0 (0.00%) Active: 1298
summary =   7447 in 00:02:37 =   47.4/s Err:     0 (0.00%)
summary +   2920 in 00:00:30 =   97.4/s Err:     0 (0.00%) Active: 1500
summary =  10367 in 00:03:07 =   55.4/s Err:     0 (0.00%)

<time marker 1 - I have broken a single connection between Posts and MySQL>
```

```
summary +   2998 in 00:00:30 =   99.9/s Err:     3 (0.10%) Active: 1500
summary =  13365 in 00:03:37 =   61.6/s Err:     3 (0.02%)
summary +   2999 in 00:00:30 =  100.0/s Err:     0 (0.00%) Active: 1500
summary =  16364 in 00:04:07 =   66.3/s Err:     3 (0.02%)
summary +   2993 in 00:00:30 =   99.8/s Err:     1 (0.03%) Active: 1500
summary =  19357 in 00:04:37 =   69.9/s Err:     4 (0.02%)
summary +   3001 in 00:00:30 =  100.1/s Err:     1 (0.03%) Active: 1500
summary =  22358 in 00:05:07 =   72.8/s Err:     5 (0.02%)
summary +   2994 in 00:00:30 =   99.8/s Err:     1 (0.03%) Active: 1500
summary =  25352 in 00:05:37 =   75.2/s Err:     6 (0.02%)
summary +   3005 in 00:00:30 =  100.1/s Err:     2 (0.07%) Active: 1500
summary =  28357 in 00:06:07 =   77.3/s Err:     8 (0.03%)
summary +   3001 in 00:00:30 =  100.1/s Err:     1 (0.03%) Active: 1500
summary =  31358 in 00:06:37 =   79.0/s Err:     9 (0.03%)

<time marker 2 - I have repaired the connection between Posts and MySQL>

summary +   2999 in 00:00:30 =  100.0/s Err:     1 (0.03%) Active: 1500
summary =  34357 in 00:07:07 =   80.5/s Err:    10 (0.03%)
summary +   3000 in 00:00:30 =  100.0/s Err:     1 (0.03%) Active: 1500
summary =  37357 in 00:07:37 =   81.7/s Err:    11 (0.03%)
summary +   3009 in 00:00:30 =  100.3/s Err:     0 (0.00%) Active: 1500
summary =  40366 in 00:08:07 =   82.9/s Err:    11 (0.03%)
```

Вы убедились, что с помощью всего лишь нескольких простых элементов управления, ограничивающих количество попыток повторной отправки запроса и занимающих какое-то время между ними, можно реализовать преимущества повторных отправок, избегая ухудшения условий в уже неполноценной системе.

9.1.8. Когда не нужно использовать повторную отправку запроса

Вы только что отчетливо видели преимущества повторных отправок запроса, и вам следует без предрассудков использовать их при разработке своего программного обеспечения. За исключением тех случаев, когда делать этого не стоит. Будет много специфических причин, по которым вы, возможно, предпочтете не прибегать к переотправке запроса (например, использование кеширования в качестве альтернативы может повысить производительность), и я не буду освещать их здесь. Но мне хотелось бы на мгновение вернуться к теме, о которой я говорила в начале этой главы: о случаях, когда повторные отправки запроса небезопасны (например, когда вы не получаете ответа после нажатия кнопки **Купить**).

Я выбрала слово «безопасный» вполне намеренно, потому что в HTTP есть формальное определение *безопасности*, которое подходит именно к тому моменту, который я хочу сформулировать. Вот два определения из спецификации HTTP (**www.w3.org/Protocols/rfc2616/rfc2616-sec9.html**):

- *безопасный* метод – это метод, который можно вызывать *ноль* или более раз с одинаковым эффектом. Этот метод не должен иметь побочных эффектов;
- *идемпотентный* метод – это метод, при котором вызов метода один или несколько раз будет иметь одинаковый эффект. В этом случае допустимы побочные эффекты, но все повторные вызовы должны иметь тот же побочный эффект, что и первый.

То, что вы делаете с повторными отправками запроса, касается фразы «или более». Но какое из этих утверждений относится к нашей модели? Говоря кратко,

это первый вариант – когда речь идет о повторной отправке, *следует использовать только безопасные методы*. Когда вы делаете запрос в сети, нет никакой гарантии, что любой из ваших запросов достигнет предполагаемого получателя, поэтому вы можете оказаться в ситуации, когда *нулевое* число ваших попыток окажется успешным. Поэтому, как правило, следует использовать только безопасные методы. Если вы хотите реализовать обработку сбоев вокруг небезопасных методов, вы должны реализовать компенсирующее поведение, например шаблон «Сага».

Вот важный момент: вы как разработчик должны знать, безопасны ваши вызовы или нет. Возвращаясь к спецификации HTTP, видно, что безопасными HTTP-запросами являются GET, HEAD, OPTIONS и TRACE. Но Spring Retry не видит ни одного из HTTP-запросов, которые вы делаете из методов `@Retryable`, поэтому вы можете добавлять эту аннотацию только к безопасным методам. Если бы у вас был метод, который инкапсулировал POST- запрос, с помощью которого с вашего банковского счета снималось бы по 100 долл., повторная отправка запроса вас вряд ли бы обрадовала. Используйте их только тогда, когда это безопасно.

9.2. Альтернативная логика

Проектирование ради сбоя. Это мантра программного обеспечения для облачной среды, которую, я надеюсь, вы уже усвоили на протяжении чтения данной книги. Как пример повторная попытка запроса, когда первая была неудачной, – хороший вариант. Но что вы делаете, если ваши попытки восстановления также не увенчались успехом? Что происходит, когда вы повторяете попытку несколько раз и по-прежнему не получаете ответа? В предыдущих примерах этой главы мы вернули ошибку, но можно сделать еще лучше.

Одним из наиболее фундаментальных шаблонов проектирования ради сбоя является реализация альтернативных методов – кода, который выполняется при сбое основной логики. Конечно, иногда, когда программное обеспечение не может выполнить свою задачу, правильнее всего вернуть ошибку. Но в мире сильно распределенных, постоянно меняющихся программных развертываний с большим количеством случаев сбоев необходимо наращивать новые мускулы. Вам нужно выработать привычку думать об альтернативных результатах, даже если они не идеальны.

Пример, который использовался в этой главе, дает отличную возможность натренировать этот мускул, и расширение логики повторной отправки запроса является идеальным местом для этого. При проектировании вариантов альтернативного поведения (и любых шаблонов устойчивости, описанных в этой книге) необходимо продумать реальный сценарий, к которому обращается ваше программное обеспечение. В той части реализации, которую вы будете расширять, вы пытаетесь получить список постов в блоге от какого-то числа пользователей. Хотя некоторые из этих пользователей могут быть довольно плодовитыми, и, возможно, они публикуют посты по нескольку раз в неделю или даже чаще одного раза в день, создание новых постов в блоге по-прежнему происходит довольно редко. Если бы пользователю нужно было получить доступ к своей сводной ленте новостей в то время, когда база данных MySQL, в которой хранятся сообщения, была недоступна, возможно, лучше было бы вернуть набор постов, в которых могут отсутствовать только самые новые записи, вместо того чтобы не возвращать

вообще ничего. Лично я могу сказать вам, что когда я получаю доступ к сводному набору рецептов, чтобы решить, что приготовить на ужин сегодня вечером, я все равно могу приготовить что-нибудь довольно вкусное, даже если у меня нет самого последнего рецепта, который выложил пользователь с ником Food52.

9.2.1. Давайте посмотрим, как это работает: реализация альтернативной логики

Давайте посмотрим, как это работает на практике. Скачайте приведенный тег из репозитория:

```
git checkout requestretries/0.0.4
```

Здесь я не буду повторять инструкции по необязательной сборке. Пожалуйста, посмотрите предыдущие примеры из этой главы (и книги), если вам нужно изменить код и выполнить развертывание самостоятельно.

Как и всегда, я все предварительно собрала и сделала доступными образы Docker в Docker Hub.

Перед тестированием давайте посмотрим на реализацию. В случае когда служба Posts не дает действительного результата, служба Connections' Posts будет просто возвращать самые последние посты, которые она видела ранее. Для этого мы добавили в реализацию простое кеширование. Напомню, что реализация службы Connections' Posts уже привязана к хранилищу Redis типа «ключ/значение», базе данных, идеально подходящей для кеширования.

Поэтому теперь, когда вызов Posts дает результат, логика в Connections' Posts будет сохранять этот результат в Redis, перед тем как вернуть его. Это хранилище настраивает вас на то, чтобы иметь возможность затем реализовать альтернативное поведение, когда служба Posts работает плохо. В верхней части рис. 9.6 показан поток, который кеширует результаты, когда служба Posts доступна и доставляет результаты. В нижней части рис. 9.6 показан поток, который читает результаты из кеша, когда служба Posts испытывает проблемы.

Тогда добавить альтернативную реализацию просто. С помощью Spring Retry вы добавляете метод в службу с аннотацией `@Recover`, и Spring будет вызывать этот метод после того, как все попытки повторной отправки запроса будут исчерпаны. Сигнатура метода должна совпадать с сигнатурой метода, реализующего основную логику, с добавлением типа исключения в качестве первого аргумента. Метод восстановления будет вызываться только при определенных обстоятельствах, как было определено типом ошибки.

Листинг 9.5 ❖ Метод из файла PostsServiceClient.java

```
@Recover
public ArrayList<PostSummary> returnCached(
                              ResourceAccessException e,
                              String ids, RestTemplate restTemplate)
                                                    throws Exception {
  logger.info("Failed ... Posts service - returning cached results");

  PostResults postResults = postResultsRepository.findOne(ids);
  ObjectMapper objectMapper = new ObjectMapper();
  ArrayList<PostSummary> postSummaries;
```

```
  try {
    postSummaries = objectMapper.readValue(
                          postResults.getSummariesJson(),
                          new TypeReference<ArrayList<PostSummary>>() {});
  } catch (Exception ec) {
    logger.info("Exception on deserialization " + ec.getClass()
                + " message = " + ec.getMessage());
    return null;
  }
  return postSummaries;
}
```

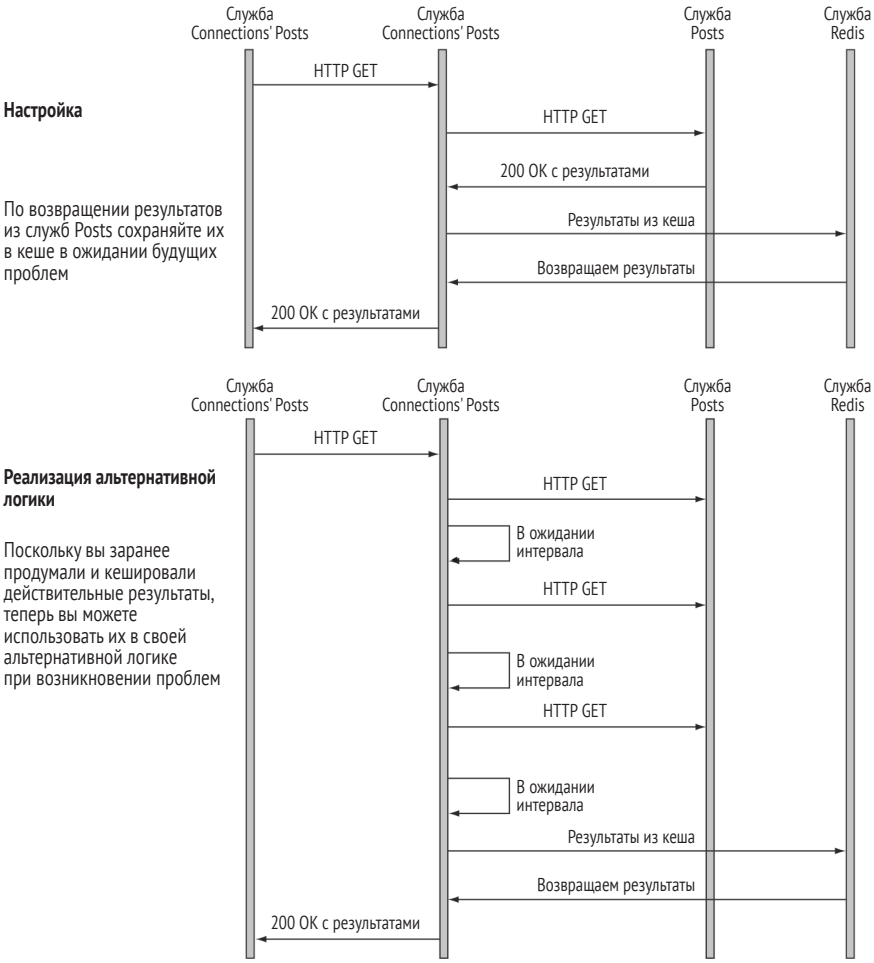


Рис. 9.6 ❖ Думая наперед, вы кешируете результаты, когда они доступны, поэтому, когда позже вы столкнетесь с проблемами, у вас будут эти кешированные значения, которые будут использоваться как часть альтернативной логики

Хотя это и может казаться очевидным, когда вы смотрите на этот простой пример, я хочу обратить ваше внимание на тот факт, что в большинстве случаев ваше альтернативное поведение требует некоторой настройки. В этом примере предыдущий код – это не все, что требуется для альтернативы. Для этого логику, которая кеширует результаты, когда они успешно получены, необходимо настроить.

Наш метод `@Retryable` – это место, где вы думаете наперед, прежде чем наступят мрачные дни.

Листинг 9.6 ❖ Метод из файла PostsServiceClient.java

```
@Retryable( value = ResourceAccessException.class,
            maxAttempts = 3, backoff = @Backoff(delay = 500))
public ArrayList<PostSummary> getPosts(String ids,
                                       RestTemplate restTemplate)
                                                    throws Exception {

  ArrayList<PostSummary> postSummaries = new ArrayList<PostSummary>();

  String secretQueryParam = "&secret=" + utils.getPostsSecret();

  logger.info("Trying getPosts: " + postsUrl + ids + secretQueryParam);

  ResponseEntity<ConnectionsPostsController.PostResult[]> respPosts
    = restTemplate.getForEntity(
        postsUrl + ids + secretQueryParam,
        ConnectionsPostsController.PostResult[].class);
  if (respPosts.getStatusCode().is5xxServerError()) {
    throw new HttpServerErrorException(respPosts.getStatusCode(),
                "Exception thrown in obtaining Posts");
  } else {
    ConnectionsPostsController.PostResult[] posts = respPosts.getBody();
    for (int i = 0; i < posts.length; i++)
      postSummaries.add(
        new PostSummary(getUsersname(posts[i].getUserId(), restTemplate),
        posts[i].getTitle(), posts[i].getDate()));
    // Думая наперед, кешируем результат;
    ObjectMapper objectMapper = new ObjectMapper();
    String postSummariesJson =
             objectMapper.writeValueAsString(postSummaries);
    PostResults postResults = new PostResults(ids, postSummariesJson);
    postResultsRepository.save(postResults);
    return postSummaries;
  }
}
```

Давайте теперь посмотрим на то, как добавление альтернативного поведения влияет на стабильность нашей реализации. Мы выполним тот же нагрузочный тест, что и до этого.

Если вы хотите продолжить, обновите свое развертывание, повторно запустив скрипт развертывания приложения:

```
./deployApps.sh
```

Теперь запускаем нагрузочный тест с помощью нашей обычной команды:

```
kubectl create -f loadTesting/jmeter-deployment.yaml
```

Как и всегда, после того как тест достиг полной емкости, вы прерываете работу сети между всеми экземплярами службы Posts и MySQL на 3 минуты (метка времени 1), а затем восстанавливаете ее (метка времени 2). Прежде чем посмотреть на результаты тестирования, давайте посмотрим на вывод журнала одной из служб Connections' Posts:

```
(the network is currently broken...)
... : [10.36.4.11:8080] getting posts for user network cdavisafc
... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret
... : Failed to connect to or obtain results from Posts service - returning
    cached results
... : Failed to connect to or obtain results from Posts service - returning
    cached results
... : [10.36.4.11:8080] connections = 2,3
... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret
... : Failed to connect to or obtain results from Posts service - returning
    cached results

(after restoring the network)

... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret
... : [10.36.4.11:8080] getting posts for user network cdavisafc
... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret
... : [10.36.4.11:8080] connections = 2,3
... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret
... : [10.36.4.11:8080] getting posts for user network cdavisafc
... : [10.36.4.11:8080] connections = 2,3
... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret
... : [10.36.4.11:8080] getting posts for user network cdavisafc
... : [10.36.4.11:8080] connections = 2,3
... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret
... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret
```

Пока работа сети нарушена, Spring Retry сначала повторяет попытку доступа три раза, а затем вызывает метод `@Recover`, возвращая кешированные результаты. После восстановления работы сети текущие результаты снова возвращаются.

Давайте теперь посмотрим, как эта реализация работает с нагрузкой. Ниже приводится вывод журнала из теста с JMeter:

```
START Running Jmeter on Mon Feb 18 23:10:22 UTC 2019
JVM_ARGS=-Xmn506m -Xms2024m -Xmx2024m
jmeter args=-n -t /etc/jmeter/jmeter_run.jmx -l resultsconnectionsposts
Feb 18, 2019 11:10:24 PM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
Creating summariser <summary>
Created the tree successfully using /etc/jmeter/jmeter_run.jmx
Starting the test @ Mon Feb 18 23:10:24 UTC 2019 (1550531424214)
Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445
summary +    194 in 00:00:19 =   10.0/s Err:     0 (0.00%) Active: 159
summary +    687 in 00:00:30 =   22.9/s Err:     0 (0.00%) Active: 406
summary =    881 in 00:00:49 =   17.8/s Err:     0 (0.00%)
summary +   1184 in 00:00:30 =   39.5/s Err:     0 (0.00%) Active: 655
summary =   2065 in 00:01:19 =   26.0/s Err:     0 (0.00%)
summary +   1682 in 00:00:30 =   56.1/s Err:     0 (0.00%) Active: 904
```

```
summary =   3747 in 00:01:49 =   34.2/s Err:     0 (0.00%)
summary +   2176 in 00:00:30 =   72.6/s Err:     0 (0.00%) Active: 1151
summary =   5923 in 00:02:19 =   42.5/s Err:     0 (0.00%)
summary +   2676 in 00:00:30 =   89.2/s Err:     0 (0.00%) Active: 1400
summary =   8599 in 00:02:49 =   50.8/s Err:     0 (0.00%)
summary +   3000 in 00:00:30 =  100.0/s Err:     0 (0.00%) Active: 1500
summary =  11599 in 00:03:19 =   58.2/s Err:     0 (0.00%)

<time marker 1 - I have broken the network between Posts and MySQL>

summary +   2752 in 00:00:30 =   91.7/s Err:     0 (0.00%) Active: 1500
summary =  14351 in 00:03:49 =   62.6/s Err:     0 (0.00%)
summary +   3000 in 00:00:30 =   99.9/s Err:     0 (0.00%) Active: 1500
summary =  17351 in 00:04:19 =   66.9/s Err:     0 (0.00%)
summary +   3001 in 00:00:30 =  100.1/s Err:     0 (0.00%) Active: 1500
summary =  20352 in 00:04:49 =   70.3/s Err:     0 (0.00%)
summary +   2998 in 00:00:30 =   99.9/s Err:     0 (0.00%) Active: 1500
summary =  23350 in 00:05:19 =   73.1/s Err:     0 (0.00%)
summary +   3038 in 00:00:30 =  101.3/s Err:     0 (0.00%) Active: 1500
summary =  26388 in 00:05:49 =   75.5/s Err:     0 (0.00%)
summary +   3039 in 00:00:30 =  101.3/s Err:     0 (0.00%) Active: 1500
summary =  29427 in 00:06:19 =   77.6/s Err:     0 (0.00%)
summary +   3000 in 00:00:30 =  100.0/s Err:     0 (0.00%) Active: 1500
summary =  32427 in 00:06:49 =   79.2/s Err:     0 (0.00%)

<time marker 2 - I have repaired the network between Posts and MySQL>

summary +   3089 in 00:00:30 =  102.9/s Err:     0 (0.00%) Active: 1500
summary =  35516 in 00:07:19 =   80.8/s Err:     0 (0.00%)
summary +   3080 in 00:00:30 =  102.7/s Err:     0 (0.00%) Active: 1500
summary =  38596 in 00:07:49 =   82.2/s Err:     0 (0.00%)
```

Это именно то, чего вы и ожидали. Поскольку Connections' Posts возвращает кешированные результаты, когда зависимая служба Posts не дает результатов, клиент Connections' Posts (JMeter) не видит ошибки во время сбоя. Тем не менее, как вы видели в предыдущем журнале Connections' Posts, как только работа сети восстанавливается, текущие значения возвращаются.

> **ПРИМЕЧАНИЕ** Выглядит довольно надежно. Пользователи вашего программного обеспечения *не* видели ошибок, даже когда некоторые части системы испытывали проблемы. Проблемы в одной из частей вашего программного обеспечения не обрушились потоком на распределенную систему.

Возвращаясь к серии тестов, которые вы только что выполнили, обобщим результаты в табл. 9.1.

Как ясно из этой таблицы, если в противном случае вы будете использовать хороший шаблон для облачной среды слишком упрощенным способом, это может создать дополнительные проблемы. Но, используя простую повторную отправку запроса с другими шаблонами, негативные последствия могут быть значительно уменьшены или даже полностью устранены.

Как вы видите, различные варианты компенсирующего поведения, которые вы реализовали, оказали огромное положительное влияние на стабильность системы и взаимодействие с пользователем.

Проектирование ради сбоя имеет значение!

Таблица 9.1. Использование простой повторной отправки запроса с другими шаблонами может привести к устранению или уменьшению негативных воздействий

Версия службы Connections' Posts	Во время отключения сети	Время появления начальных признаков восстановления	Время полного восстановления
Простая повторная отправка запроса	100 % ошибок	9 минут	12–13 минут
Мягкая повторная отправка с использованием Spring Retry при отсутствии альтернативной логики	100 % ошибок	1 минута	3 минуты
Мягкая повторная отправка с использованием Spring Retry и альтернативного метода	0,0 % ошибок	Н/П – нет сбоя во время отключения сети	Н/П

Теперь, когда мы реализовали несколько шаблонов на клиентской стороне взаимодействия, давайте посмотрим на начальную диаграмму этой главы и внесем несколько деталей. На рис. 9.7 видно, что на стороне клиента реализованы и повторные попытки, и альтернативное поведение.

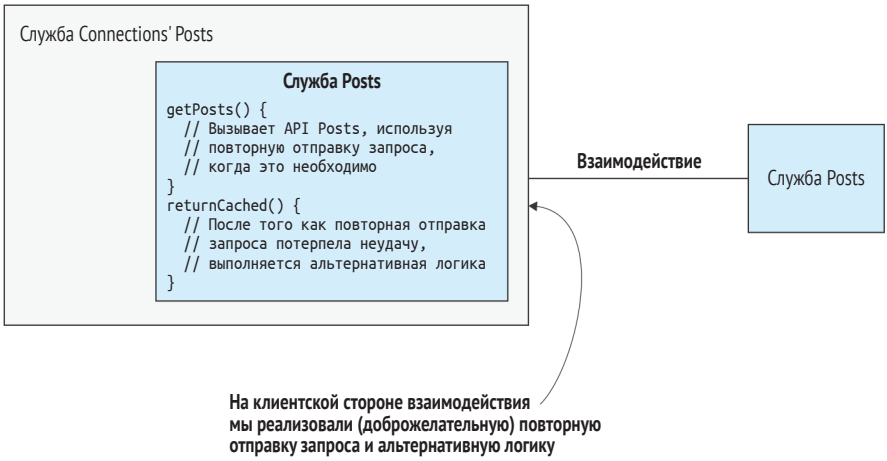


Рис. 9.7 ❖ Реализация шаблонов на стороне клиента, таких как повторная отправка запроса и альтернативы, дает гораздо более надежную систему (мы перейдем к взаимодействию на стороне службы в следующей главе)

В главе 10 мы перейдем к другому концу этого взаимодействия.

9.3. Циклы управления

Несмотря на их кажущуюся простоту, я потратила много времени на освещение повторных отправок запроса по двум причинам. Во-первых, они являются важным инструментом для построения отказоустойчивых распределенных систем, и, как вы видели, их не так-то просто использовать правильно. Но что еще более важно, я использую их как конкретный пример более общего шаблона, о котором хочу сейчас рассказать подробнее: циклы управления.

9.3.1. Типы циклов управления

Повторные отправки запросов, которые вы изучали здесь,– не первый пример избыточных действий, о которых идет речь в этой главе, хотя до сих пор я лишь кратко упоминала о них. Например, развертывая свои приложения в окружении Kubernetes, вы используете, по крайней мере, один из циклов управления, встроенных в эту платформу: контроллер репликации. Контроллер репликации Kubernetes реализует цикл управления, который позволяет декларативно указать развертывание приложения, а Kubernetes будет создавать и поддерживать топологию приложения. Цикл управления не ожидает достижения состояния *готовности*. Он предназначен для постоянного поиска неизбежных изменений и адекватного реагирования.

Эта книга не о Kubernetes, поэтому я не буду подробно останавливаться на них, но контроллеры постоянно (в цикле управления) сравнивают фактическое состояние рабочих нагрузок, работающих в кластере Kubernetes, с желаемым состоянием этих нагрузок, которое он получает (в цикле управления) от сервера API Kubernetes источника истины для желаемого состояния кластера. Вот небольшая выборка некоторых циклов управления, реализуемых этой платформой:

- *контроллер репликации* – этот контроллер управляет развертываниями (см. файлы YAML наших развертываний приложений), гарантируя, что необходимое количество реплик остается в рабочем состоянии после сбоев и обновлений;
- *набор демонов* – набор демонов Kubernetes определяет модули; ровно по одному будет работать на каждом рабочем узле (физической или виртуальной машине) кластера Kubernetes. Этот контроллер гарантирует, что на всех узлах развернуты все нужные наборы демонов;
- *контроллер конечных точек* – по мере динамического развертывания рабочих нагрузок и назначения IP-адресов контроллер конечных точек будет обновлять DNS-службу Kubernetes (помимо прочего);
- *контроллер пространств имен* – пространства имен могут использоваться в качестве арендатора в кластере Kubernetes, и при их создании к ним могут применяться определенные политики. Например, можно создать сегмент сети и назначить его для изоляции сетевого трафика для приложений, развернутых в этом пространстве имен. Контроллер пространств имен отслеживает изменения в списке пространств имен Kubernetes и выполняет любые необходимые действия.

Позвольте мне вернуться на мгновение к последнему примеру. Я говорю о *циклах* управления. Зачем нам цикл? Разве наша система не может просто выполнить необходимые действия, когда, например, кто-то выполняет команду `kubectl create namespace`? В теории да. Но, как вы видели в примере в первой части этой главы, вполне возможно, что этот код может быть недоступен при выполнении команды. Если это произойдет, вам не удастся создать пространство имен? Вы автоматически будете выполнять повторную отправку запроса раз или два? Шаблон контроллера явно предназначен для решения таких типов проблем, и он делает это настолько хорошо, что его использование распространено по всей современной распределенной системе, такой как Kubernetes. Поэтому вам также следует свободно использовать его в своем программном обеспечении для облачной среды.

9.3.2. Контроль над циклом управления

Ранее в этой главе я говорила об управлении циклом повторной отправки запроса. Элементы управления, которые вы применяли, ограничивали количество выполнений цикла, контролировали его частоту и выбирали условия, при которых должно быть инициировано действие (тип исключения). Опять-таки, так же как цикл повторной отправки запроса распространяется на базовый цикл управления, это же делают и некоторые параметры, которые вы можете применить к ним.

Например, тип исключения, к которому применяется метод `@Retryable`, похож на тип данных. Обратите внимание, что контроллеры Kubernetes, которые я перечислила ранее, применялись к различным типам объектов Kubernetes. Хотя в общем смысле циклы контроллера бесконечны, вполне приемлемо изменить этот принцип, как вы это делали, когда ограничивали общее количество повторных попыток для определенного удаленного запроса. И наконец, давайте посмотрим на частоту, с которой выполняются действия в результате цикла управления.

Говоря о повторных отправках запросов, я привела вам пример; вы снизили частоту отправок (до половины секунды), но сохранили интервал. Возможно, вы заметили аннотацию `@Backoff` в объявлении `@Retryable`. Данная аннотация намекает на возможность настройки алгоритма отката с возвратом. Вы можете реализовать любую линейную или нелинейную политику отката по вашему выбору, но в Spring Retry уже есть ряд распространенных встроенных политик. Вы уже видели пример линейной политики, когда ждали полсекунды между попытками повторной отправки запроса. Позвольте мне теперь показать вам нелинейный пример в действии. Для начала, если вы еще этого не сделали, скачайте приведенный ниже тег из своего репозитория в Git:

```
git checkout requestretries/0.0.5
```

Манифесты развертывания приложения описывают менее крупное развертывание нашего программного обеспечения, подходящего для развертывания в вашем кластере Minikube. Вы можете развернуть соответствующую версию примера, выполнив bash-скрипт `deployApps.sh`. Несмотря на то что вы, безусловно, можете отправлять команды `curl` в службу Connections' Posts на этом этапе, я хочу сосредоточиться на поведении контроллера репликации Kubernetes – цикле управления, который наблюдает за состоянием развертываний вашего приложения и поддерживает его.

Посмотрите на вывод команды `kubectl get pods`. Мне бы хотелось, чтобы вы наблюдали за ним постоянно, поэтому наберите команду `watch kubectl get pods`. Вы увидите нечто вроде этого:

```
$ kubectl get pods

NAME                                READY STATUS  RESTARTS  AGE
connection-posts-67d8db4c7b-tscf8   1/1   Running 0         10h
connections-748dc47cc6-7bzzr        1/1   Running 0         10h
mysql-64bd6d89d8-ggwss              1/1   Running 0         1d
posts-649d88dff-kmmx8               1/1   Running 0         9h
redis-846b8c56fb-8k8f7              1/1   Running 0         1d
sccs-84cc988f57-fjhzx               1/1   Running 0         1d
```

Это новая установка с одним экземпляром каждого из наших образцов микросервисов. Теперь я хочу, чтобы вы инфицировали службу Posts; вы сделаете ее неработоспособной. Это можно сделать, выполнив приведенную ниже команду:

```
curl -i -X POST $(minikube service --url posts-svc)/infect
```

А теперь просто следите за выводом команды `get pods`. Примерно через 15–30 секунд вы увидите, что служба Posts перезапускается. Если это происходит быстро, можно заметить только то, что счетчик в столбце RESTARTS увеличивается:

```
$ kubectl get pods

NAME                                READY STATUS  RESTARTS AGE
connection-posts-67d8db4c7b-tscf8   1/1   Running 0        10h
connections-748dc47cc6-7bzzr        1/1   Running 0        10h
mysql-64bd6d89d8-ggwss              1/1   Running 0        1d
posts-649d88dff-kmmx8               1/1   Running 1        9h
redis-846b8c56fb-8k8f7              1/1   Running 0        1d
sccs-84cc988f57-fjhzx               1/1   Running 0        1d
```

Теперь заразите этот экземпляр снова, выполнив ту же команду `curl`. Примерно через 15–30 секунд вы увидите, что приложение снова перезапускается. Продолжайте заражать его каждый раз, когда оно возвращается. После того как вы проделаете это четыре или пять раз, вместо перезапуска приложения вы увидите надпись `CrashLoopBackOff`. Контроллер репликации заметил, что приложение несколько раз становилось неработоспособным, и подождет еще немного, прежде чем предпринять очередную попытку перезапуска. Он реализовал нелинейную политику отката с возвратом.

```
$ kubectl get pods

NAME                                READY STATUS           RESTARTS AGE
connection-posts-67d8db4c7b-tscf8   1/1   Running          0        10h
connections-748dc47cc6-7bzzr        1/1   Running          0        10h
mysql-64bd6d89d8-ggwss              1/1   Running          0        1d
posts-649d88dff-kmmx8               1/1   CrashLoopBackOff 5        9h
redis-846b8c56fb-8k8f7              1/1   Running          0        1d
sccs-84cc988f57-fjhzx               1/1   Running          0        1d
```

Я надеюсь, что из этого диалога и всей главы в целом вы поняли, что необходимо приступать к поиску циклов управления при разработке программного обеспечения для облачной среды. Хотя многим из нас императивный стиль программирования кажется естественным, проблем в распределенной системе предостаточно. Поначалу они могут не проявлять себя, но за кажущимися разумными реализациями скрываются крайние случаи, представляющие опасность, и реализации, которые терпят неудачу, когда происходят неожиданные изменения. Проектирование согласованного в конечном счете программного обеспечения на базе циклов управления намного лучше подойдет для распределенных систем, из которых состоит наше ПО для облачной среды.

Резюме

- Повторная отправка запросов с истечением времени ожидания может поглотить ошибки, которые в противном случае распространялись бы по системе.
- Если что-то будет сделано неправильно, очередь повторных отправок может перегрузить систему даже после устранения проблем с подключением;
- Правильно сконфигурированные повторные отправки запросов могут значительно снизить риск шквалов подобных отправок, при этом обеспечивая существенные преимущества при менее серьезных отключениях.
- Будучи разработчиком, вы должны использовать повторные отправки запросов, только когда это *безопасно*.
- Вы должны выработать привычку реализовывать не только основной поток своего сервиса, но и логику возврата к рабочему состоянию, на случай если подход с использованием счастливого пути окажется неудачным.
- Повторная попытка запроса является всего лишь одним из примеров шаблона цикла управления.
- Циклы управления – это важный метод для распределенных систем, из которых состоит ПО для облачной среды.

Глава 10

Лицом к лицу с сервисами: предохранители и API-шлюзы

О чем идет речь в этой главе:

- сервисная сторона взаимодействия двух микросервисов;
- предохранители;
- API-шлюзы;
- сайдкары и сервисная сетка.

Я начала говорить о взаимодействиях между службами в главе 8 с акцентом на динамическую маршрутизацию и обнаружение служб; я говорила о том, как клиенты могут найти и получить доступ к службе, от которой они зависят. После того как клиент находит необходимый сервис и получает доступ к нему, он инициирует взаимодействие. В этой и предыдущей главах рассматриваются обе стороны этого взаимодействия – как показано на рис. 10.1. В главе 9 я рассказала об устойчивости данного взаимодействия; в основном я говорила об избыточности запросов, о том, за что отвечает клиент и к чему применяет элементы управления. Теперь я хочу перейти к сервисной стороне взаимодействия «клиент/сервис» и к основным шаблонам проектирования, которые здесь играют роль.

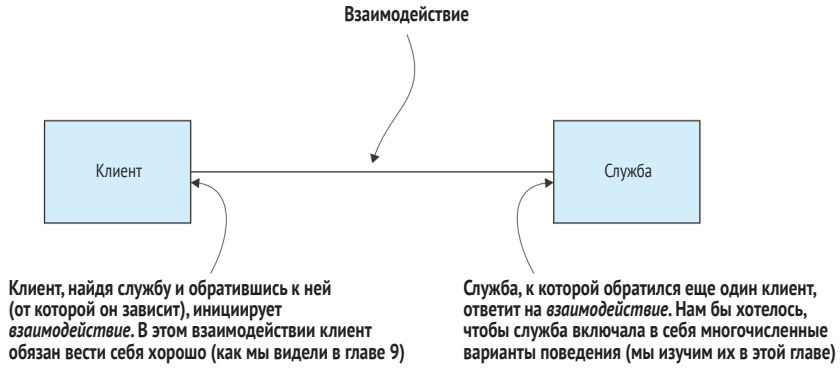


Рис. 10.1 ❖ Равно как клиент может и должен реализовать определенные шаблоны, чтобы действовать как полноценный участник взаимодействия, так же должна поступать и служба. Это шаблоны, о которых идет речь в данной главе

Будучи разработчиком сервиса, вы должны учитывать множество проблем, связанных со взаимодействием:

- в предыдущей главе я представила решение проблемы шквала повторных отправок запроса, которая была реализована на клиентской стороне взаимодействия (*доброжелательные повторные отправки*, как я их назвала). Но разработчик сервиса не может зависеть от того, чтобы клиенты всегда были добрыми, поэтому он должен остерегаться подобных шквалов. С точки зрения сервиса, шквал повторных отправок запросов – это просто случай получения бóльшего количества входящих запросов, чем он может обработать. Сервис в конечном итоге несет ответственность за свою защиту от преднамеренных или непреднамеренных атак типа «отказ в обслуживании»;
- ранее я говорила о методах развертывания новой версии службы, в частности о сине-зеленых развертываниях и последовательных обновлениях. Как вы помните, я также рассказала о параллельном развертывании, при котором несколько версий службы выполняется одновременно, причем некоторые запросы обслуживаются одной службой, а некоторые – другой. В большинстве случаев решения о том, какая версия сервиса должна отвечать на данный запрос, обрабатываются на серверной стороне взаимодействия «клиент/сервис»;
- сервис должен отвечать только на запросы авторизованных сторон;
- он также отвечает за предоставление информации о мониторинге и журналировании (об этом речь пойдет в следующей главе).

В этой главе рассматриваются два шаблона, которые решают эти проблемы: предохранители и API-шлюзы. Предохранители явно нацелены на первую из числа этих проблем и используются для защиты службы от чрезмерного трафика. API-шлюзы используются для решения всех этих проблем. Хотя API-шлюзы используются уже на протяжении некоторого периода времени, я особо остановлюсь на потребностях архитектур для облачной среды, которые стали использоваться совсем недавно.

В завершение главы речь пойдет о ставшем недавно популярном подходе к реализации шаблонов как на стороне сервера, так и на стороне клиента: сайдкарах. Да, я буду говорить об Istio и его друзьях.

10.1. Предохранители

Концепция *предохранителя* для вашего программного обеспечения точно такая же, как и в электросети у вас дома. В вашем доме есть вещи, которые питаются от электричества: освещение, розетки, бытовые приборы и т. д. Чем больше энергии одновременно подается на ваши провода, тем горячее они становятся, а если нагрузка будет слишком сильной, провода могут нагреться достаточно для того, чтобы осветить стены, в которых они горят. Чтобы этого не произошло, провода пропускаются через предохранитель, который определяет, когда потребление энергии становится опасно высоким, и размыкает цепь, поэтому все электричество отключается. Лучше сидеть без света, чем поджечь дом.

10.1.1. Предохранитель для программного обеспечения

В вашем программном обеспечении предохранители работают, по сути, так же. Когда нагрузка слишком высокая, цепь разомкнута и препятствует прохождению трафика. Но есть два различия. Во-первых, механизм определения того, когда цепь должна размыкаться, основан на фактических сбоях, а не на прогнозировании возможных (вам не нужно, чтобы электрическая цепь отключалась только после обнаружения небольшого пожара). И во-вторых, у предохранителя в программном обеспечении обычно имеется встроенный механизм самовосстановления (в отличие от обесточенного дома, когда вы в темноте ищите электрическую панель, чтобы вручную щелкнуть выключателем).

Основная идея заключается в следующем: если в службе происходит серьезный сбой, вы на некоторое время останавливаете весь трафик, идущий к этой службе, надеясь дать ей время на восстановление. Затем, спустя некоторое время, вы проверяете, как обстоят дела, пропуская один запрос. Если этот запрос не выполняется, вы оставляете защиту, не допуская дальнейшего трафика. Если запрос выполнен успешно, вы рассматриваете службу как исправную и позволяете трафику снова свободно идти дальше.

Можно смоделировать это поведение, определив три состояния, в которых может находиться предохранитель (замкнут, разомкнут или наполовину разомкнут), как показано на рис. 10.2. Затем мы можем описать события, которые вызывают изменения в состоянии, следующим образом:

- идеальное состояние вашего предохранителя – *замкнуто*: трафик идет по цепи, к службе, которую защищает цепь;
- предохранитель находится в этом потоке трафика и выявляет сбои. Небольшое количество отказов не является проблемой; действительно, устойчивость к таким «скачкам» – часть правильного проектирования ПО для облачной среды. Когда частота отказов становится слишком высокой, состояние предохранителя переходит в положение «разомкнуто»;
- пока предохранитель разомкнут, трафик к сервису, который находится под защитой переключателя, идти не будет. Если в работе сервиса стали происходить сбои из-за того, что он был перегружен запросами, или проблемы были вызваны прерывистым отключением сети, перерыв в обработке нагрузки может позволить сервису вернуться в рабочее состояние;
- по прошествии некоторого количества времени вы хотите снова опробовать сервис, чтобы проверить, восстановился ли он. Для этого вы переводите предохранитель в состояние «наполовину разомкнуто»;
- в этом состоянии реализация предохранителя будет тестировать сервис, пропуская один или небольшое количество запросов;
- если тестовые запросы выполнены успешно, предохранитель вернется в состояние «замкнуто». Если тест не будет пройден, он вернется в состояние «разомкнуто» и подождет еще немного.

Я описала работу предохранителя интуитивно, но вам и/или его реализации необходимо конкретно определить особенности изменений состояния. Например, что означает «слишком много сбоев»? Через мгновение вы увидите конкретную реализацию и рассмотрите эти детали. Вначале я хочу обратить ваше внимание на одну концепцию, которая не изображена на этой диаграмме: как влияет ис-

пользование предохранителя на взаимодействие типа «клиент/сервис», которое является очень важным в этой и предыдущей главах.

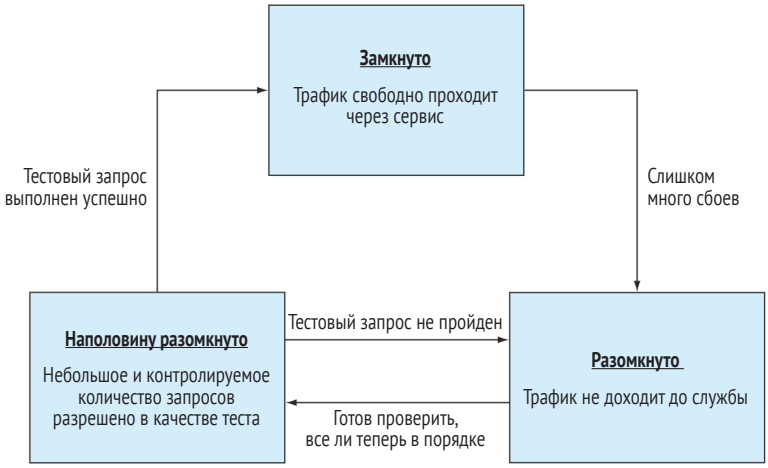


Рис. 10.2 ❖ Вы моделируете работу предохранителя с помощью трех состояний и определяете условия или события, которые вызывают переходы между ними. В замкнутом состоянии трафик течет свободно. В разомкнутом состоянии запросы не доходят до службы. Наполовину разомкнутое состояние является временным средством, с помощью которого можно перейти в замкнутое состояние

На диаграмме последовательности на рис. 10.3 показаны три сценария одного взаимодействия между клиентом и сервисом в тот момент, когда сервис испытывает проблемы. В первом случае предохранитель не используется. Во втором у вас есть предохранитель в состоянии «замкнуто». Наконец, у вас есть предохранитель в состоянии «разомкнуто».

В первых двух случаях видно, что поведение фактически идентично: клиент делает запрос, и из-за проблем, с которыми сталкивается сервис, во время ожидания ответа происходит тайм-аут. Но в последнем случае, когда мы видим состояние «разомкнуто», потому что предохранитель обнаружил неисправность, клиент быстро получит ответ. Ключевым моментом здесь является то, что задержки являются катастрофическими в сложной распределенной системе, и предохранитель значительно уменьшает их длину и частоту. Мне нравится рассматривать предохранители как шаблон «доброты», реализуемый на стороне сервиса.

Давайте теперь посмотрим на реализацию предохранителя в нашем примере, которая продемонстрирует базовое использование и конфигурируемость и позволит вам немного глубже поразмыслить над структурой реализаций ваших сервисов.

10.1.2. Реализация предохранителя

Как обычно, вы можете запустить примеры кода с помощью репозитория Git, воспользовавшись двумя конкретными тегами для приведенных здесь примеров и развернув кластер Kubernetes. Как и прежде, я создала примеры кода и объеди-

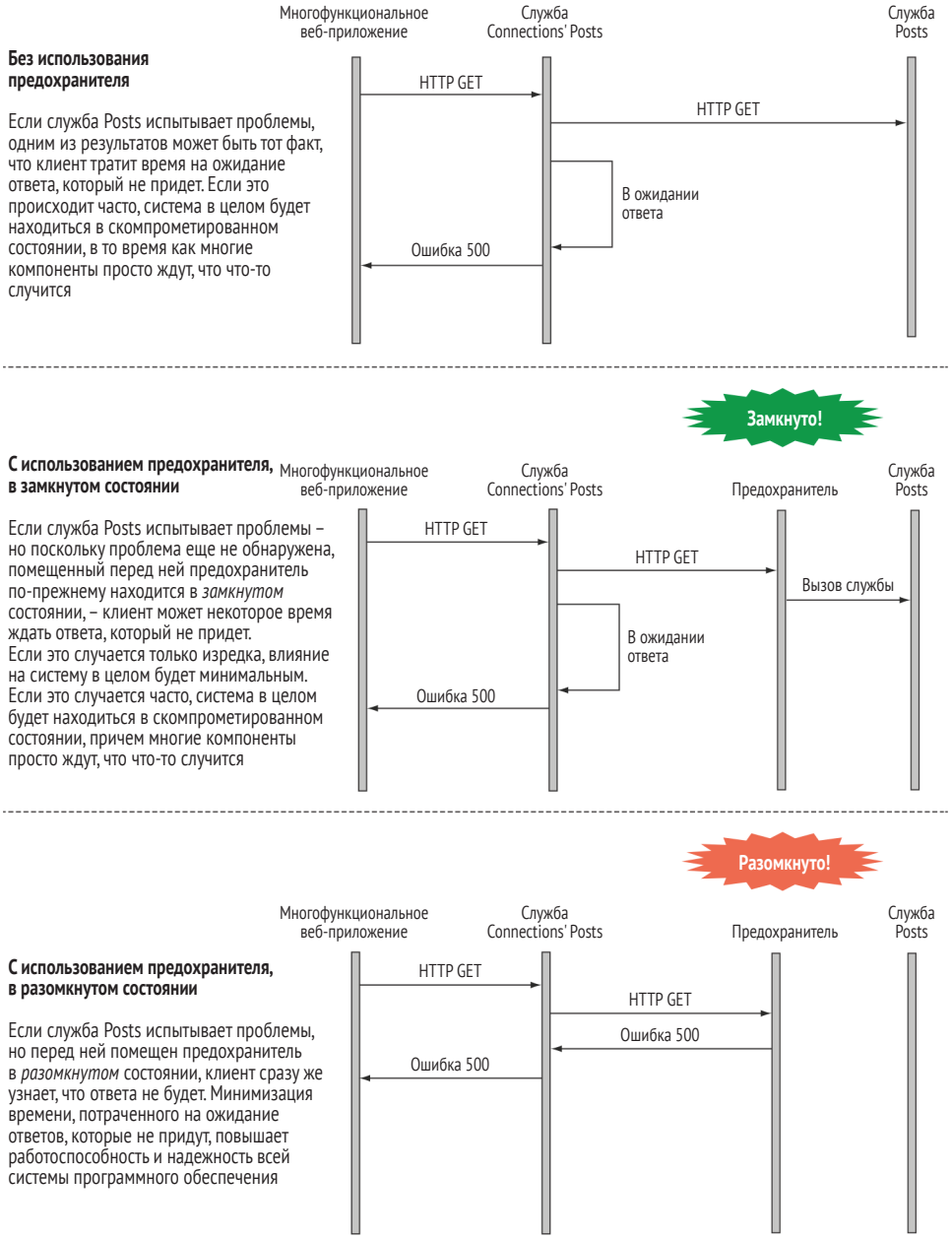


Рис. 10.3 ❖ В тех случаях, когда служба недоступна из-за перебоев в работе сети, проблем с самой службой или чего-либо еще, одним из основных преимуществ предохранителей является то, что они значительно сокращают время, затрачиваемое на ожидание ответов, которые в тот момент вряд ли придут

нила их в образах Docker, которые доступны в Docker Hub, поэтому вам не нужно собирать код из исходника. Если вы все же хотите это сделать, я включила сюда файлы сборки Maven и Docker, чтобы вам было удобно. Перед запуском примеров давайте взглянем на код.

Предполагая, что вы уже клонировали репозиторий, скачайте свой первый тег для главы 10 с помощью приведенной ниже команды:

```
git checkout circuitbreaker/0.0.1
```

Весь код находится в каталоге cloudnative-circuitbreaker, поэтому перейдите к нему сейчас. Вы заметите, что здесь есть реализации только для служб Posts и Connections, поскольку служба Connections' Posts является клиентской стороной взаимодействия и не отличается от той версии, что была в предыдущей главе.

Служба, защиту которой вы будете обеспечивать с помощью предохранителя, – это служба Posts, поэтому давайте начнем с просмотра кода в исходном каталоге этой службы. Первое, что вы заметите, – это новый класс Java: `PostsService`. Предохранитель стоит перед фактической службой, и таким образом у вас есть предохранитель, работающий в том же процессе, что и реализация основного сервиса; предохранитель находится рядом с фактической службой (внимание – я объясню это подробнее, когда мы будем говорить об Istio в конце этой главы).

Изначально у вас было много логики в самом контроллере Posts. Но то, что вы сделали сейчас, – это поместили ядро реализации сервиса в новый класс `PostsService`, и теперь у вас есть контроллер, который обрабатывает только передний край взаимодействия сервиса. Контроллер по-прежнему обрабатывает такие задачи, как парсинг запросов и генерация ответов, а также некоторая базовая логика аутентификации и авторизации. Новый класс `PostsService` не работает с протоколом HTTP, а вместо этого сосредоточен только на основной логике службы, которая в нашем простом примере представляет собой только запрос к базе данных и генерацию объекта – ответа.

Наиболее подходящим для нашего обсуждения является добавление аннотации вокруг метода `get`, как показано в листинге 10.1.

Листинг 10.1 ❖ Метод из файла PostsService.java

```
@HystrixCommand()
public Iterable<Post> getPostsByUserId(String userIds,
                                       String secret) throws Exception {

    logger.info(utils.ipTag() + “Attempting getPostsByUserId”);
    Iterable<Post> posts;

    if (userIds == null) {
        logger.info(utils.ipTag() + “getting all posts”);
        posts = postRepository.findAll();
        return posts;
    } else {
        ArrayList<Post> postsForUsers = new ArrayList<Post>();
        String userId[] = userIds.split(“,”);
        for (int i = 0; i < userId.length; i++) {
            logger.info(utils.ipTag() +
                        “getting posts for userId “ + userId[i]);
            posts = postRepository.findByUserId(Long.parseLong(userId[i]));
```

```
            posts.forEach(post -> postsForUsers.add(post));
        }
        return postsForUsers;

    }
}
```

Аннотация `@HystrixCommand()` указывает на то, что этот метод должен находиться впереди, и Spring Framework вставит эту реализацию. Он делает это с аспектом, который перехватывает все входящие запросы и реализует описанный ранее протокол.

Итак, давайте посмотрим, как это работает, в частности, через призму сценария с использованием шквала повторных отправок запросов, о котором вы узнали в предыдущей главе. Я хочу воспользоваться простой реализацией повторных отправок для службы Connections' Posts, которая привела к тому, что система оставалась в неработоспособном состоянии в течение продолжительного периода времени после восстановления работы сети, и связать ее с предохранителем, который защищает службу Posts. Посмотрите на рис. 10.4. Мы будем выполнять те же нагрузочные тесты, что и прежде.

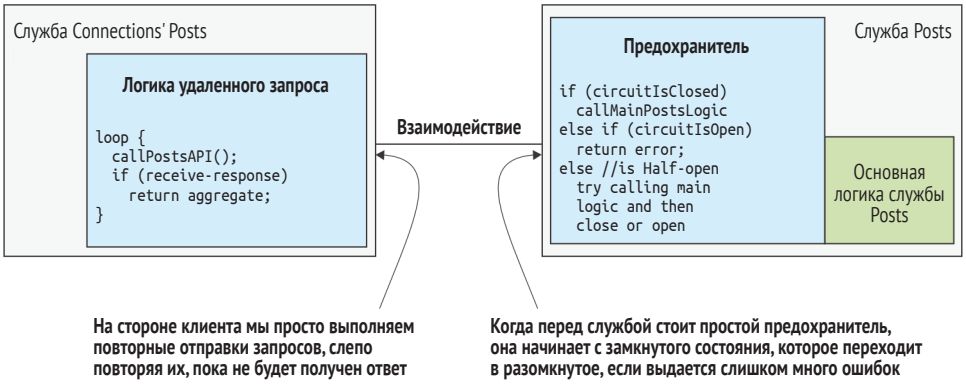


Рис. 10.4 ❖ При первом тестовом запуске вы будете работать с простыми повторными отправками запросов на клиентской стороне взаимодействия и простым предохранителем, размещенным в передней части службы

Настройка

И снова я отсылаю вас к инструкциям по настройке для запуска примеров, которые вы найдете в предыдущих главах. В этой главе нет новых требований для запуска образца.

Вы будете получать доступ к файлам в каталоге cloudnative-circuitbreaker, поэтому перейдите в этот каталог в окне терминала.

И, как я писала в предыдущих главах, я уже выполнила предварительную сборку образов Docker и сделала их доступными в Docker Hub. Если вы хотите собрать исходный код Java и образы Docker и перенести их в собственный репозиторий образов, я отсылаю вас к предыдущим главам (самые подробные инструкции даны в главе 5).

Запуск приложений

В ходе чтения этой главы вы будете использовать разные версии предохранителя, поэтому для начала вам нужно скачать правильный тег в репозитории GitHub:

```
git checkout circuitbreaker/0.0.1
```

Вам понадобится кластер Kubernetes с достаточной вместимостью, как я описывала в главе 9. Если у вас все еще есть примеры из предыдущей главы, нет необходимости выполнять очистку и запускать все по новой; команды, которые вы будете выполнять здесь, обновят версии всех микросервисов соответствующим образом. Если вы хотите начать с нуля, можете использовать скрипт `deleteDeploymentComplete.sh`, как и прежде. Он позволяет поддерживать службы MySQL, Redis и SCCS в работающем состоянии. При вызове скрипта без параметров удаляются только три развертывания микросервисов; при вызове скрипта с аргументом `all` службы MySQL, Redis и SCSS также удаляются.

Учитывая, что вы скачали тег Git, как описано ранее, вы можете развернуть или обновить работающие службы, запустив этот скрипт:

```
./deployApps.sh
```

Если вы сделаете это во время выполнения команды `watch kubectl get all` в другом окне, вы либо увидите обновленную службу Posts – что касается первого примера, изменилась только эта служба, – либо увидите, что развернуты все три микросервиса. Топология приложения показана на рис. 10.5, где развернуты следующие версии приложений:

- *Connections' Posts* – это версия из проекта обеспечения устойчивости запросов (из главы 9), в котором реализована простая реализация повторной

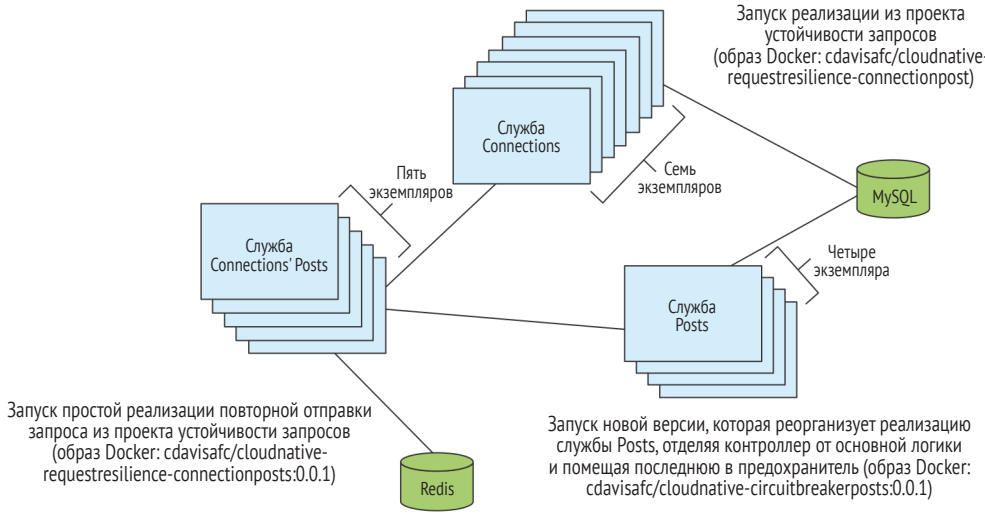


Рис. 10.5 ❖ Топология развертывания содержит версии служб Connections 'Posts и Connections из предыдущей главы и предоставляет новую версию службы Posts. Эта реализация оборачивает основную логику Posts в предохранитель

отправки запроса, та, что слепо выполняет повторную отправку запросов с тайм-аутом;

- *Connections* – это версия из проекта обеспечения устойчивости запросов и стандартная реализация;
- *Posts* – это новая версия приложения, которая подверглась рефакторингу, чтобы отделить контроллер от основной логики службы. Основной метод в службе теперь обернут в предохранитель Hystrix.

Давайте теперь отправим в эту реализацию какую-нибудь нагрузку. Это делается с помощью приведенных далее двух команд:

```
kubectl create configmap jmeter-config \
 --from-file=jmeter_run.jmx=loadTesting/ConnectionsPostsLoad.jmx
kubectl create -f loadTesting/jmeter-deployment.yaml
```

Если вы выполнили первую команду во время экспериментов из главы 9, повторный запуск здесь не требуется, поскольку словарь конфигурации для развертывания Apache JMeter уже существует. Теперь давайте посмотрим на результаты нагрузочного теста:

```
$ kubectl logs -f <name of your jmeter pod>
START Running Jmeter on Sun Feb 24 05:21:46 UTC 2019
JVM_ARGS=-Xmn442m -Xms1768m -Xmx1768m
jmeter args=-n -t /etc/jmeter/jmeter_run.jmx -l resultsconnectionsposts
Feb 24, 2019 5:21:48 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
Creating summariser <summary>
Created the tree successfully using /etc/jmeter/jmeter_run.jmx
Starting the test @ Sun Feb 24 05:21:48 UTC 2019 (1550985708891)
Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445
summary +     85 in 00:00:10 =    8.1/s Err:     0 (0.00%) Active: 85
summary +    538 in 00:00:30 =   18.0/s Err:     0 (0.00%) Active: 332
summary =    623 in 00:00:40 =   15.4/s Err:     0 (0.00%)
summary +   1033 in 00:00:30 =   34.5/s Err:     0 (0.00%) Active: 579
summary =   1656 in 00:01:10 =   23.5/s Err:     0 (0.00%)
summary +   1529 in 00:00:30 =   51.0/s Err:     0 (0.00%) Active: 829
summary =   3185 in 00:01:40 =   31.7/s Err:     0 (0.00%)
summary +   2029 in 00:00:30 =   67.6/s Err:     0 (0.00%) Active: 1077
summary =   5214 in 00:02:10 =   40.0/s Err:     0 (0.00%)
summary +   2520 in 00:00:30 =   84.1/s Err:     0 (0.00%) Active: 1325
summary =   7734 in 00:02:40 =   48.2/s Err:     0 (0.00%)
summary +   2893 in 00:00:30 =   96.4/s Err:     0 (0.00%) Active: 1500
summary =  10627 in 00:03:10 =   55.8/s Err:     0 (0.00%)
summary +   3055 in 00:00:30 =  101.8/s Err:     0 (0.00%) Active: 1500
summary =  13682 in 00:03:40 =   62.1/s Err:     0 (0.00%)
summary +   3007 in 00:00:30 =  100.2/s Err:     0 (0.00%) Active: 1500
summary =  16689 in 00:04:10 =   66.7/s Err:     0 (0.00%)

<time marker 1 - I have broken the network between Posts and MySQL>

summary +   2510 in 00:00:30 =   83.6/s Err:  2084 (83.03%) Active: 1500
summary =  19199 in 00:04:40 =   68.5/s Err:  2084 (10.85%)
summary +   3000 in 00:00:30 =  100.0/s Err:  3000 (100.00%) Active: 1500
```

```
summary =  22199 in 00:05:10 =   71.5/s Err:  5084 (22.90%)
summary +   3000 in 00:00:30 =  100.0/s Err:  3000 (100.00%) Active: 1500
summary =  25199 in 00:05:40 =   74.0/s Err:  8084 (32.08%)
summary +   2953 in 00:00:30 =   98.4/s Err:  2953 (100.00%) Active: 1500
summary =  28152 in 00:06:10 =   76.0/s Err: 11037 (39.21%)
summary +   2916 in 00:00:30 =   96.9/s Err:  2916 (100.00%) Active: 1500
summary =  31068 in 00:06:40 =   77.6/s Err: 13953 (44.91%)
summary +   3046 in 00:00:30 =  101.7/s Err:  3046 (100.00%) Active: 1500
summary =  34114 in 00:07:10 =   79.3/s Err: 16999 (49.83%)
summary +   3019 in 00:00:30 =  100.7/s Err:  3019 (100.00%) Active: 1500
summary =  37133 in 00:07:40 =   80.7/s Err: 20018 (53.91%)

<time marker 2 - I have repaired the network between Posts and MySQL>

summary +   2980 in 00:00:30 =   99.3/s Err:  2980 (100.00%) Active: 1500
summary =  40113 in 00:08:10 =   81.8/s Err: 22998 (57.33%)
summary +   3015 in 00:00:30 =  100.5/s Err:  3015 (100.00%) Active: 1500
summary =  43128 in 00:08:40 =   82.9/s Err: 26013 (60.32%)
summary +   3020 in 00:00:30 =  100.7/s Err:  3020 (100.00%) Active: 1500
summary =  46148 in 00:09:10 =   83.8/s Err: 29033 (62.91%)
summary +   3075 in 00:00:30 =  102.5/s Err:  3072 (99.90%) Active: 1500
summary =  49223 in 00:09:40 =   84.8/s Err: 32105 (65.22%)
summary +   3049 in 00:00:30 =  101.6/s Err:  2395 (78.55%) Active: 1500
summary =  52272 in 00:10:10 =   85.6/s Err: 34500 (66.00%)
summary +   3191 in 00:00:30 =  106.4/s Err:  2263 (70.92%) Active: 1500
summary =  55463 in 00:10:40 =   86.6/s Err: 36763 (66.28%)
summary +   2995 in 00:00:30 =   99.7/s Err:  1203 (40.17%) Active: 1500
summary =  58458 in 00:11:10 =   87.2/s Err: 37966 (64.95%)
summary +   3031 in 00:00:30 =  101.1/s Err:  1193 (39.36%) Active: 1500
summary =  61489 in 00:11:40 =   87.8/s Err: 39159 (63.68%)
summary +   3009 in 00:00:30 =  100.3/s Err:  1182 (39.28%) Active: 1500
summary =  64498 in 00:12:10 =   88.3/s Err: 40341 (62.55%)
summary +   3083 in 00:00:30 =  102.8/s Err:   859 (27.86%) Active: 1500
summary =  67581 in 00:12:40 =   88.9/s Err: 41200 (60.96%)
summary +   3110 in 00:00:30 =  103.7/s Err:   597 (19.20%) Active: 1500
summary =  70691 in 00:13:10 =   89.4/s Err: 41797 (59.13%)
summary +   2999 in 00:00:30 =   99.9/s Err:     0 (0.00%) Active: 1500
summary =  73690 in 00:13:40 =   89.8/s Err: 41797 (56.72%)
summary +   3001 in 00:00:30 =  100.1/s Err:     0 (0.00%) Active: 1500
summary =  76691 in 00:14:10 =   90.2/s Err: 41797 (54.50%)
```

Как и в случае с тестами в предыдущей главе, после того как вся нагрузка была установлена (на метке времени 1 в предыдущем журнале), вы нарушаете работу сети между всеми экземплярами службы Posts и базой данных MySQL. Как видите, это приводит к неудачным попыткам для всех запросов к службе Connections' Posts (то, что вы вызываете из тестов с JMeter). После примерно трехминутного простоя вы восстанавливаете работу сети на метке времени 2. Изучив вывод журнала, видно, что для появления первых признаков восстановления потребовалось всего около одной минуты, а для полного восстановления – еще 3,5–4 минуты. Эти результаты приведены в табл. 10.1 бок о бок с результатами простой реализации повторной отправки запроса (в самом деле недоброжелательный клиент) без защиты службы.

Таблица 10.1. Предохранитель обеспечивает существенную защиту от шквалов повторных отправок запросов

Версия службы Connections' Posts	Версия службы Posts	Сколько понадобилось времени до появления начальных признаков восстановления	Дополнительное время для полного восстановления
Простая реализация повторной отправки запроса	Предохранитель не используется	9 минут	12–13 минут
Простая реализация повторной отправки запроса	Предохранитель защищает службу	1–2 минуты	4–5 минут

Это уже серьезная разница! Отмечу, что предохранитель обеспечивает защиту не только от шквалов повторных отправок запросов, но и от чрезмерной нагрузки или других сбоев, независимо от причины. Но в этом случае каким образом предохранитель изменил взаимодействие между Connections' Posts и службой Posts, что позволило системе восстановиться намного быстрее? Возвращаемся к рис. 10.3; вы реализовали третий сценарий, поэтому вместо тайм-аута Connections' Posts для каждой повторной отправки запроса, после того как предохранитель разомкнут, он быстро получает ответ от службы Posts, ответ, который четко указывает на проблему (с кодом состояния 500), поэтому оставшихся повторных отправок запросов, если хотите, будет намного меньше.

Давайте рассмотрим первую реализацию. Как вы представляете, аннотация `@HistrixCommand()` позволяет многочисленным опциям конфигурации управлять своим поведением. В первом примере вы просто приняли значения по умолчанию. Глядя на простую диаграмму состояний, на рис. 10.6 я аннотировала ее, используя эти значения. Предохранитель отключается, когда 50 % запросов к службе завершаются неудачно, и остается разомкнутым в течение 5 секунд перед переходом в полуоткрытое состояние.

Для предохранителя Hystrix можно использовать множество других вариантов конфигурации[1]. Например, можно установить минимальное количество отказов, прежде чем предохранитель отключится. Но я хочу сосредоточиться на том, чтобы установить альтернативный метод. Вы помните, что когда вы реализовали более мягкий вариант избыточности запросов с помощью Spring Retry в предыдущей главе, вы добавили альтернативный метод, при котором кешировали предыдущие результаты и использовали их, когда служба Posts не отвечала? Здесь то же самое: когда предохранитель разомкнут, вместо того чтобы возвращать ошибку, как это делает ваша реализация в настоящее время, вы вернете что-то, чтобы заменить реальный результат.

> **ПРИМЕЧАНИЕ** Подзаголовок этой книги – «Проектирование программного обеспечения, устойчивого к изменениям». Одна из самых важных вещей, которые вы должны делать при проектировании программного обеспечения, – всегда думать: «Что должно делать программное обеспечение в случае, если операция, которую я вызываю, не увенчалась успехом?»

[1] Эти параметры конфигурации доступны на GitHub по адресу **http://mng.bz/O2rK**.

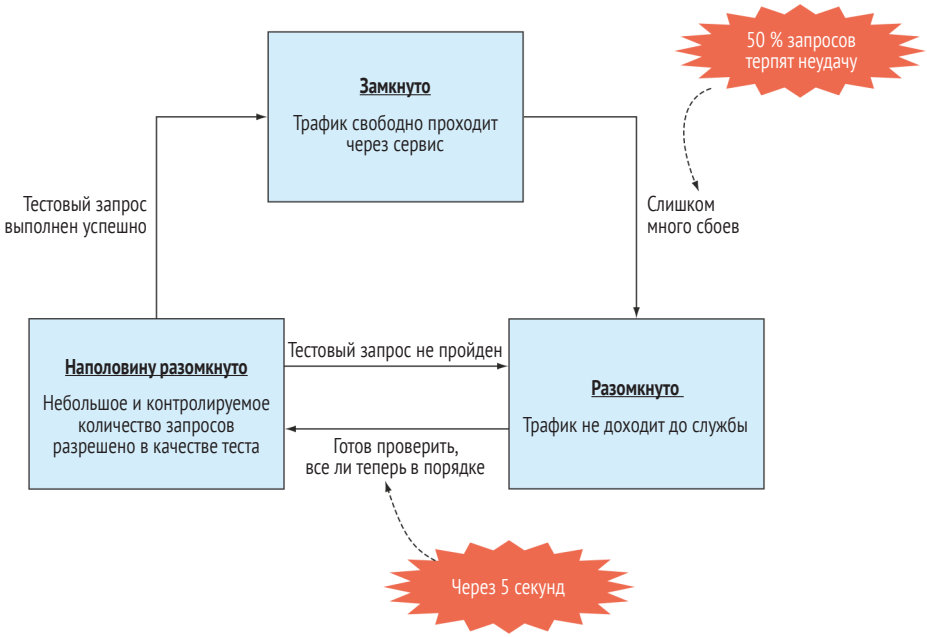


Рис. 10.6 ❖ Реализация Hystrix по умолчанию отключает предохранитель при сбое 50 % запросов и переходит из состояния «разомкнуто» в состояние «наполовину разомкнуто» через несколько секунд после входа в состояние «разомкнуто». Успешные или неудачные запросы, сделанные в состоянии «наполовину разомкнуто», переведут предохранитель в состояния «замкнуто» и «разомкнуто» соответственно

Весьма показательно, что во фреймворках, которые помогают реализовать шаблоны устойчивости, есть встроенные примитивы для альтернативы, по обе стороны взаимодействия; на рис. 10.7 это наглядно показано.

Контекст для каждого из этих альтернативных методов отличается. В левой части рис. 10.7 служба Connections' Posts является *потребителем* информации, запрашиваемой в ходе взаимодействия, и она может решать, что делать, если актуальная информация недоступна. Устаревший контент или его отсутствие – что лучше? В нашей окончательной реализации в главе 9 вы сделали вызов и вернули кешированный контент. В правой части рис. 10.7 служба Posts – это *поставщик* информации, запрашиваемой посредством взаимодействия, и она должна решить, что лучше: ответ типа «успешно» или полный сбой. Каким бы ни был ответ со статусом «успешно», это поведение должно быть хорошо задокументировано, чтобы клиенты не были склонны полагать, что у них есть один набор данных, когда на самом деле они получают альтернативный набор.

Давайте проверим это на работающей реализации. Мы внесем лишь незначительные изменения в наш предыдущий пример. Чтобы увидеть их, пожалуйста, скачайте приведенный ниже тег Git с помощью этой команды:

```
git checkout circuitbreaker/0.0.2
```

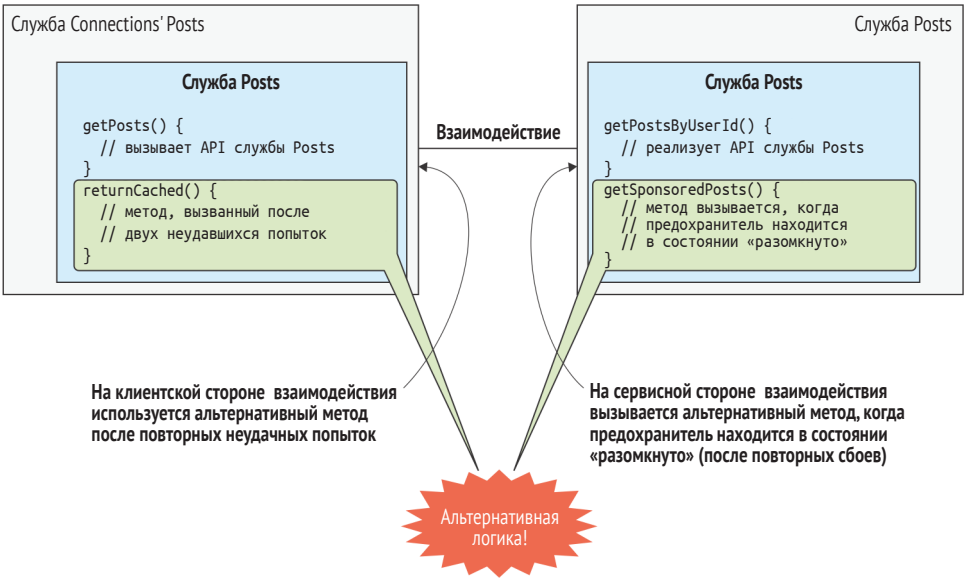


Рис. 10.7 ❖ Сбои могут возникать на любой стороне взаимодействия, а альтернативы обеспечивают меры защиту

Глядя на код API Posts и, в частности, на класс `PostsService`, видно, что теперь тут есть альтернативный метод, и указывает на него аннотация `@HystrixCommand`. В листинге 10.2 вы видите, что реализация альтернативной логики возвращает предопределенные результаты, спонсорский контент, если живые данные недоступны.

Листинг 10.2 ❖ Методы из файла PostsService.java

```
@HystrixCommand(fallbackMethod = “getSponsoredPosts”)
public Iterable<Post> getPostsByUserId(String userIds,
                                       String secret) throws Exception {

    logger.info(utils.ipTag() + “Attempting getPostsByUserId”);

    Iterable<Post> posts;

    if (userIds == null) {
        logger.info(utils.ipTag() + “getting all posts”);
        posts = postRepository.findAll();
        return posts;
    } else {
        ArrayList<Post> postsForUsers = new ArrayList<Post>();
        String userId[] = userIds.split(“,”);
        for (int i = 0; i < userId.length; i++) {
            logger.info(utils.ipTag() +
                        “getting posts for userId “ + userId[i]);
        posts = postRepository.findByUserId(Long.parseLong(userId[i]));
        posts.forEach(post -> postsForUsers.add(post));
        }
    return postsForUsers;
    }
```

```
}

public Iterable<Post> getSponsoredPosts(String userIds,
                                        String secret) {
    logger.info(utils.ipTag() +
                "Accessing Hystrix fallback getSponsoredPosts");
    ArrayList<Post> posts = new ArrayList<Post>();
    posts.add(new Post(999L, "Some catchy title",
                       "Some great sponsored content"));
    posts.add(new Post(999L, "Another catchy title",
                       "Some more great sponsored content"));
    return posts;
}
```

Здесь я хочу обратить ваше внимание на два момента:

- альтернативный метод вызывается каждый раз, когда из команды, защищенной Hystrix, возвращается ошибка (в данном случае это метод `getPostsByUserId`), даже когда предохранитель замкнут. Библиотека Hystrix поддерживает попытки использования альтернатив во всех случаях сбоя, даже если они никогда не являются частью серьезного сбоя;
- альтернативные методы Hystrix могут быть связаны; если основной метод завершается неудачно, можно вызвать метод `fallbackMethod1`. Он может, например, попытаться вычислить результаты, используя кешированные данные, или загрузить данные через альтернативный канал. Если метод `fallback-Method1` потерпит неудачу, управление может быть передано методу `fallbackMethod2` и т. д. Это мощная абстракция, которая находится в вашем распоряжении.

Вы заметите, что наша реализация альтернативной логики очень и очень проста. Она даже жестко кодирует контент (в коде!), вместо того чтобы извлекать его из хранилища данных; это делается только для простоты реализации. Пожалуйста, не нужно жестко кодировать контент в свой исходный код!

Запуск приложений

Я предполагаю, что у вас уже есть пример, приведенный ранее в этом разделе, и что вы скачали ветку Git, как я описывала ранее. Если ваш предыдущий нагрузочный тест все еще выполняется, остановите его с помощью этой команды:

```
kubectl delete deploy jmeter-deployment
```

Вы можете обновить свое развертывание до версии, которая реализует альтернативное поведение, запустив приведенный ниже скрипт или выполнив содержащиеся в нем команды:

```
./deployApps.sh
```

Опять же, если вы понаблюдаете за командой `kubectl get all`, то увидите, что службы Connections и Posts обновляются. Обновление было необходимо службе Connections только для предварительной загрузки идентификатора пользователя спонсора. Служба Connections' Posts не была обновлена. Вы по-прежнему будете запускать простую реализацию повторных отправок запросов из главы 9. И наконец, давайте поместим нагрузку на это развертывание:

```
kubectl create -f loadTesting/jmeter-deployment.yaml
```

А теперь посмотрите журналы этого развертывания:

```
START Running Jmeter on Sun Feb 24 04:39:23 UTC 2019
JVM_ARGS=-Xmn542m -Xms2168m -Xmx2168m
jmeter args=-n -t /etc/jmeter/jmeter_run.jmx -l resultsconnectionsposts
Feb 24, 2019 4:39:25 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
Creating summariser <summary>
Created the tree successfully using /etc/jmeter/jmeter_run.jmx
Starting the test @ Sun Feb 24 04:39:25 UTC 2019 (1550983165958)
Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445
summary +    217 in 00:00:21 =   10.4/s Err:     0 (0.00%) Active: 171
summary +    712 in 00:00:30 =   23.7/s Err:     0 (0.00%) Active: 419
summary =    929 in 00:00:51 =   18.3/s Err:     0 (0.00%)
summary +   1209 in 00:00:30 =   40.3/s Err:     0 (0.00%) Active: 667
summary =   2138 in 00:01:21 =   26.4/s Err:     0 (0.00%)
summary +   1706 in 00:00:30 =   57.0/s Err:     0 (0.00%) Active: 916
summary =   3844 in 00:01:51 =   34.7/s Err:     0 (0.00%)
summary +   2205 in 00:00:30 =   73.5/s Err:     0 (0.00%) Active: 1166
summary =   6049 in 00:02:21 =   43.0/s Err:     0 (0.00%)
summary +   2705 in 00:00:30 =   90.2/s Err:     0 (0.00%) Active: 1415
summary =   8754 in 00:02:51 =   51.2/s Err:     0 (0.00%)
summary +   2998 in 00:00:30 =   99.9/s Err:     0 (0.00%) Active: 1500
summary =  11752 in 00:03:21 =   58.5/s Err:     0 (0.00%)

<time marker 1 - I have broken the network between Posts and MySQL>

summary +   3004 in 00:00:30 =  100.0/s Err:     0 (0.00%) Active: 1500
summary =  14756 in 00:03:51 =   63.9/s Err:     0 (0.00%)
summary +   2997 in 00:00:30 =   99.9/s Err:     0 (0.00%) Active: 1500
summary =  17753 in 00:04:21 =   68.1/s Err:     0 (0.00%)
summary +   3001 in 00:00:30 =  100.1/s Err:     0 (0.00%) Active: 1500
summary =  20754 in 00:04:51 =   71.4/s Err:     0 (0.00%)
summary +   3000 in 00:00:30 =  100.0/s Err:     0 (0.00%) Active: 1500
summary =  23754 in 00:05:21 =   74.0/s Err:     0 (0.00%)
summary +   3000 in 00:00:30 =  100.0/s Err:     0 (0.00%) Active: 1500
summary =  26754 in 00:05:51 =   76.3/s Err:     0 (0.00%)
summary +   3000 in 00:00:30 =  100.0/s Err:     0 (0.00%) Active: 1500
summary =  29754 in 00:06:21 =   78.1/s Err:     0 (0.00%)
summary +   2995 in 00:00:30 =   99.9/s Err:     0 (0.00%) Active: 1500
summary =  32749 in 00:06:51 =   79.7/s Err:     0 (0.00%)

<time marker 2 - I have repaired the network between Posts and MySQL>

summary +   3005 in 00:00:30 =  100.2/s Err:     0 (0.00%) Active: 1500
summary =  35754 in 00:07:21 =   81.1/s Err:     0 (0.00%)
summary +   2997 in 00:00:30 =   99.9/s Err:     0 (0.00%) Active: 1500
summary =  38751 in 00:07:51 =   82.3/s Err:     0 (0.00%)
```

Как обычно, временная метка 1 указывает время, когда вы разорвали сетевое соединение между службой Posts и базой данных MySQL, а метка 2 показывает, когда вы восстановили соединения. И, как вы видите, вызовы к службе Connections' Posts ни разу не окончились неудачей, даже во время отключения сети. Это именно то, чего вы и ожидали, учитывая, что альтернативный метод предохранителя возвращает спонсорский контент при любых сбоях службы Posts.

Возможно, более интересным показателем является то, насколько быстро оперативный контент возвращается после восстановления работы сети. Есть догадки? Да, вы правы: менее чем за 5 секунд. Напомню, что значение по умолчанию для `sleepWindowMilliseconds` равно 5000, а это означает, что состояние предохранителя будет установлено на «наполовину разомкнуто» через 5 секунд после установки в положение «разомкнуто». Как только это произойдет, пробный запрос, который вы пропускаете через логику службы Posts, будет успешным, и предохранитель закроется, вернув приложение в стабильное состояние. Этот переход можно увидеть в выводе журнала одного из экземпляров службы Posts:

```
2019-02-23 02:59:03.084 getting posts for userId 2
2019-02-23 02:59:03.148 Attempting getPostsByUserId
2019-02-23 02:59:03.148 getting posts for userId 2
2019-02-23 02:59:03.167 Attempting getPostsByUserId
2019-02-23 02:59:03.167 getting posts for userId 2

<time marker 1 - I have broken the network between Posts and MySQL>

2019-02-23 02:59:03.213 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 02:59:03.237 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 02:59:03.243 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 02:59:03.313 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 02:59:03.351 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 02:59:03.357 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 02:59:03.394 Accessing Hystrix fallback getSponsoredPosts
... (there are many more of these log lines)

<time marker 2 - I have repaired the network between Posts and MySQL>

... (another 5 seconds or so of Hystrix mentioning messages)
(then, ...)

2019-02-23 03:02:33.705 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:33.717 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:33.717 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:33.898 getting posts for userId 3
2019-02-23 03:02:33.898 getting posts for userId 3
2019-02-23 03:02:33.899 getting posts for userId 3
2019-02-23 03:02:33.899 getting posts for userId 3
2019-02-23 03:02:33.900 getting posts for userId 3
2019-02-23 03:02:33.905 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:33.911 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:33.943 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:34.080 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:34.100 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:34.113 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:34.216 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:34.225 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:34.300 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:34.368 Accessing Hystrix fallback getSponsoredPosts
2019-02-23 03:02:34.398 Attempting getPostsByUserId
2019-02-23 03:02:34.398 getting posts for userId 2
2019-02-23 03:02:34.400 getting posts for userId 3
2019-02-23 03:02:34.433 Attempting getPostsByUserId
```

```
2019-02-23 03:02:34.433 getting posts for userId 2
2019-02-23 03:02:34.434 Attempting getPostsByUserId
2019-02-23 03:02:34.434 getting posts for userId 2
2019-02-23 03:02:34.435 getting posts for userId 3
2019-02-23 03:02:34.437 getting posts for userId 3
2019-02-23 03:02:34.472 Attempting getPostsByUserId
2019-02-23 03:02:34.472 getting posts for userId 2
2019-02-23 03:02:34.475 getting posts for userId 3
2019-02-23 03:02:34.556 Attempting getPostsByUserId
2019-02-23 03:02:34.556 getting posts for userId 2
2019-02-23 03:02:34.559 getting posts for userId 3
2019-02-23 03:02:34.622 Attempting getPostsByUserId
(and operation has returned to normal)
```

В табл. 10.2 показаны результаты каждого из тестов, которые вы выполняли в этой и предыдущей главах. В каждом случае имитировалось одно и то же трехминутное отключение сети между службами Connections' Posts и Posts, но с различными шаблонами, применяемыми на клиентской (в глава 9) или серверной стороне (в главе 10) взаимодействия.

Эти результаты действительно интересны. Реализация шаблонов, ориентированных на устойчивость взаимодействий, имеет большое значение для общего состояния вашего программного обеспечения. Хотя и очевидно, что шаблоны могут применяться как на клиентской, так и на серверной стороне взаимодействия, зачастую вы не будете нести ответственность за реализации на обеих сторонах. Следовательно, если вы реализуете потребителя, очень важно, чтобы вы полностью представляли себе, что такое контракт для API – будет ли сервис, который вы используете, менять результаты при отклонении выполнения от «счастливого пути». И когда вы предоставляете сервис, убедитесь, что полностью указали этот контракт.

Размышляя о том, что делает предохранитель, особенно в свете использования аспектов в реализации Hystrix, видно, что, по сути, он действует как шлюз для службы Posts. Но предохранитель – это всего лишь один из примеров функциональности, который вы, возможно, захотите разместить перед службой. Давайте теперь рассмотрим шлюз API как более общий шаблон.

10.2. API-шлюзы

Доступность шлюзов API с открытым исходным кодом и коммерческих вариантов предшествует появлению микросервисных и облачных архитектур. Например, компании Apigee (с момента приобретения ее Google) и Mashery (с того момента, как ее приобрела Intel, а затем продала ее TIBCO) были основаны в начале 2000-х годов, и обе эти компании занимались шлюзами API.

Роль API-шлюза в архитектуре программного обеспечения всегда была именно такой, как явствует из заголовка этой главы, – находиться перед фрагментами реализации и предоставлять целый ряд сервисов. Вот что могут включать в себя эти сервисы:

- *аутентификация и авторизация* – управление доступом к службе, перед которой стоит шлюз. Механизмы для этого контроля доступа варьируют и могут включать в себя подходы на базе секретов, такие как использование

Таблица 10.2. Результаты моделирования отключения сети показывают преимущества использования определенных шаблонов для облачной среды со взаимодействиями сервисов

Версия службы Connections' Posts	Версия службы Posts	Во время отключения сети	Сколько понадобилось времени до появления начальных признаков восстановления	Количество времени для полного восстановления	Прогон теста в главе
Простая реализация повторной отправки запроса	Предохранитель не используется	100 % ошибок	9 минут	12–13 минут	9
Мягкая повторная отправка с использованием Spring Retry при отсутствии альтернативной логики	Предохранитель не используется	100 % ошибок	1 минута	3 минуты	9
Мягкая повторная отправка с использованием Spring Retry и альтернативного метода	Предохранитель не используется	0 % ошибок	Н/П – нет сбоя во время отключения сети	Н/П	9
Простая реализация повторной отправки запроса	Предохранитель, защищающий службу, – альтернативный метод не используется	100 % ошибок	1–2 минуты	4–5 минут	10
Простая реализация повторной отправки запроса	Предохранитель, защищающий службу, с использованием альтернативного метода	0 % ошибок	Н/П – нет сбоя во время отключения сети	< 5 секунд Учитывая «полное восстановление», время, когда фактические результаты, а не результаты спонсора, снова возвращаются	10

паролей или маркеров, или это могут быть подходы на базе сети, либо интеграция с сервисами типа межсетевого экрана, либо их реализация;
- *шифрование данных на лету* – шлюз API может обрабатывать дешифрование, и поэтому это то место, где необходимо управлять сертификатами;
- *защита службы от скачков нагрузки* – при правильной настройке шлюз API становится единственным способом, с помощью которого клиенты могут получить доступ к службе. Следовательно, механизмы регулирования нагрузки, реализованные здесь, могут обеспечить существенную защиту. Вы можете подумать, что это несколько похоже на то, о чем мы только что говорили в предыдущих разделах, посвященных предохранителям, и будете правы;
- *ведение журналов доступа* – поскольку весь трафик в службу проходит через шлюз API, у вас есть возможность регистрировать весь доступ. Эти журналы могут поддерживать множество вариантов использования, включая аудит и наблюдаемость за операциями.

Любая из этих проблем может быть решена внутри самой службы, но очевидно, что это общие проблемы, которые не нужно реализовывать снова и снова. Использование API-шлюзов освобождает разработчика от действий, которые можно рассматривать как слесарные работы, позволяя сосредоточиться на потребностях бизнеса. Но, возможно, что еще более важно, оно обеспечивает точку, в которой средства управления предприятием могут применяться единообразно. Конечно, одна из самых сложных задач для ИТ-процессов – продемонстрировать, что требования к обеспечению информационной безопасности и соответствия выполняются во всем – централизованное управление является ключевым.

Шлюз API может выполнять свои функции, взаимодействуя со множеством других служб. Например, шлюз сам по себе не хранит пользователей, которые должны быть аутентифицированы и авторизованы. Он зависит от решения, реализующего управление идентификацией и доступом в хранилища удостоверений (например, LDAP) для этих служб; он просто обеспечивает соблюдение изложенной там политики.

На рис. 10.8 изображен простой сценарий: шлюз API находится перед парой сервисов, весь доступ к этим сервисам осуществляется через шлюз, и он взаимодействует с другими компонентами для поддержки функциональности, которую он предлагает. Администратор ИТ-системы, используя интерфейс для шлюза API, настроит необходимые политики. На рисунке также изображен аудитор, просматривающий журналы доступа каждой из служб.

10.2.1. API-шлюзы в программном обеспечении для облачной среды

Если шлюзы API существуют и используются на протяжении более 15 лет, почему я говорю о них в этой книге? Ну, как вы можете себе представить, как и во многих других темах, которые мы обсуждали до сих пор, переход к архитектурам программного обеспечения для облачной среды вводит новые требования к шлюзу API:

- совершенно очевидно, что компонентное представление программного обеспечения, которое дает гораздо больше независимых (микро)сервисов,

увеличивает количество сервисов, которыми нужно управлять, на несколько порядков. Хот, конечно, даже тогда это не было идеальным, по крайней мере для ИТ-персонала теоретически было возможно управлять доступом к сервисам без централизованной плоскости управления. Когда у вас есть тысячи или даже десятки тысяч экземпляров сервисов, это больше не представляется возможным;

- постоянное изменение, оказываемое на экземпляры сервисов, так как они воссоздаются во время отключений и запланированных обновлений, аналогично означает, что любые ручные конфигурации, которые могли быть выполнены, когда развертывания изменялись только ежегодно или раз в два года (например, правила брандмауэра), теперь совершенно неразрешимы без соответствующего программного обеспечения, которое может помочь;
- сильно распределенные системы привели к внедрению других шаблонов устойчивости, таких как повторная отправка вызова, которые вы только что изучали, которые привносят различные профили нагрузки в службу. Нагрузка, которая будет поступать, менее предсказуема, по сравнению с тем, что было раньше. Вам необходимо защитить сервисы от неожиданных и экстремальных объемов запросов. Предохранитель, о котором шла речь в начале этой главы, – один из видов этой защиты, который проник в шлюз API;

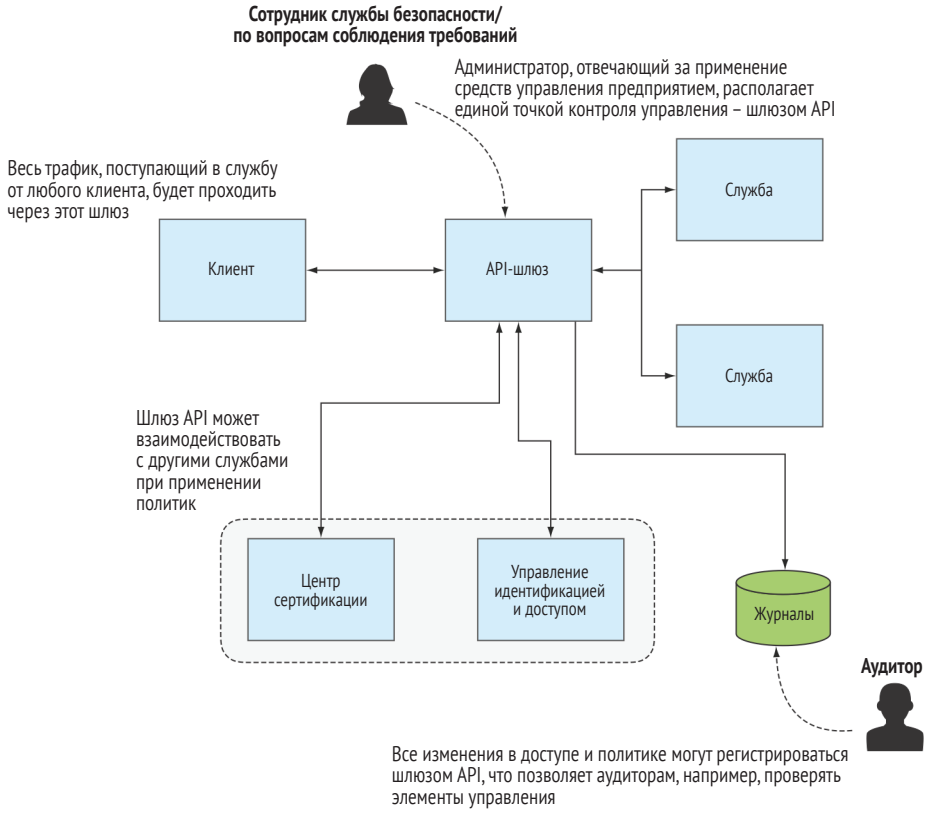


Рис. 10.8 ❖ Шлюз API располагается перед службами и становится для них точкой настройки и применения политик

- архитектуры облачной среды сыграли свою роль в создании новых бизнес-моделей, которые допускают платное потребление сервисов. Шлюз API обеспечивает необходимый учет, возможно, с регулированием нагрузки;
- в предыдущих главах я говорила о параллельных развертываниях. Шлюз API является отличным местом для реализации логики маршрутизации, которая так важна для таких вещей, как безопасные процессы обновления.

В то время как лет десять назад могли настоятельно рекомендовать использовать API-шлюзы, но это не было обязательным, характеристики программного обеспечения для облачной среды делают их крайне важными в настоящее время.

10.2.2. Топология шлюза API

Я надеюсь, что сейчас вы думаете о чем-то вроде: «Хорошо, я понимаю, для чего они нужны, но мне не нравится ваша цифра 10.8. Конечно, централизованный шлюз выглядит как антишаблон для облачной среды». Вы правы! Вам необходимо согласованное применение политик во всех ваших сервисах – это одно из преимуществ моделей API-шлюзов. Но это не значит, что реализация должна быть централизованной. Да, 15 лет назад шлюз API часто развертывался как централизованный, даже кластерный компонент, но в архитектурах для облачной среды теперь все по-другому.

Как я уже неоднократно упоминала, предохранитель, который мы изучали и реализовали в первом разделе этой главы, представляет собой пример шаблона шлюза, и эта реализация, безусловно, была распределенной. Фактически была выполнена компиляция в двоичный файл для самой службы (помните, включение аннотации `@HystrixCommand`?). Чтобы добраться до сути вопроса, необходимы распределенные реализации шаблонов API-шлюза. Чтобы увидеть, как это выглядит, посмотрите на рис. 10.9. На нем видно, что у каждого сервиса был шлюз, прикрепленный к нему спереди, и этот шлюз, как и на предыдущей диаграмме, взаимодействует с набором компонентов, которые необходимы для соответствующих операций или поддерживают их.

Вы также заметите, что я изобразила намного больше экземпляров сервисов, чем на предыдущем рисунке. Можете себе представить, что если бы у вас было так много сервисов с централизованным API-шлюзом и *все* взаимодействия проходили через этот шлюз, вам нужно было бы позаботиться о том, чтобы шлюз был правильно подобран для обработки потока трафика в разнородный и, следовательно, трудно предсказуемый набор экземпляров приложения. Распределяя обработку, каждый экземпляр шлюза обрабатывает нагрузку только для своего сервиса, и гораздо легче правильно определить размеры шлюза.

Я хотела бы потратить немного времени на изучение API-шлюза с открытым исходным кодом, который стал популярным в последние несколько лет, от компании Netflix. Zuul (назван так в честь персонажа привратника из фильма «*Охотники за привидениями*») описывается как «пограничный сервис, который обеспечивает динамическую маршрутизацию, мониторинг, отказоустойчивость, безопасность и т. д.». Это как раз те самые вещи, которые я приписывала шаблону шлюза API. Zuul использует или встраивает несколько других компонентов из инфраструкту-

ры микросервисов Netflix, включая Hystrix (предохранители), Ribbon (распределение нагрузки), Turbine (показатели) и др.

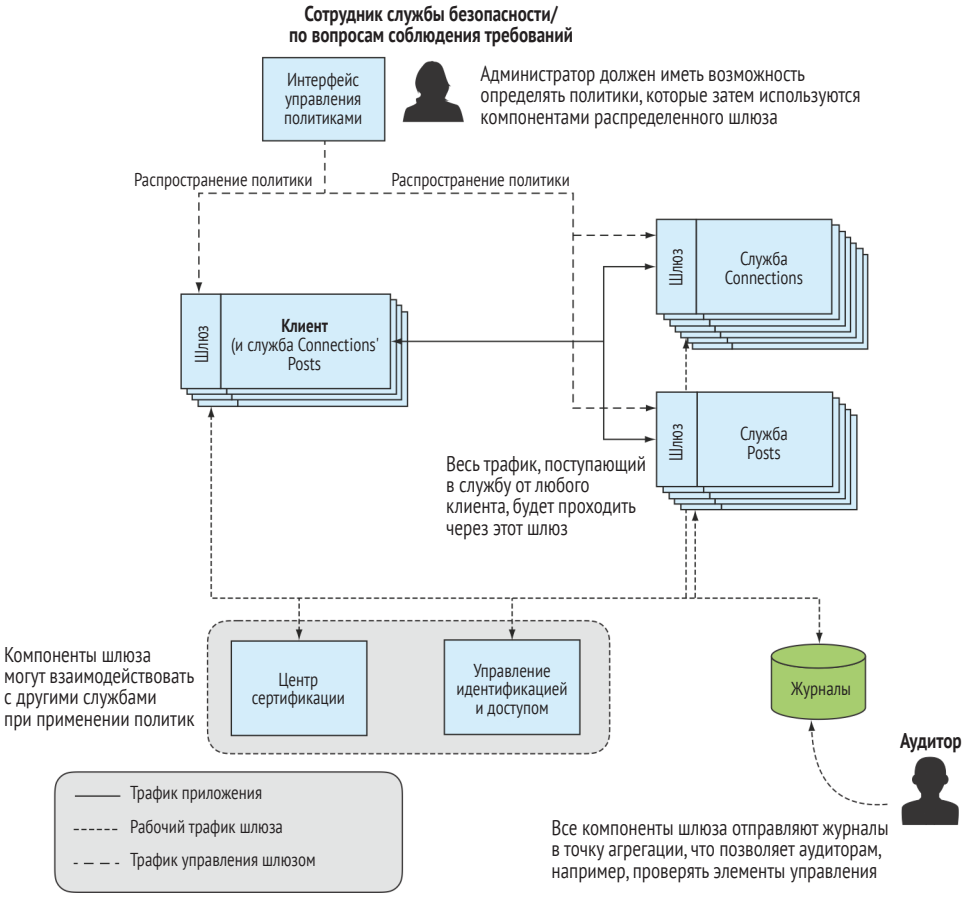


Рис. 10.9 ❖ Можно рассматривать шлюз как единую логическую сущность, которая необходима для администрирования. Но в случае с архитектурами для облачной среды реализация лучше распределена

Zuul, написанный на Java и, следовательно, работающий в виртуальной машине Java, настроен на работу с сервисами через URL-адрес. Например, чтобы настроить его в качестве шлюза для нашей службы Posts, нужно предоставить следующие данные конфигурации:

```
zuul.routes.connectionPosts.url=http://localhost:8090/connectionPosts
server.port=8080
```

Тогда возникает вопрос, как включить Zuul в нашу топологию программного обеспечения. Совершенно возможно создать развертывание, которое выглядит так, как показано на рис. 10.8, но в сильно распределенной архитектуре рекомен-

дуется использовать вариант, близкий к тому, что показан на рис. 10.9. Фактически Spring Cloud предоставляет Zuul возможность встроиться в ваш сервис во многом так же, как и предохранитель в предыдущем примере[1]. Таким образом, вы получаете топологию развертывания, показанную на рис. 10.9.

Встраивание шлюза в сервис дает некоторые очевидные преимущества: между шлюзом и самой службой нет транзитного участка сети, имя хоста для конфигурации больше не требуется, нужен только путь, проблемы совместного использования ресурсов между разными источниками (CORS) исчезают и т. д. Но есть и ряд недостатков.

Во-первых, из предыдущих обсуждений жизненного цикла приложения вы помните, что настройка привязки ближе к концу цикла обеспечивает бóльшую гибкость. Если вы включили предыдущую конфигурацию в файл application.properties, для изменения конфигурации потребуется повторная компиляция. Как мы уже обсуждали, значения свойств могут внедряться позже через переменные среды, но для этого все равно требуется перезапуск виртуальной машины Java (или, по крайней мере, обновление контекста приложения).

Во-вторых, если вы встраиваете компонент Java, это в значительной степени означает, что реализация вашего сервиса также должна быть написана на Java или, по крайней мере, работать на виртуальной машине Java. Хотя все приведенные здесь примеры кода написаны на Java, поддерживаемые мной шаблоны применимы и должны быть реализованы на любом языке, наиболее подходящем для вашего случая. Я не большой поклонник решений, в которых используется только Java.

И наконец, одна из целей шаблона API-шлюза состоит в том, чтобы отделить интересы разработчика сервиса от интересов оператора. Вам нужно предоставить операторам возможность применять согласованные элементы управления ко всем работающим службам и предлагать им плоскость управления, которая делает этот процесс управляемым.

Как же тогда добиться чего-то подобного, как этот шаблон шлюза, с помощью способа, который не зависит от языка программирования, более слабосвязанный и управляемый? Используйте сервисную сеть.

10.3. Сервисная сеть

Нам не нужно проходить весь путь до сервисной сети за один шаг, поэтому позвольте мне вернуться немного назад и начать с примитива, который играет ключевую роль в сервисной сети. Затем я перейду к знакомству с сервисной сетью и ее растущей ролью в архитектуре программного обеспечения для облачной среды.

10.3.1. Сайдкар

Возвращаясь к вопросу, который я только что задала, – как обеспечить функциональность распределенного шлюза API, позволяющего избежать недостатков встраиваемого компонента Java, – ответ на этот вопрос – сайдкар. На самом простом уровне *сайдкар* – это процесс, который работает наряду с вашим основным сервисом. Если вы вернетесь к рис. 10.9, то сможете представить себе, что службы

[1] Как это обычно бывает в Spring Boot, включение в состав Zuul так же просто, как и в случае с `spring-cloud-starternetflix-zuul` в зависимостях Maven или Gradle: **http://mng.bz/YP4o**.

шлюза могут рассматриваться как работающие наряду со службами, которые не обязательно являются встраиваемыми. Чтобы соответствовать требованию касательно того, что не должно быть никакой компиляции в двоичный файл сервиса, это, конечно, означает, что сайдкар шлюза работает как отдельный процесс наряду с основным.

Kubernetes предлагает абстракцию, которая делает эту работу красиво: модуль Kubernetes. *Модуль* – это наименьшая единица развертывания в Kubernetes, которая содержит один или несколько контейнеров. Вы можете разместить свою основную службу в одном контейнере, а службы шлюза – в другом, и оба они будут работать в одном модуле. Теперь мы можем переделать предыдущую диаграмму, чтобы использовать эти конструкции; см. рис. 10.10.

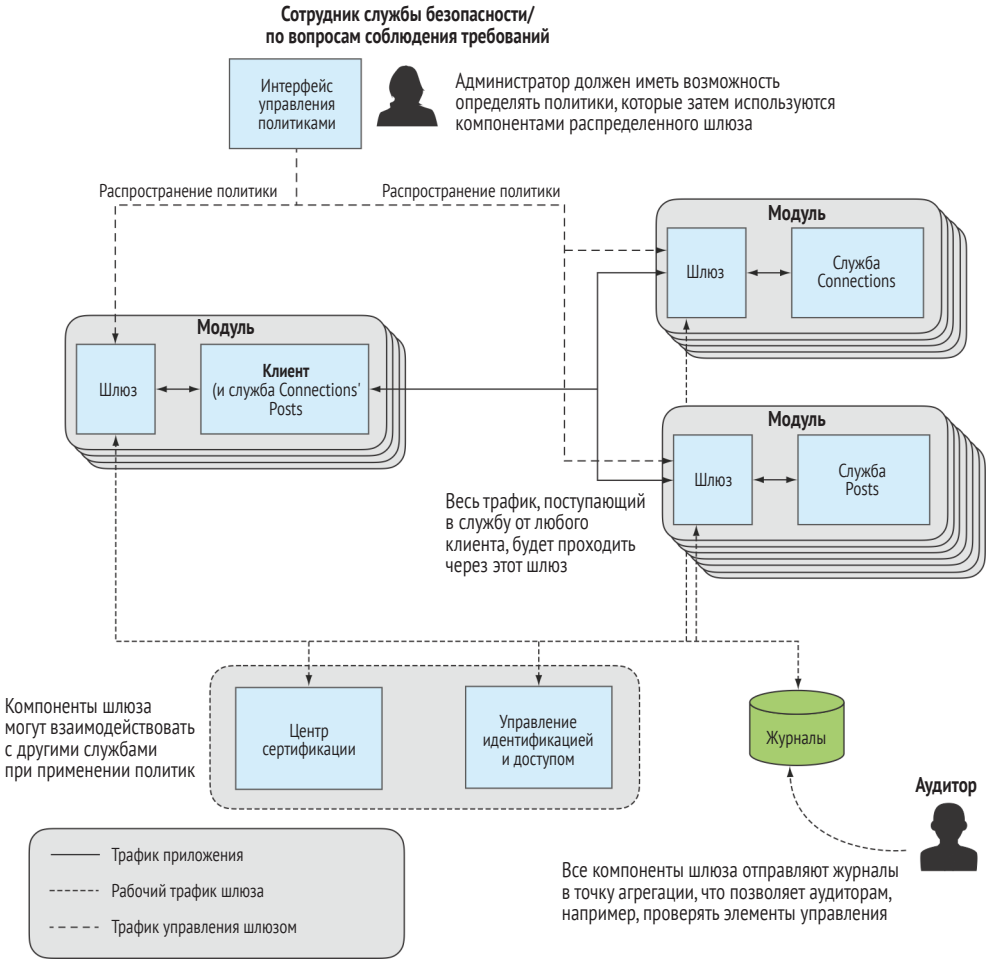


Рис. 10.10 ❖ Распределенный шлюз работает как сайдкар для каждой службы. В Kubernetes это достигается путем запуска двух контейнеров в одном модуле: один из них – это основная служба, а другой – шлюз

У каждого контейнера есть своя среда выполнения, поэтому, скажем, основная служба может работать на виртуальной машине Java, а сайдкар можно реализовать, например, на языке C ++. Посмотрите – один недостаток мы устранили. Но теперь обмен данными между шлюзом и основным сервисом является межпроцессным и даже межконтейнерным, а это означает переход в сети. И снова к нам на помощь приходит архитектура Kubernetes. Все службы, работающие в модуле Kubernetes, размещены на одном IP-адресе, а это означает, что они могут обращаться друг к другу через локальный хост, следовательно, сетевой переход будет минимальным.

Одна из наиболее популярных реализаций сайдкара, используемая сегодня, – это Envoy (**www.envoyproxy.io**). Первоначально созданный компанией Lyft, Envoy – это распределенный прокси-сервер, написанный на языке C ++, что делает его чрезвычайно эффективным. Его можно использовать в различных топологиях развертывания, хотя наиболее распространенным вариантом является наличие каждого экземпляра перед одним экземпляром службы (в топологии, подобной той, что показана на рис. 10.10).

Но это описание не совсем честное. Обратите внимание, что я назвала Envoy прокси-сервером, а не шлюзом. Envoy не просто работает как шлюз; он также выполняет проксирование клиентов. Я хочу обратить ваше внимание на рис. 10.7, на котором изображены клиент и служба, участвующие во взаимодействии. Эта диаграмма конкретно показывает, что мы добавили повторную отправку запроса к исходящему взаимодействию на стороне клиента и добавили предохранитель перед входом на стороне службы. Второе реализует шаблон шлюза, а первое – это прокси-сервер. Так в чем изюминка? Envoy реализует прокси-сервер на стороне клиента и реверс – прокси/шлюз на стороне службы.

Ладно, это очень круто.

Переделав рис. 10.7 и 10.10 и заменив их на рис. 10.11 и 10.12 соответственно, вы теперь видите, что ключевой элемент в архитектурах для облачной среды, взаимодействие, программируется с помощью сайдкаров. Envoy реализует множество шаблонов на краях этих взаимодействий, среди которых: повторная отправка запроса, предохранители, ограничение скорости, балансировка нагрузки, обнаружение сервисов, наблюдаемость и многое другое. Как я уже неоднократно говорила, хотя вы, будучи разработчиком или архитектором приложений, должны разбираться в шаблонах, описанных в этой книге, вы не всегда несете ответственность за их реализацию. И повторюсь: это очень круто.

Обратите внимание, что теперь взаимодействие происходит между прокси-серверами; то же самое и на рис. 10.12.

Я не учла два других недостатка встраиваемого шлюза, оба из которых сводятся к управляемости прокси-серверов и шлюзов. Вот тут и вступает в дело сервисная сетка.

10.3.2. Уровень управления

Глядя на рис. 10.12, вы видите целую группу прокси-серверов Envoy, соединенных через каналы, которые будут осуществлять взаимодействие между ними. Это похоже на сетку; отсюда и название. Сервисная сеть охватывает набор связанных между собой сайдкаров и добавляет уровень управления для управления этими прокси-серверами.

Одна из наиболее широко используемых сервисных сеток, доступных сегодня, принадлежит Istio (**https://istio.io/**). Это проект с открытым исходным кодом, кото-

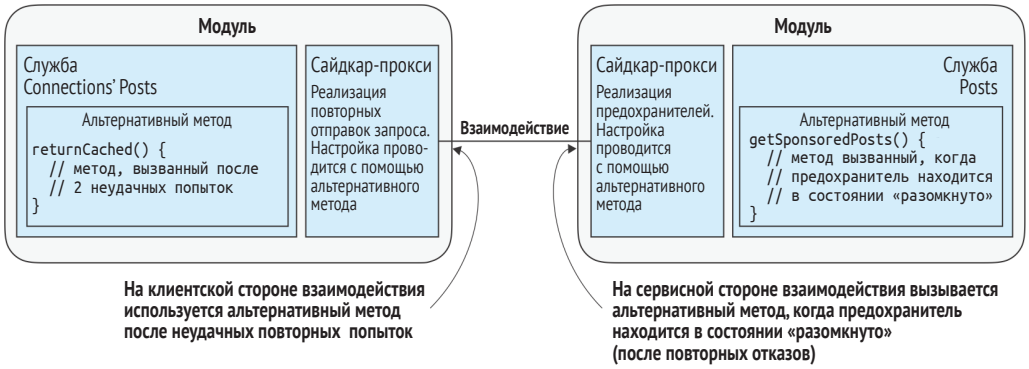


Рис. 10.11 ❖ Новая версия рис. 10.7, где показана повторная отправка запроса на клиентской стороне взаимодействия, реализованная в сайдкаре, а предохранитель на стороне сервиса также реализован в сайдкаре

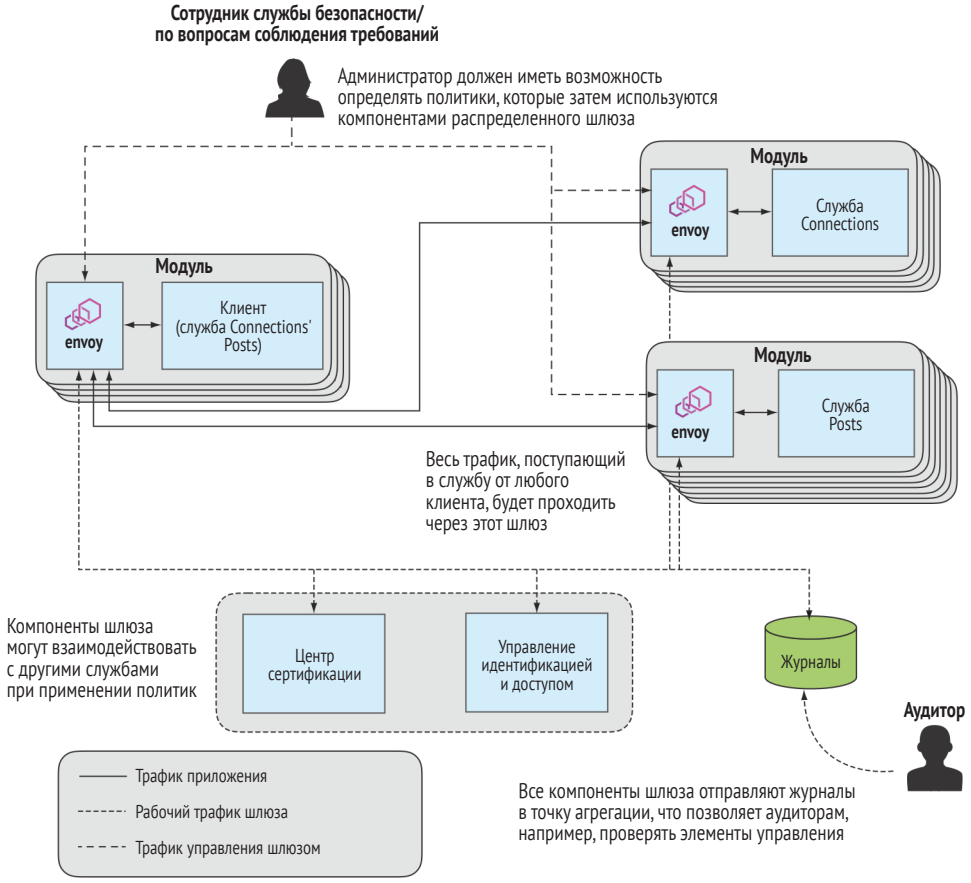


Рис. 10.12 ❖ Абстрактный «шлюз» на рис. 10.10 теперь стал конкретным с помощью сайдкара Envoy, одной из нескольких реализаций сайдкара

рый был инкубирован компаниями Google, IBM и Lyft. Он расширяет Kubernetes, используя модуль-примитив в качестве механизма развертывания сайдкаров Envoy. Слоган Istio – «Подключайтесь, защищайте, контролируйте и наблюдайте за сервисами», и именно это он и делает, поддерживая автоматическое внедрение сайдкаров и предоставляя компоненты, поддерживающие настройку прокси-серверов Envoy, обработку сертификатов и реализацию политик. API уровня управления предлагает интерфейс для этой плоскости управления.

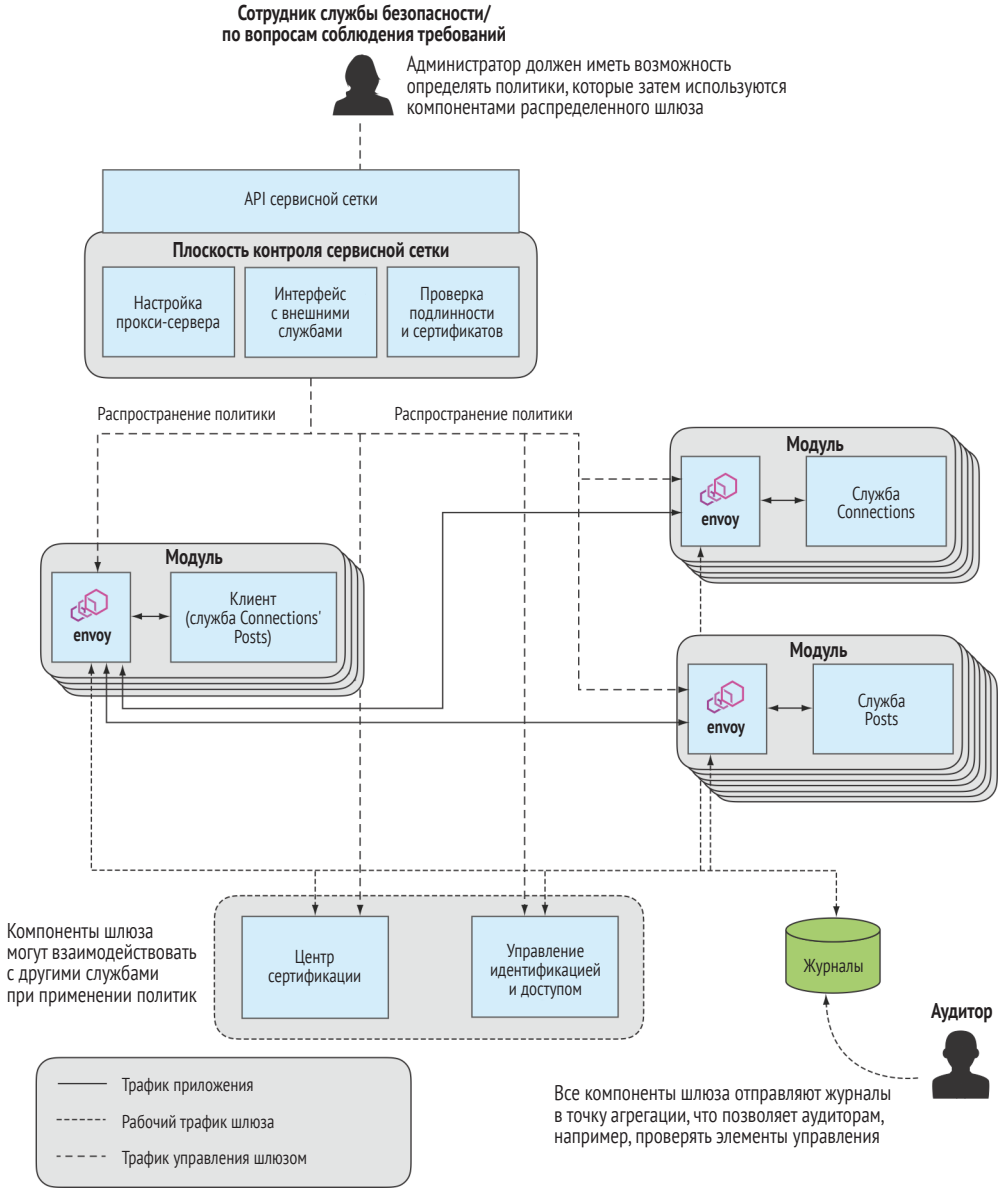


Рис. 10.13 ❖ Сервисная сетка объединяет воедино сайдкары и плоскость контроля для управления ими

Рисунок 10.13 дополняет картину, которую мы получили в этой главе. Эта и предыдущая главы сфокусированы на двух сторонах взаимодействия между сервисами. Поскольку взаимодействие пересекает процессы и иногда даже сетевые границы, для обеспечения надежной реализации программного обеспечения, устойчивой к неизбежным изменениям в распределенных развертываниях на базе облака, необходимы различные шаблоны. Я рассмотрела два ключевых из них: повторная отправка запроса на стороне клиента и предохранители на стороне сервиса, причем последний вариант связан с шаблоном шлюза. В частности, сервисная сетка стала неотъемлемой частью платформы для запуска приложений для облачной среды. Я настоятельно рекомендую вам использовать эту технологию.

Я хочу затронуть одну заключительную тему, посвященную взаимодействию: поиск и устранение неполадок. Независимо от того, является ли основной поток запросом/ответом или управляется событиями, опыт, который пользователь получает при работе с программным обеспечением, отражает работу десятков или даже сотен служб, все из которых взаимодействуют друг с другом. Как же найти коренную причину проблемы, когда что-то идет не так? Об этом рассказывается в следующей главе.

Резюме

- Для размещения на переднем крае службы существует множество шаблонов, которые управляют способом обработки взаимодействия с этой службой.
- Предохранители являются важным шаблоном для защиты сервиса от чрезмерной нагрузки, в том числе для трафика, вызванного шквалом повторных отправок запросов.
- Шлюзы API, предшествующие появлению архитектур программного обеспечения для облачной среды, эволюционировали, чтобы правильно работать в новом контексте сильно распределенных, постоянно меняющихся развертываний.
- Шаблоны, применяемые как на клиентской, так и на сервисной стороне взаимодействия, могут быть инкапсулированы и развернуты в качестве прокси-сервера сайдкара.
- Сервисная сеть дает этому прокси-серверу дополнительную плоскость управления, которая позволяет оператору контролировать безопасность, обеспечивать наблюдаемость и разрешать настройку набора служб/приложений, которые образуют программное обеспечение для облачной среды.

Глава 11

Поиск и устранение неполадок: найти иголку в стоге сена

О чем идет речь в этой главе:

- ведение журналов приложений для сервисов в эфемерном окружении;
- мониторинг приложений для сервисов в эфемерном окружении;
- трассировка распределенных приложений.

В 2013 году я работала с одним из первых корпоративных клиентов платформы с открытым исходным кодом Cloud Foundry. Я навещала определенного клиента каждые пару недель, чтобы проверить, как его успехи, и помочь объяснить возможности, которые в то время были действительно совершенно новыми для этой отрасли. И независимо от того, каким классным был набор функций, которые я описывала, один из инженеров этой компании неизменно говорил в ответ что-то вроде: «Корнелия, вы предлагаете мне Ferrari без приборной панели». Дело в том, что мы еще не проделали основательную работу по добавлению возможностей в область наблюдаемости, и просто не было никакого способа, чтобы этот клиент (или любой другой клиент) мог запустить систему в эксплуатацию без возможности адекватного мониторинга ее работоспособности и работоспособности приложений, работающих на платформе. Этот специалист, Срини, был абсолютно прав!

Решения для наблюдаемости систем и приложений не являются чем-то новым. Большая часть эксплуатационных методов для фрагмента программного обеспечения, часто содержащаяся в перечне задач, сосредоточена на том, как оценить, хорошо ли работает это программное обеспечение, и как, по возможности чем раньше тем лучше, распознать, когда что-то пошло не так. За последние десятилетия появились инструменты и передовые методы, которые помогли превратить задачу, связанную с наблюдаемостью, в надежную практику. Но, как и во многих других устоявшихся аспектах программного обеспечения, архитектура приложений для облачной среды приносит с собой новый набор задач, для решения которых необходимо создать новый набор инструментов и методов. Что представляют из себя новые проблемы сильно распределенного, постоянно меняющегося программного обеспечения?

Как вы уже неоднократно видели, постоянные изменения, о которых я говорю, проявляются в виде эфемерности работающих приложений и окружения, в котором они находятся. Контейнер, в котором работает служба, находится в посто-

янном движении – он удаляется и заменяется новыми экземплярами во время операций жизненного цикла, таких как обновление, или для того, чтобы восстановить экземпляры, с которыми произошло что-то катастрофическое (например, нехватка памяти). Это создает проблемы при использовании многих знакомых методов устранения неполадок, применявшихся в прошлом, которые часто включали в себя поиск подсказок в среде выполнения. Теперь, когда нельзя рассчитывать на наличие этой среды выполнения, как обеспечить доступ к информации, необходимой для диагностики потенциальных проблем?

И сильно распределенная природа нашего программного обеспечения также ставит новые задачи. Когда один пользовательский запрос разветвляется на десятки или сотни последующих запросов, как точно определить причину проблем в такой сложной иерархии? Там, где у вас когда-то было множество компонентов, работающих в одном процессе, и поэтому вы могли перемещаться по стеку вызовов относительно легко, теперь у вас есть вызовы, охватывающие большое количество распределенных сервисов, но вам все равно хотелось бы понять, как выглядит этот «стек вызовов».

Данная глава посвящена обоим этим элементам. Вы узнаете, как создавать и обрабатывать данные журналов и метрические данные таким образом, чтобы учитывать эфемерные среды выполнения. И вы узнаете о трассировке распределенных приложений – наборе методов и инструментов, которые имитируют методы внутрипроцессной трассировки прошлого, позволяя оператору отслеживать поток связанных запросов в распределенной сети микросервисов.

11.1. Ведение журналов приложений

Я не буду тратить время на то, чтобы убедить вас писать записи в журналы; это необходимое, но достаточное условие. Но некоторые из вас, возможно, выполняли управление журналами из приложения; например, вы могли открывать файлы и писать в них. Я буду утверждать, что управление журналами должно полностью выполняться вне кода приложения.

По правде говоря, этот аргумент характерен не только для приложений для облачной среды. Это хорошая идея для любого программного обеспечения. Код приложения должен выражать то, что должно быть зарегистрировано, а место, где появляется эта запись журнала, должно полностью контролироваться развертыванием приложения, а не самим приложением. Такой подход поддерживается множеством фреймворков: например, Apache Log4j и его преемник Logback (**https://logback.qos.ch/**) делают именно так; мы использовали Apache Log4j в наших примерах кода на протяжении всей этой книги. Это просто позволяет писать в коде приложения такие выражения, как

```
logger.info(utils.ipTag() + “New post with title “ + newPost.getTitle());
```

Появляется ли затем это сообщение журнала в определенном файле, или в консоли, или где-то еще, определяется как часть развертывания.

В случае с приложениями для облачной среды эта конфигурация развертывания должна отправлять строки журнала в потоки stdout и stderr. Я знаю, что это довольно самоуверенное утверждение, поэтому позвольте мне немного подкрепить его:

- файлы использовать запрещено. Локальная файловая система существует только до тех пор, пока работает контейнер. Вам потребуется доступ к журналам даже после того, как (смею сказать, особенно после того, как) экземпляр приложения и его контейнер исчезнут. Да, некоторые системы оркестровки контейнеров поддерживают возможность подключения контейнера к внешнему хранилищу, жизненный цикл которого не зависит от жизненного цикла контейнера, но семантика этого процесса сложна, и риск возникновения конфликтов во многих других приложениях и экземплярах приложений довольно значителен;
- во многом благодаря росту популярности открытого исходного кода и сопутствующей ему устойчивости к проприетарным решениям мы стремимся к некоторому уровню стандартизации везде, где можем получить его. Мы не хотим вести журнал одним способом, если выполним развертывание на JBoss, другим способом, если мы сделаем это на WebSphere, и еще одним способом, если мы работаем с Web-Logic. Потоки stdout и stderr вездесущи; здесь нет привязки к поставщику;
- stdout и stderr являются независимыми не только от поставщика, но и от операционной системы. Концепции будут одинаковыми, будь то Linux, Windows или другая ОС, а реализации предоставляют те же возможности;
- stdout и stderr – это потоковые API, а журналы, безусловно, в большинстве своем являются потоками. У них нет начала или конца; наоборот, записи просто продолжают течь. Поскольку эти записи появляются в потоке, система обработки потоков может соответствующим образом обрабатывать их.

Итак, разработчикам приложений не нужно больше заботиться о чем-то еще, кроме вызова метода для логгера (например), но давайте на минутку поговорим о том, как обрабатываются журналы. Как и во многих других темах, которые мы уже обсуждали, платформы – ваши друзья. Показательный пример: вы уже достаточно попрактиковались в качестве пользователя функций обработки журналов в Kubernetes. Экземпляры наших приложений используют объекты SLF4J (фасад фреймворков для ведения журналов, таких как Logback), чтобы создавать записи журнала, которые отправляются в потоки stdout и stderr. Когда вы выполняете такую команду, как `kubectl logs -f pod/posts-fc74d75bc-92txh`, подключится интерфейс командной строки Kubernetes и выведет записи потока на ваш терминал.

О чем я не рассказывала – так это о том, как работает журналирование, когда у вас много экземпляров сервиса. По большей части мы имели дело с потоковой передачей журналов для одного экземпляра приложения. Но в некоторых ситуациях вас может заинтересовать просмотр журналов какого-либо приложения, независимо от количества запущенных экземпляров. Например, возможно, вам нужно проверить, обрабатывался ли конкретный запрос каким-либо из ваших экземпляров приложения, без необходимости проверять журналы каждого экземпляра по отдельности. Kubernetes позволит вам сделать это с помощью такой команды:

```
$ kubectl logs -l app=posts
2018-12-02 22:41:42.644 ... s.c.a.AnnotationConfigApplicationContext ...
2018-12-02 22:41:43.582 ... trationDelegate$BeanPostProcessorChecker ...
```

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__' | / / / /
 =========|_|==============|___/=/_/_/_/
:: Spring Boot :: (v1.5.6.RELEASE)
2018-12-02 22:41:44.309 ... c.c.c.ConfigServicePropertySourceLocator ...
...
2018-12-02 22:42:38.098 : [10.44.4.61:8080] Accessing posts using secret
2018-12-02 22:42:38.102 : [10.44.4.61:8080] getting posts for userId 2
2018-12-02 22:42:38.119 : [10.44.4.61:8080] getting posts for userId 3
2018-12-02 22:42:40.806 : [10.44.4.61:8080] Accessing posts using secret
2018-12-02 22:42:40.809 : [10.44.4.61:8080] getting posts for userId 2
2018-12-02 22:42:40.819 : [10.44.4.61:8080] getting posts for userId 3
2018-12-02 22:42:43.399 : [10.44.4.61:8080] Accessing posts using secret
2018-12-02 22:42:43.399 : [10.44.4.61:8080] getting posts for userId 2
2018-12-02 22:42:43.408 : [10.44.4.61:8080] getting posts for userId 3
2018-12-02 22:53:27.039 : [10.44.4.61:8080] Accessing posts using secret
2018-12-02 22:53:27.039 : [10.44.4.61:8080] getting posts for userId 2
2018-12-02 22:53:27.047 : [10.44.4.61:8080] getting posts for userId 3
2018-12-02 22:41:21.155 ... s.c.a.AnnotationConfigApplicationContext ...
2018-12-02 22:41:22.130 ... trationDelegate$BeanPostProcessorChecker ...
. ____ _ __ _ _
/\\ / ___’_ __ _ _(_)_ __ __ _ \ \ \ \
( ( )\___ | ‘_ | ‘_| | ‘_ \/ _` | \ \ \ \
\\/ ___)| |_)| | | | | || (_| | ) ) ) )
‘ |____| .__|_| |_|_| |_\__, | / / / /
=========|_|==============|___/=/_/_/_/
:: Spring Boot :: (v1.5.6.RELEASE)
2018-12-02 22:41:23.085 ... c.c.c.ConfigServicePropertySourceLocator ...
...
2018-12-02 22:42:46.297 : [10.44.2.57:8080] Accessing posts using secret
2018-12-02 22:42:46.298 : [10.44.2.57:8080] getting posts for userId 2
2018-12-02 22:42:46.305 : [10.44.2.57:8080] getting posts for userId 3
2018-12-02 22:53:30.260 : [10.44.2.57:8080] Accessing posts using secret
2018-12-02 22:53:30.260 : [10.44.2.57:8080] getting posts for userId 2
2018-12-02 22:53:30.266 : [10.44.2.57:8080] getting posts for userId 3
```

При более внимательном рассмотрении этого вывода вы заметите, что в первой части показаны записи журнала из одного экземпляра модуля, за которыми следуют записи из второго экземпляра; журналы из нескольких экземпляров не чередуются. Во многих случаях может быть полезно просматривать сообщения по времени во всех экземплярах. Например, вот расстановка предыдущих журналов:

```
2018-12-02 22:41:21.155 ... s.c.a.AnnotationConfigApplicationContext ...
2018-12-02 22:41:22.130 ... trationDelegate$BeanPostProcessorChecker ...
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__' | / / / /
 =========|_|==============|___/=/_/_/_/
```

```
:: Spring Boot :: (v1.5.6.RELEASE)
2018-12-02 22:41:23.085 ... c.c.c.ConfigServicePropertySourceLocator ...
2018-12-02 22:41:42.644 ... s.c.a.AnnotationConfigApplicationContext ...
2018-12-02 22:41:43.582 ... trationDelegate$BeanPostProcessorChecker ...
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__' | / / / /
 =========|_|==============|___/=/_/_/_/
:: Spring Boot :: (v1.5.6.RELEASE)
2018-12-02 22:41:44.309 ... c.c.c.ConfigServicePropertySourceLocator ...
...
2018-12-02 22:42:38.098 : [10.44.4.61:8080] Accessing posts using secret
2018-12-02 22:42:38.102 : [10.44.4.61:8080] getting posts for userId 2
2018-12-02 22:42:38.119 : [10.44.4.61:8080] getting posts for userId 3
2018-12-02 22:42:40.806 : [10.44.4.61:8080] Accessing posts using secret
2018-12-02 22:42:40.809 : [10.44.4.61:8080] getting posts for userId 2
2018-12-02 22:42:40.819 : [10.44.4.61:8080] getting posts for userId 3
2018-12-02 22:42:43.399 : [10.44.4.61:8080] Accessing posts using secret
2018-12-02 22:42:43.399 : [10.44.4.61:8080] getting posts for userId 2
2018-12-02 22:42:43.408 : [10.44.4.61:8080] getting posts for userId 3
2018-12-02 22:42:46.297 : [10.44.2.57:8080] Accessing posts using secret
2018-12-02 22:42:46.298 : [10.44.2.57:8080] getting posts for userId 2
2018-12-02 22:42:46.305 : [10.44.2.57:8080] getting posts for userId 3
2018-12-02 22:53:27.039 : [10.44.4.61:8080] Accessing posts using secret
2018-12-02 22:53:27.039 : [10.44.4.61:8080] getting posts for userId 2
2018-12-02 22:53:27.047 : [10.44.4.61:8080] getting posts for userId 3
2018-12-02 22:53:30.260 : [10.44.2.57:8080] Accessing posts using secret
2018-12-02 22:53:30.260 : [10.44.2.57:8080] getting posts for userId 2
2018-12-02 22:53:30.266 : [10.44.2.57:8080] getting posts for userId 3
```

Хотя может быть полезно увидеть журналы, агрегированные таким образом, с чередованием записей по времени, почти всегда важно, чтобы записи журнала можно было отнести к конкретным экземплярам приложения. В этих записях журнала это видно по IP-адресу: один экземпляр имеет IP-адрес 10.44.4.61, а другой – 10.44.2.57. В идеале обозначение экземпляра должно добавляться фреймворком или платформой, чтобы код приложения мог быть независимым от среды выполнения.

Kubernetes этого не делает. IP-адрес и порт, которые вы видите здесь, добавляются через пакет `Utils` из нашей реализации. Я, используя руководство из главы 6 о конфигурации приложения, позаботилась о том, чтобы абстрагировать детали платформы, вводя IP-адрес через переменную среды, но я бы предпочла, чтобы платформа включала подобное обозначение, без всяких усилий со стороны разработчика. Вывод заключается в том, что вам как разработчику приложения может потребоваться обратить внимание на то, чтобы в выводе вашего журнала содержалась информация, идентифицирующая, к какому экземпляру приложения относится запись.

Когда я говорю о платформе для обработки журналов, агрегация является лишь одним из необходимых элементов. Журналы должны поступать в широких

масштабах и должны сохраняться, а интерфейсы должны поддерживать поиск и анализ этих потенциально больших объемов данных. Стек ELK (**www.elastic.co/elk-stack**) объединяет три проекта с открытым исходным кодом – Elasticsearch, Logstash и Kibana – для удовлетворения этих требований. Коммерческие предложения, такие как Splunk, предоставляют сопоставимые возможности. Когда вы отправляете свои журналы в потоки stdout и stderr и гарантируете, что записи относятся к конкретным экземплярам приложения, вы делаете все, что необходимо, для того чтобы такие системы могли обеспечивать мощные функции наблюдаемости. И вы разрешите сохранение записей журнала, даже когда контейнеры приложений исчезнут.

11.2. Метрики приложений

В дополнение к данным журнала метрики необходимы для целостного мониторинга приложений.

Обычно метрики обеспечивают более детальное представление о работающих приложениях, по сравнению с файлами журналов; метрики структурированы, тогда как файлы журналов обычно неструктурированы или, в лучшем случае, структурированы частично. Фреймворки практически всегда используются как для автоматической генерации набора метрик по умолчанию, так и для выдачи специальных метрик кода. Метрики по умолчанию обычно включают в себя значения, касающиеся использования памяти и ЦП, а также взаимодействия по протоколу HTTP (при необходимости). Для таких языков, как Java, также часто включаются метрики для сборки мусора и загрузчик классов.

Вы уже использовали Spring Framework для работы с метриками, просто включив в его состав зависимость Actuator. В дополнение к конечной точке `/actuator/env`, которую вы использовали ранее, Actuator предоставляет конечную точку `/actuator/metrics`, которая генерирует стандартные и специальные метрики для ваших приложений на Spring Boot. Ниже приводится вывод, обслуживаемый этой конечной точкой в службе Connections' Posts:

```
$ curl 35.232.22.58/actuator/metrics | jq
{
  “mem”: 853279,
  “mem.free”: 486663,
  “processors”: 2,
  “instance.uptime”: 2960448,
  “uptime”: 2975881,
  “systemload.average”: 1.33203125,
  “heap.committed”: 765440,
  “heap.init”: 120832,
  “heap.used”: 278776,
  “heap”: 1702400,
  “nonheap.committed”: 90584,
  “nonheap.init”: 2496,
  “nonheap.used”: 87839,
  “nonheap”: 0,
  “threads.peak”: 43,
  “threads.daemon”: 41,
```

```
  “threads.totalStarted”: 63,
  “threads”: 43,
  “classes”: 8581,
  “classes.loaded”: 8583,
  “classes.unloaded”: 2,
  “gc.ps_scavenge.count”: 1019,
  “gc.ps_scavenge.time”: 8156,
  “gc.ps_marksweep.count”: 3,
  “gc.ps_marksweep.time”: 643,
  “httpsessions.max”: -1,
  “httpsessions.active”: 0,
  “gauge.response.metrics”: 1,
  “gauge.response.connectionPosts”: 56,
  “gauge.response.star-star”: 20,
  “gauge.response.login”: 2,
  “counter.span.accepted”: 973,
  “counter.status.200.metrics”: 3,
  “counter.status.404.star-star”: 1,
  “counter.status.200.connectionPosts”: 32396,
  “counter.status.200.login”: 53
}
```

В дополнение к значениям, указанным для памяти, потоков и загрузчика классов, обратите внимание на то, что этот экземпляр успешно обработал огромное множество (32 396) результатов на конечной точке `/connectionsposts`, несколько результатов (53) на конечной точке `/login` и три результата на конечной точке `/actuator/metrics`, которую вы используете для получения этих данных. Он также ответил один раз ошибкой с кодом состояния 404.

Метрики приложений широко используются гораздо дольше, чем приложения для облачной среды, и опять же, я хочу сосредоточиться на изменениях в этом новом облачном контексте. Как и в случае с журналами, основной проблемой является обеспечение доступности данных метрик даже после того, как среда выполнения перестает быть доступной. Вам необходимо извлечь метрики из контекста приложения и среды выполнения, и для этого есть два основных подхода: модель на базе извлечения и модель на базе размещения.

11.2.1. Извлечение метрик

При подходе на базе извлечения агрегатор метрик реализуется в качестве сборщика, который запрашивает данные метрики из каждого экземпляра приложения и сохраняет эти метрики в базе данных временных рядов (рис. 11.1). Это немного похоже на использование команды `curl` с конечной точкой `/actuator/metrics`, что вы видели только что. Сборщик, будучи клиентом, делает запрос, а экземпляр приложения отвечает необходимыми данными.

Но то, что мы сделали, используя команду `curl`, не совсем правильно. В те периоды, когда у вас был только один экземпляр приложения или вы рассматривали каждый из нескольких экземпляров как отдельную сущность, выполнение запроса через протокол HTTP было вполне приемлемым; вы могли нацеливаться на каждый экземпляр приложения напрямую. Но теперь, когда у вас есть несколько экземпляров приложения, которые сбалансированы по нагрузке, вы будете полу-

чать метрики только от одного экземпляра, и кроме того, вы не знаете, от какого именно. Звучит знакомо, правда? Это та же проблема, что была у вас при работе с записями журнала, не связанными с конкретным экземпляром, проблема, которую вы, по крайней мере, частично решили с помощью пакета `Utils`, который включал в себя идентификатор экземпляра в записи журнала. Но есть кое-что еще: при сборе метрик вам нужно последовательно собирать значения для каждого экземпляра через регулярные промежутки времени, а балансировщики нагрузки, как правило, не будут распределять запросы так равномерно, как вам это нужно.

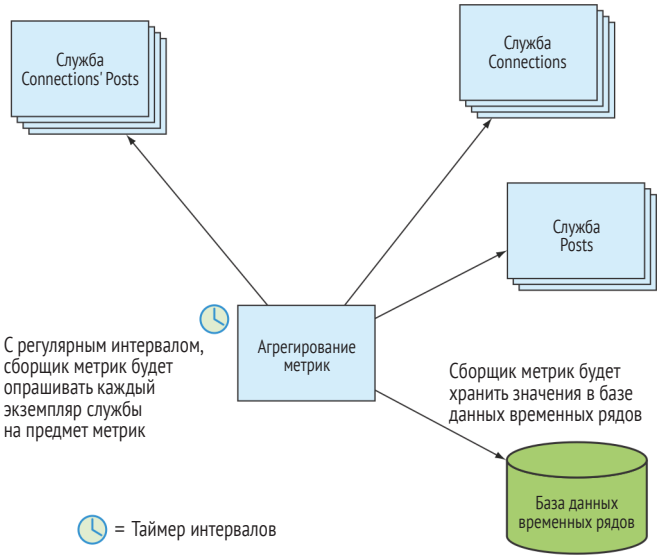


Рис. 11.1 ❖ При таком подходе к сбору метрик в каждой службе реализована конечная точка метрик, к которой осуществляется доступ через регулярные интервалы для сбора и хранения значений для последующего поиска и анализа

Решение состоит в том, чтобы сборщик полностью контролировал, из каких экземпляров приложения он будет извлекать метрики и с каким интервалом. Ему нужно контролировать, где делаются запросы, вместо того чтобы позволять балансировщику нагрузки выбирать; опять же, надеюсь, это звучит знакомо. Это похоже на балансировку нагрузки на стороне клиента, о которой вы узнали в главе 8, а часть балансировки нагрузки на стороне клиента – это обнаружение сервисов. На рис. 11.2 изображен поток:

- на каждом интервале сборщик запрашивает данные метрик из каждого экземпляра;
- набор экземпляров обнаруживается с помощью протокола обнаружения сервисов, и частота, с которой сборщик вызывает этот протокол для получения последних идентификаторов экземпляров, может различаться. Делать это на каждом интервале может быть затратно, но это гарантирует, что любые изменения в топологии приложения будут отражены как можно быст-

рее. Если допустимо, чтобы метрики нового экземпляра были на короткое время исключены из коллекции, протокол обнаружения сервисов может выполняться реже;

- выполнение обнаружения сервисов с интервалом, отдельным от сбора метрик, дает более слабосвязанное решение.

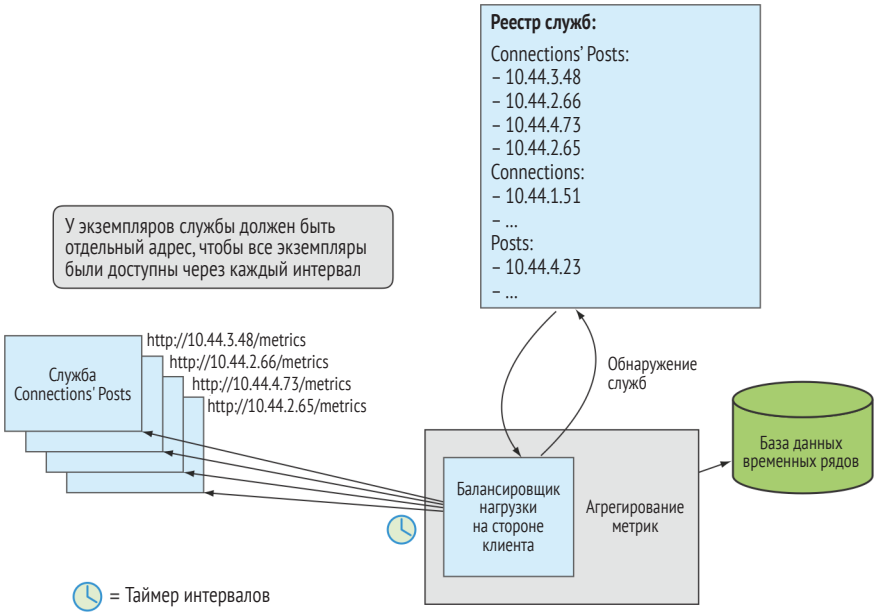


Рис. 11.2 ❖ Сборщик, который реализует агрегатор метрик, должен получать доступ к каждому экземпляру службы через каждый интервал и, следовательно, должен управлять балансировкой нагрузки. Он будет взаимодействовать через протокол обнаружения служб, чтобы быть в курсе изменений IP-адресов

Одна из сложностей подхода на базе извлечения состоит в том, что сборщик, конечно же, должен иметь доступ к каждому экземпляру, из которого он будет запрашивать данные. IP-адрес каждого экземпляра должен быть адресуемым из агрегатора метрик. Часто экземпляры сервисов могут быть адресуемы только индивидуально из среды выполнения.

Вы видели это в развертываниях наших образцов, где из-за пределов кластера была доступна только служба Connections' Posts, поэтому сборщик метрик также должен быть развернут в этом сетевом пространстве. Общая топология развертывания для окружений на базе Kubernetes – это развертывание Prometheus (**https://prometheus.io/**) *внутри* самого кластера Kubernetes. В этой топологии Prometheus использует встраиваемую DNS-службу и имеет прямой доступ к экземплярам приложений (см. рис. 11.3).

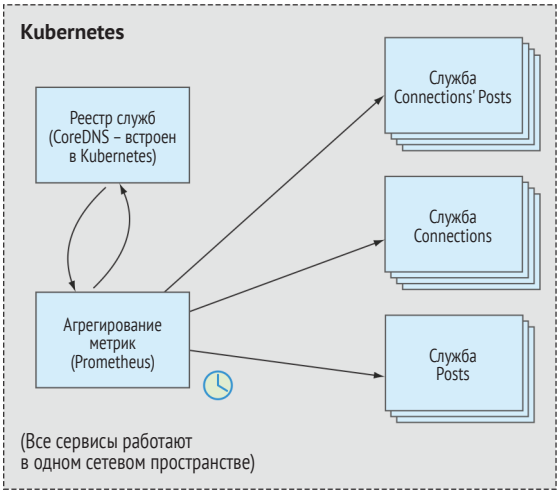


Рис. 11.3 ❖ Сборщик метрик должен присваивать адрес каждому экземпляру службы индивидуально, и поскольку эти IP-адреса доступны только в среде выполнения, например Kubernetes (внешний доступ осуществляется через балансировщик нагрузки), сборщик метрик также обычно развертывается в этом сетевом пространстве. В случае с Kubernetes это тоже позволяет использовать встроенную службу DNS для обнаружения сервисов

11.2.2. Размещение метрик

Альтернативой модели на базе извлечения, чтобы осуществлять сбор метрик приложения, является модель, основанная на размещении, где каждый экземпляр приложения отвечает за доставку метрик в агрегатор через регулярные промежутки времени (рис. 11.4). Будучи разработчиком приложения, вы можете отказаться от бремени доставки данных метрик – работа над кодом для реализации этого процесса уводит вас от основной бизнес-логики, которая приносит желаемый результат вашим клиентам и компании. Хорошая новость заключается в том, что, как и значительная часть внедряемых действий, о которых я говорила здесь, большая часть работы по созданию и доставке метрик выполняется нашими надежными фреймворками и платформами.

Фреймворки, предоставляющие реализации метрик на базе размещения, обычно делают это с помощью агента, который заботится о сборе и доставке метрик в агрегатор. Агент обычно компилируется в двоичный файл приложения с включением зависимости в нечто наподобие проектно-объектной модели (POM) или файла сборки Gradle. Более сложным, из-за постоянно меняющейся среды, в которой находятся наши приложения и агенты, является правильная настройка этого агента при развертывании и во время текущего управления системами.

Например, IP-адрес агрегатора метрик должен быть настроен в работающем приложении, чтобы агент знал, куда отправлять метрики. Используя передовые методы, описанные в главе 6, это просто при первоначальном развертывании, но внесение изменений в конфигурацию уже запущенного приложения должно выполняться с осторожностью, как описано в главе 7. Возможно, вы думаете, что это похоже на стандартное обнаружение сервисов (что также обсуждалось ранее, в главе 8), но поскольку доставка метрик часто требует значительных ресурсов, добавление протокола обнаружения сервисов в поток доставки метрик может сгенерировать недопустимую задержку.

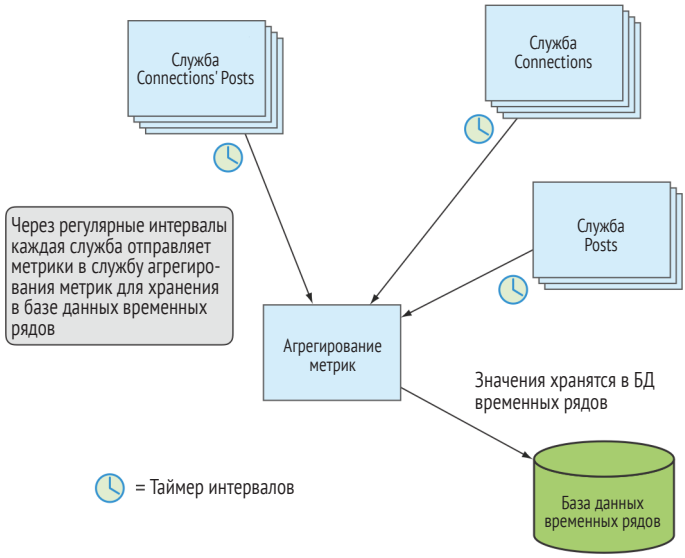


Рис. 11.4 ❖ При таком подходе каждая служба отправляет метрики службе агрегирования и хранения через определенный интервал

Еще одна обратная ссылка: в главе 10 я говорила о сайдкаре, который обеспечивает функциональность шлюза API и реализует протоколы при использовании повторных отправок запросов и предохранителей. Но сайдкар также идеально подходит для сбора метрик. Напомню, что сайдкар адресуем из других контейнеров в модуле через локальный хост, эффективно защищая приложения от изменений в службе сбора метрик. Агент в приложении просто доставляет метрики в сайдкар, который затем берет на себя ответственность за пересылку данных на внешний сборщик (рис. 11.5). Если координаты сборщика меняются, конфигурация приложения остается неизменной, и, следовательно, никаких операций жизненного цикла приложения не требуется. Сайдкар/сервисная сеть специально разработана для обработки постоянных изменений, присутствующих в приложениях для облачной среды, и теперь он берет на себя эту ответственность. Например, плоскость управления для сервисной сетки может помещать любой новый IP-адрес в сетку, обновляя все сайдкары. А такие сайдкары, как Envoy, предназначены для более легкой адаптации к изменениям конфигурации приложений с помощью таких возможностей, как «горячий» перезапуск.

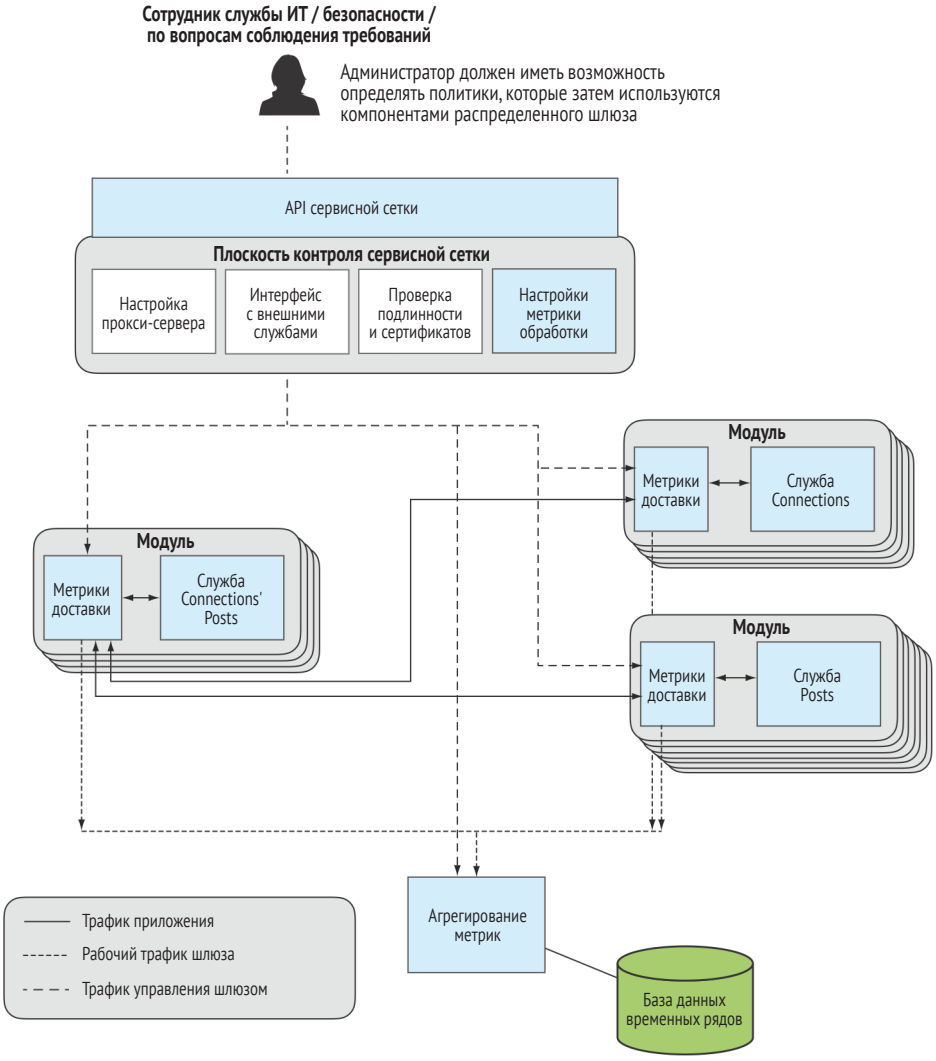


Рис. 11.5 ❖ При использовании сервисной сетки службы приложений просто настраиваются для подключения к локальному сайдкар-прокси, а плоскость управления сервисной сетки используется для поддержания конфигурации компонентов доставки метрик в актуальном состоянии

Вы заметили? В предыдущем обсуждении я дала большое количество ссылок на предыдущие главы. Решать проблемы управления метрикой для приложений облачной среды лучше всего с помощью шаблонов. В этом разделе приводится отличный пример.

И наконец, наличие сайдкара может дать желаемые результаты помимо простого проксирования исходящих метрик, поскольку оно также может обеспечить некоторый уровень наблюдаемости, даже если в приложение не установлен агент. Поскольку он проксирует трафик, поступающий и выходящий из приложения, он

может генерировать множество метрик от имени приложения. Например, количество кодов состояния HTTP, задержки и т. д. можно собирать или рассчитывать и доставлять, ничего не делая в коде приложения.

Это выдающийся пример умной архитектуры и инновационных фреймворков, позволяющих отделить бизнес-задачи в приложении от задач по эксплуатации. Использование правильной платформы может избавить разработчика приложений от множества проблем, позволяя ему сосредоточиться на бизнес-результатах своего кода.

11.3. Распределенная трассировка

Давайте рассмотрим еще одну возможность, которую может дать сочетание фреймворков приложений и платформ для облачной среды. Речь идет о *распределенной трассировке*, которая критически важна для сильно распределенного приложения для облачной среды.

В окружении, где весь наш код выполняется в одном и том же процессе, мы можем использовать хорошо зарекомендовавшие себя инструменты для отслеживания и устранения неполадок в процессе выполнения приложения. Отладка на уровне исходного кода будет перескакивать от метода к методу, а при правильной настройке даже перейдет в код, который был введен в приложение посредством включения библиотеки (кода, который вы не написали). При возникновении исключений стек вызовов, который отображается в консоли и/или выводится в журналы, показывает последовательность выполненных вызовов, что часто помогает при диагностике проблем.

Но теперь вызов вашего приложения может привести к потоку нисходящих запросов, которые обычно выполняются вне процесса и фактически чаще всего в совершенно разных контекстах времени выполнения (в разных контейнерах или на разных хостах). Как увидеть эквивалент стека вызовов или просто получить представление о том, что происходит в результате вызова приложения в этом распределенном сценарии? Метод, получивший широкое распространение в этой отрасли и обладающий солидным инструментарием, – это распределенная трассировка.

Распределенная трассировка полностью соответствует своему названию и предназначена для трассировки хода выполнения программ по распределенному набору компонентов. Это то, что позволяет нам получить представление о разветвлении всех нисходящих запросов, например в результате доступа к домашней странице Netflix. На рис. 11.6 точка слева обозначает запрос к домашней странице, а линии, ведущие к другим точкам, обозначают вызовы к дополнительным сервисам, осуществляемые для сбора содержимого домашней страницы пользователя.

Довольно популярной технологией, используемой сегодня, является Zipkin (**https://zipkin.io/**), проект, появившийся после исследования в области распределенной трассировки, которое впервые было опубликовано в статье Google в 2010 году[1]. Вот что лежит в основе этого метода:

- использование *трассировщиков*, уникальных идентификаторов, которые вставляются в запросы и ответы для поиска связанных вызовов приложений;

[1] Эта статья доступна по адресу **http://mng.bz/178V**.

- плоскость управления, которая использует эти трассировщики для сборки графа вызовов для набора того, что в противном случае является независимыми (по замыслу!) вызовами.

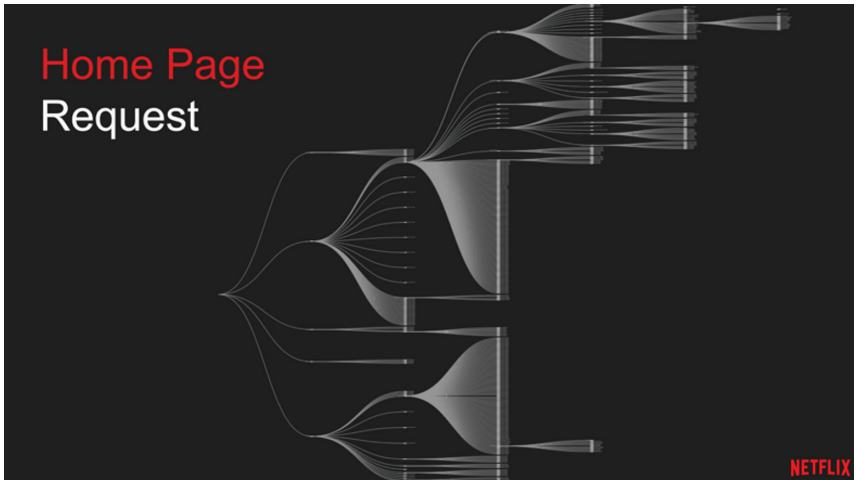


Рис. 11.6 ❖ Диаграмма, показанная во время презентации Скотта Мэнсфилда из компании Netflix, демонстрирует, что запрос к домашней странице Netflix приводит к серии нисходящих вызовов сервисов. Распределенная трассировка позволяет вам увидеть это сложное дерево вызовов

Когда служба вызывается, и эта служба отправляет нисходящий запрос другой службе, любой трассировщик, включенный в запрос к первой службе, будет передан второй службе. Этот трассировщик затем становится доступным в контексте времени выполнения каждой службы и – *вот в чем суть* – может быть включен в любой показатель или вывод журнала. Использование этого трассировщика наряду с другими данными из контекста сервисов (например, временных меток) позволяет объединить поток через набор сервисов, которые вкупе создают ответ на запрос сервиса.

На рис. 11.7 вы видите набор сервисов и вызовов, в которых содержатся эти трассировщики. На этой диаграмме также изображена база данных, которая собирает вывод каждой службы – данные, которые включают в себя значения трассировщика. Из хранящихся там данных вы можете перестроить «стек вызовов» для набора вызовов связанных компонентов. Например, видно, что запрос, поступающий в службу A, создал последующий вызов службы C; а несвязанный запрос к службе B также привел к нисходящему запросу службы C, за которым последовал запрос к службе D.

Чтобы это выглядело более конкретным, давайте выполним наш пример и посмотрим на новый вывод.

Настройка

И снова я отсылаю вас к инструкциям по настройке запуска примеров из предыдущих глав. Запуск образца в этой главе не несет в себе никаких новых требований.

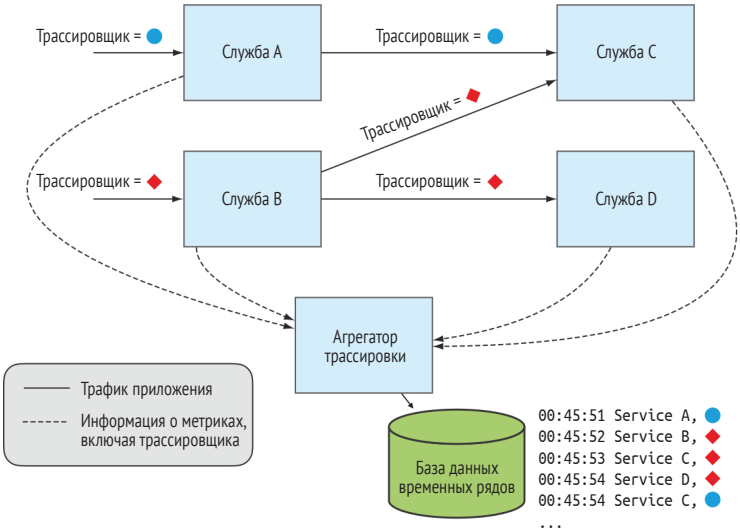


Рис. 11.7 ❖ Запросы содержат трассировщики, которые распространяются в насходящих запросах. Затем трассировщики становятся доступны в контексте времени выполнения вызова службы и аннотируют данные, которые объединяются в службу распределенной трассировки

Вы будете получать доступ к файлам в каталоге cloudnative-troubleshooting, поэтому перейдите в этот каталог в окне терминала.

И, как я уже писала в предыдущих главах, я уже выполнила предварительную сборку образов Docker и сделала их доступными в Docker Hub. Если вы хотите собрать исходный код Java и образы Docker и перенести их в собственный репозиторий образов, я отсылаю вас к предыдущим главам (самые подробные инструкции приведены в главе 5).

Запуск приложения

Вам понадобится кластер Kubernetes с достаточной емкостью, как описано в первом примере главы 9. Если примеры из предыдущей главы у вас по-прежнему работают, давайте очистим их. Запустите предоставленный мной скрипт:

```
./deleteDeploymentComplete.sh all
```

В результате этого будут удалены все экземпляры служб Posts, Connections и Connections' Posts, а также MySQL, Redis и SCCS, которые работают. Если в вашем кластере Kubernetes работает что-то другое, возможно, вы захотите очистить и их. Просто убедитесь, что у вас достаточно вместимости.

Существует небольшая зависимость порядка старта. После создания сервера MySQL вам необходимо создать на нем действительные базы данных, поэтому давайте сначала запустим его и другие сторонние службы:

```
./deployServices.sh
```

После того как база данных MySQL запущена и работает, что можно увидеть, выполнив команду `kubectl get all`, вы создадите базу данных с помощью интерфейса командной строки MySQL:

```
mysql -h <public IP address of your MySQL service> \
 -P <port for your MySQL service> -u root -p
```

Пароль – password. После входа вы выполните команду для создания базы данных:

```
create database cookbook;
```

Теперь вы можете запустить микросервисы, выполнив этот скрипт:

```
./deployApps.sh
```

Сейчас я подробно расскажу о реализации, но для начала давайте вызовем нашу службу сообщений Connections' Posts и просмотрим вывод журнала. Сначала вы выполните вход с помощью этой команды:

```
curl -i -X POST -c cookie \
  <connectionsposts-svc IP>/login?username=cdavisafc
```

А затем получите свою службу Connections' Post:

```
curl -b cookie <connectionsposts-svc IP>/connectionsposts | jq
```

Теперь давайте посмотрим журналы каждого из наших микросервисов. Как я уже говорила в предыдущих разделах, поскольку у вас есть несколько экземпляров каждого микросервиса, какой-нибудь тип агрегации журналов был бы полезен, и хотя тот, что предлагается в Kubernetes, мог бы быть и лучше, нам этого будет достаточно. Выполните каждую из приведенных ниже команд, а потом изучите результаты:

```
kubectl logs -l app=connectionsposts
kubectl logs -l app=connections
kubectl logs -l app=posts
```

11.3.1. Вывод трассировщика

Следующие три листинга вывода журнала являются выдержками из вывода трех этих команд.

Листинг 11.1 ❖ Вывод журнала из службы Connections' Posts

```
2019-02-25 02:20:11.969 [mycookbook-connectionsposts,2e30...,2e30...]
➥ getting posts for user network cdavisafc
2019-02-25 02:20:11.977 [mycookbook-connectionsposts,2e30...,2e30...] connections = 2,3
```

Листинг 11.2 ❖ Вывод журнала из службы Connections

```
2019-02-25 02:20:11.974 [mycookbook-connections,2e30...,9b5f...] getting
➥ connections for username cdavisafc
2019-02-25 02:20:11.974 [mycookbook-connections,2e30...,9b5f...] getting user cdavisafc
...
2019-02-25 02:20:11.987 [mycookbook-connections,2e30...,b915...] getting user 2
...
2019-02-25 02:20:11.994 [mycookbook-connections,2e30...,990f...] getting user 3
```

Листинг 11.3 ❖ Вывод журнала из службы Posts

```
2019-02-25 02:20:11.980 [mycookbook-posts,2e30...,33ac...] Accessing posts using secret ...
2019-02-25 02:20:11.980 [mycookbook-posts,2e30...,33ac...] getting posts for userId 2
2019-02-25 02:20:11.981... [mycookbook-posts,2e30...,33ac...] getting posts for userId 3
```

Вывод журнала теперь включает в себя новые значения, заключенные в квадратные скобки. Первое – это имя приложения. Второе – это идентификатор трассировки; это именно тот трассировщик, о котором я говорила. Третье значение – это идентификатор отдельного запроса, используемый для идентификации каждого уникального вызова приложения; например, в предыдущем выводе журнала приложение Connections вызывалось три раза, как указано тремя идентификаторами отдельных запросов (9b5f…, b915… и 990f…). Идентификатор можно использовать для корреляции метрик или выходных данных журнала, которые являются частью выполнения одного сервиса.

Изучая предыдущий вывод, где идентификаторы трассировки и отдельных запросов усекаются до первых четырех цифр в шестнадцатеричных числах, сгенерированных Spring Framework, видно следующее:

- когда вы используете команду `curl` при вызове службы Connections' Posts, генерируется идентификатор трассировки, начинающийся с 2e30;
- поскольку этот вызов – самый ближний внешний, данный номер также является идентификатором отдельного запроса (2e30…) и представляет собой работу, выполняемую для создания списка постов от пользователей, на которых подписан `cdavisafc`;
- любой вывод журнала из службы Connections' Posts содержит значения идентификатора трассировки и идентификатора отдельного запроса;
- служба Connections была вызвана три раза:
  - поскольку все выходные данные содержат идентификатор трассировки 2e30, вы знаете, что все эти вызовы были нисходящими запросами от вашей команды `curl` к службе Connections' Posts;
  - поскольку эти выходные данные имеют три идентификатора отдельного запроса, вы знаете, что служба Connections была вызвана три раза;
- служба Posts была вызвана один раз. Поскольку идентификатор трассировки – 2e30, вы знаете, что этот вызов является нисходящим запросом от вашей исходной команды `curl`;
- наконец, в начале каждой строки журнала выводится временная отметка.

Эти данные позволяют собрать воедино часть потока, как показано на рис. 11.8. Точки сопровождаются временной отметкой, показывающей моменты, когда был сгенерирован вывод журнала. На этой диаграмме показано следующее:

- вызывается служба Connections' Posts (с использованием исходной команды `curl`);
- вызывается служба Posts (чтобы получить список пользователей, на которых я подписана);
- вызывается служба Connections (со списком этих соединений, чтобы получить список постов для этих соединений);
- (для каждого поста, возвращаемого оттуда, а их два) вызывается служба Connections.

Я заключила в скобки части предыдущего потока, потому что это та семантика, которая вам и мне известна о нашем приложении, используемом в качестве примера, а не заключенные в скобки части фраз – это то, что можно увидеть в выводе трассировки. Это интересно; на рис. 11.8 вы не замечаете деталей, которые находятся в скобках в предыдущем описании. Рисунок 11.9 заполняет детали, которые взяты из этих дополнительных фраз.

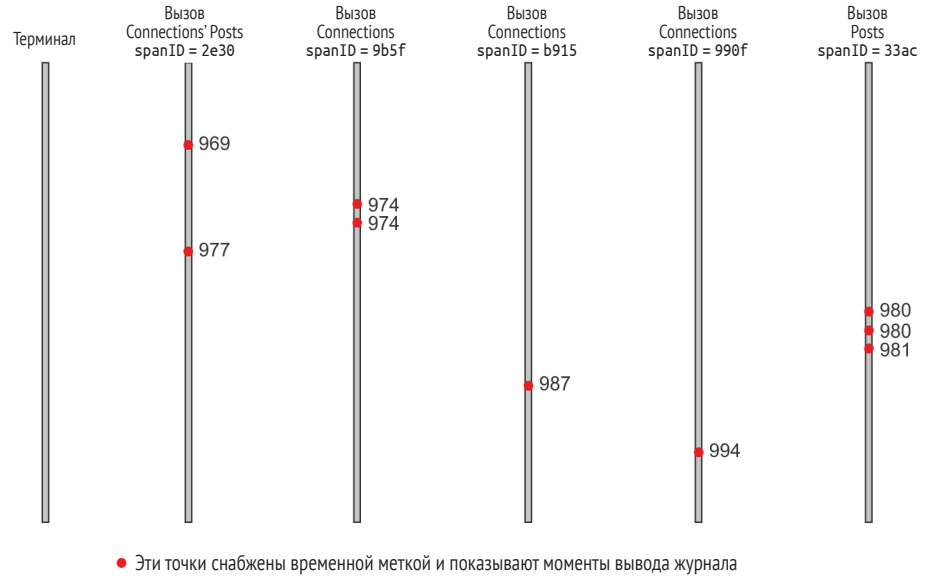


Рис. 11.8 ❖ Из аннотированного вывода журнала можно собрать воедино часть потока одного запроса – с идентификатором трассировки 2e30. Обратите внимание, что вы не видите, откуда шли вызовы

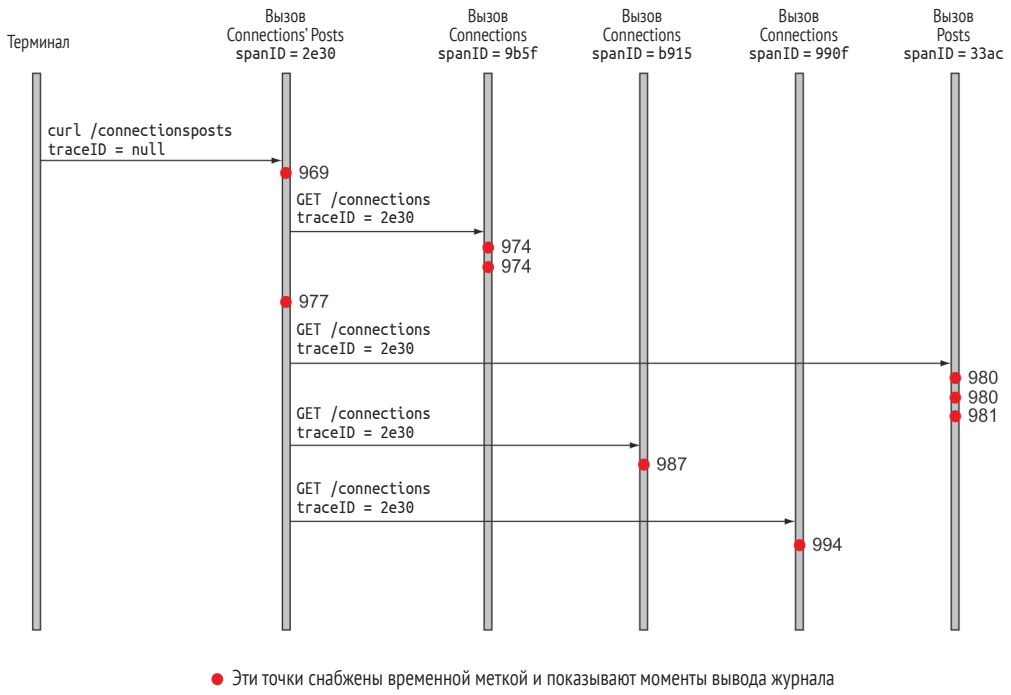


Рис. 11.9 ❖ Здесь вызовы служб накладываются на выходные данные журнала, показанные на предыдущем рисунке. Эта информация в настоящее время не отображается в ваших журналах

Обратите внимание, что данные, обозначенные стрелками на рис. 11.9, также можно вставить в вывод журнала, но это не сделано с библиотеками Spring Cloud.

11.3.2. Компоновка трассировок с помощью Zipkin

Я попросила вас изучить, что происходит с идентификаторами трассировки и отдельного запроса, просматривая журналы, но такие инструменты, как Zipkin, позволяют более эффективно анализировать эти типы значений. Zipkin предоставляет хранилище данных для метрик с аннотациями трассировок и отдельных запросов, а также пользовательский интерфейс, который отображает данные и поддерживает навигацию по этим данным.

Службы несут ответственность за доставку данных в хранилище Zipkin, что заставляет нас задуматься. Отправка данных из службы требует ресурсов; она потребляет память, циклы центрального процессора и пропускную способность подсистемы ввода/вывода. Метрики, о которых я говорила ранее, были ограничены в обслуживании. Вы собирали данные о том, как работает запущенная служба. Метрики, которые я рассматриваю сейчас, относятся к вызову службы. В первом случае вы можете собирать метрики раз в секунду, но если служба реагирует на 100 запросов в секунду, и вы собираете метрики для каждого вызова, что требует ресурсов на два порядка больше. В результате лучшим способом для распределенной трассировки является сбор метрик только для подмножества всех сервисных запросов.

Я рассмотрела здесь этот нюанс, потому что хочу, чтобы вы использовали распределенную трассировку в экспериментах, которые мы провели в главах 9 и 10. Мы нагрузим наше приложение, но вам нужно ограничить влияние, которое оказывает трассировка на нашу систему, поэтому вам нужно настроить ее таким образом, чтобы она генерировала сведения о трассировке только для подмножества вызовов. В развертывании каждого из наших сервисов вы увидите такую конфигурацию:

```
- name: SPRING_SLEUTH_SAMPLER_PERCENTAGE
  value: “0.01”
```

Это приведет к тому, что 1 % запросов сгенерирует метрики трассировки и отправит в Zipkin (сейчас я расскажу о Spring Cloud Sleuth). Теперь давайте нагрузим систему. Я изменила объем запросов в нашей симуляции, поэтому мне нужно, чтобы вы загрузили новую конфигурацию JMeter в Kubernetes:

```
kubectl create configmap zipkin-jmeter-config \
 --from-file=jmeter_run.jmx=loadTesting/ConnectionsPostsLoadZipkin.jmx
```

Затем вы можете запустить симуляцию:

```
kubectl create -f loadTesting/jmeter-deployment.yaml
```

Мы снова наблюдаем доступ к службе Connections’ Posts, а что касается подмножества этих запросов, данные трассировки хранятся в базе данных Zipkin. Чтобы получить доступ к URL-адресу пользовательского интерфейса Zipkin, найдите IP-адрес и порт службы Zipkin с помощью этой команды:

```
echo http://\
$(kubectl get service zipkin-svc \
-o=jsonpath={.status.loadBalancer.ingress[0].ip})”/”\
$(kubectl get service zipkin-svc \
-o=jsonpath={.spec.ports[0].port})
```

Откройте этот адрес в своем браузере и нажмите кнопку **Find Traces**. После этого будут показаны результаты, как на рис. 11.10.

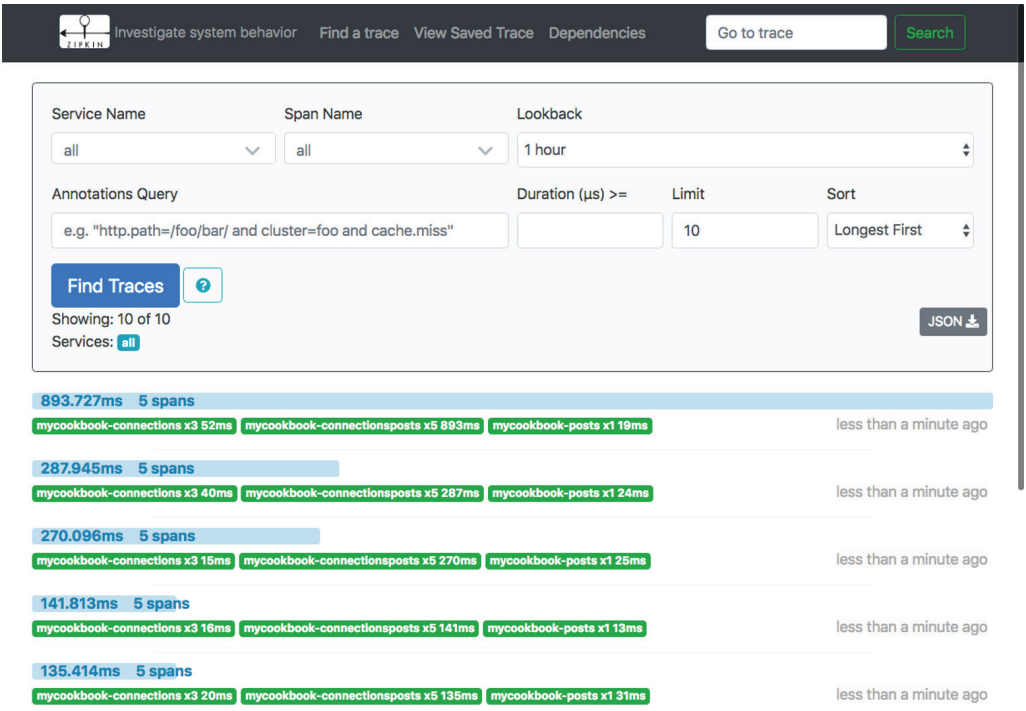


Рис. 11.10 ❖ Zipkin предоставляет пользовательский интерфейс, который позволяет осуществлять поиск по данным, хранящимся в базе данных распределенных метрик, и берет записи, которые связаны общим идентификатором трассировки. На этом рисунке показано пять таких трассировок, каждая из которых объединяет пять отдельных запросов, представленных в виде спанов

Здесь показано пять трассировок. Каждый из этих результатов соответствует отдельной команде `curl`, которую мы использовали для службы Connections' Posts. На первую ушло почти 900 мс; на вторую и третью менее 300 мс; и на две последние – менее 150 мс. При нажатии на светло-серую полоску рядом с четвертым отображаемым вызовом, где написано «141,813ms 5 spans», отобразится экран, как показано на рис. 11.11.

На этом экране видны спаны, образующие единый вызов, причем отдельные вызовы были собраны вместе, поскольку у них один и тот же идентификатор трассировки. Вспомните вывод журнала, который вы просмотрели всего минуту назад и соединили все воедино на рис. 11.8; на самом деле рис. 11.11 аналогичен перевернутому набок рис. 11.8. Видно, что отдельный вызов для службы Connections' Posts растягивается на полные 141 мс, а также видны спаны нисходящих запросов – запрос к службе Connections для получения списка пользователей, на которых я подписана, запрос к службе Posts для получения списка постов и два

запроса к службе Connections, чтобы получить имена авторов постов. Это именно тот поток, который был получен из более раннего вывода журнала.

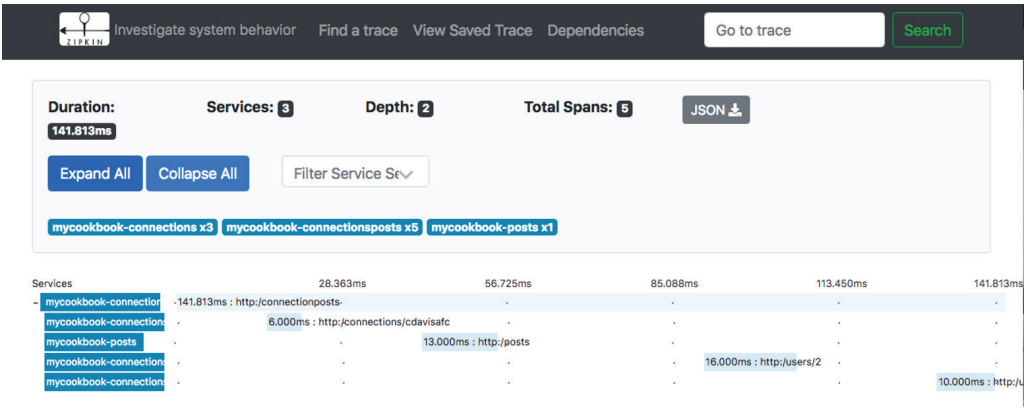


Рис. 11.11 ❖ Окно Zipkin, где показаны детали одного запроса к службе Connections' Posts. Этот запрос привел к четырем нисходящим запросам: один к службе Connections, один к службе Posts, а затем еще два запроса к службе Connections. Также показана задержка для каждого запроса, представленная в виде спана

Теперь давайте прервем работу нашей системы, вызвав сбой сети, как мы это делали в главах 9 и 10. Я дам вам скрипт, с помощью которого мы разорвем сетевые соединения между экземплярами нашего сервиса Posts и базой данных MySQL. Вам нужно будет обновить его, чтобы он указывал на ваш модуль MySQL и включал в себя IP-адреса экземпляров вашей службы Posts. Затем вы можете вызвать этот скрипт с помощью команды

```
./loadTesting/alternetwork-db.sh add
```

Оставьте сеть в таком состоянии на 10–15 секунд, а затем восстановите ее работу, выполнив эту команду:

```
./loadTesting/alternetwork-db.sh delete
```

Теперь вернитесь к главной панели инструментов Zipkin и нажмите кнопку **Find Traces**. Вы увидите нечто похожее на то, что изображено на рис. 11.12.

Полоски, обозначающие всю длину определенного вызова (те, где написано: 26.488s 22 spans), стали светло-красными, указывая на первые признаки проблемы. Рассматривая детали, вы видите, что каждый вызов имеет более пяти спанов, которые вы видели, когда система была в работоспособном состоянии. Нажмите на одну из этих красных полосок – и увидите детали, показанные на рис. 11.13.

Здесь вы видите повторные отправки запросов! Из-за разрыва сети между службой Posts и ее базой данных служба не может сгенерировать ответ, и время ожидания запроса из кода службы Connections' Posts истекло. Теперь вы видите, что версия кода, которую я взяла из главы 9 и поместила в этот проект, – это та версия, которая применяет грубую силу или, если очень жестко, повторную отправку запроса. После того как сетевое соединение восстановлено, вызов службы Posts успешно выполняется, и выполнение службы Connections' Posts завершается.

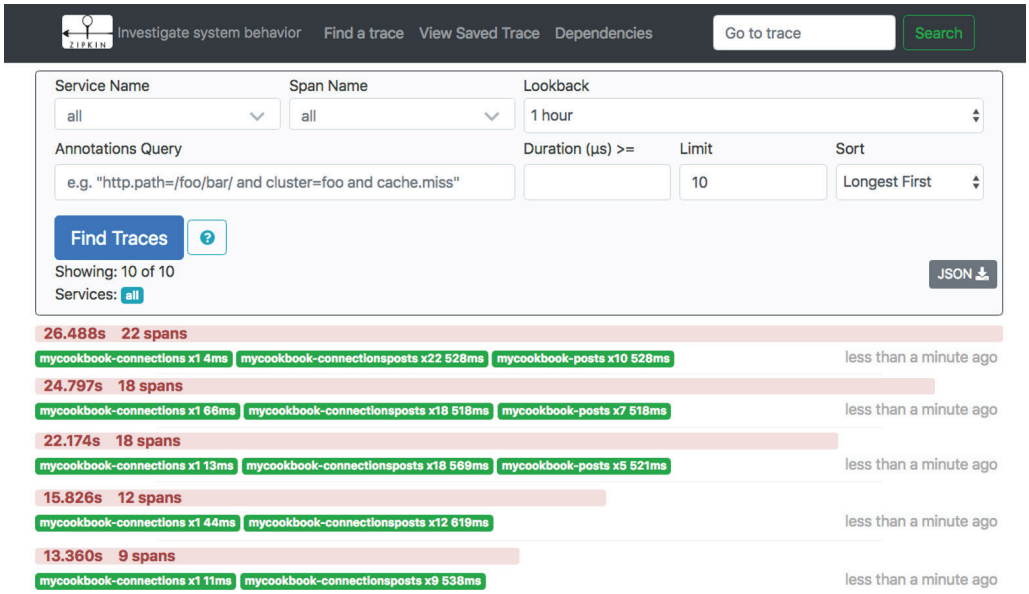


Рис. 11.12 ❖ В то время как работа сети была нарушена, запросы к службе Connections' Posts привели к сбою нисходящих запросов. Например, вы видите, что то, для чего обычно хватало четырех нисходящих запросов, привело к гораздо большему количеству запросов/спанов. Полоски, отображающие время и количество спанов (например, «26,488с 22 spans»), теперь также окрашены в красный цвет.

Это указывает на то, что некоторые из этих запросов возвращали ошибки

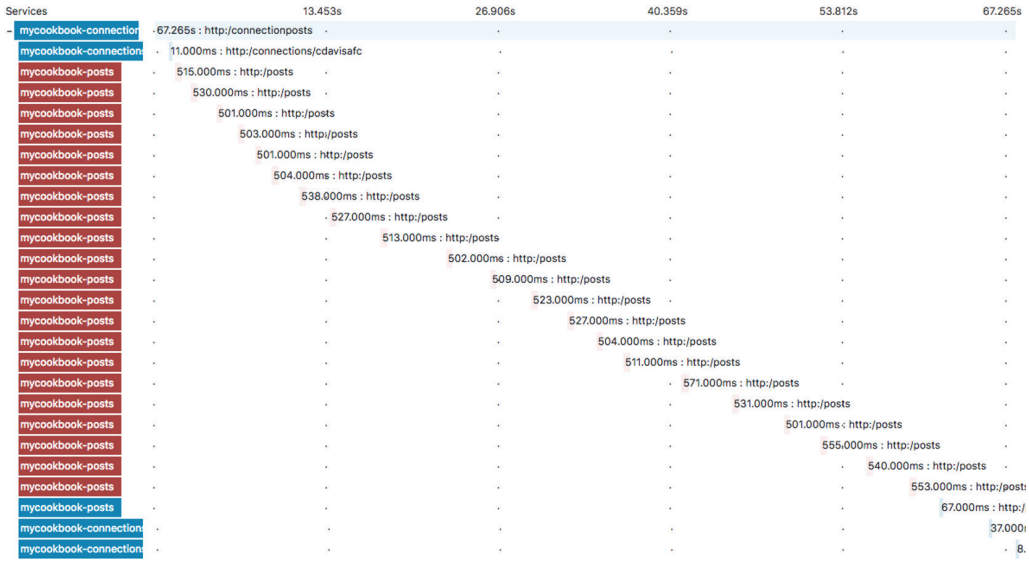


Рис. 11.13 ❖ В деталях одной из трассировок запроса показаны повторные неудачные вызовы к службе Posts, каждый из которых занимает примерно 500 мс. Это значение тайм-аута запроса для HTTP-вызовов, выполняемых службой Connections' Posts

Наконец, вернувшись на домашнюю страницу Zipkin и просмотрев список трассировок вскоре после восстановления работы сети, вы увидите данные, показанные на рис. 11.14.

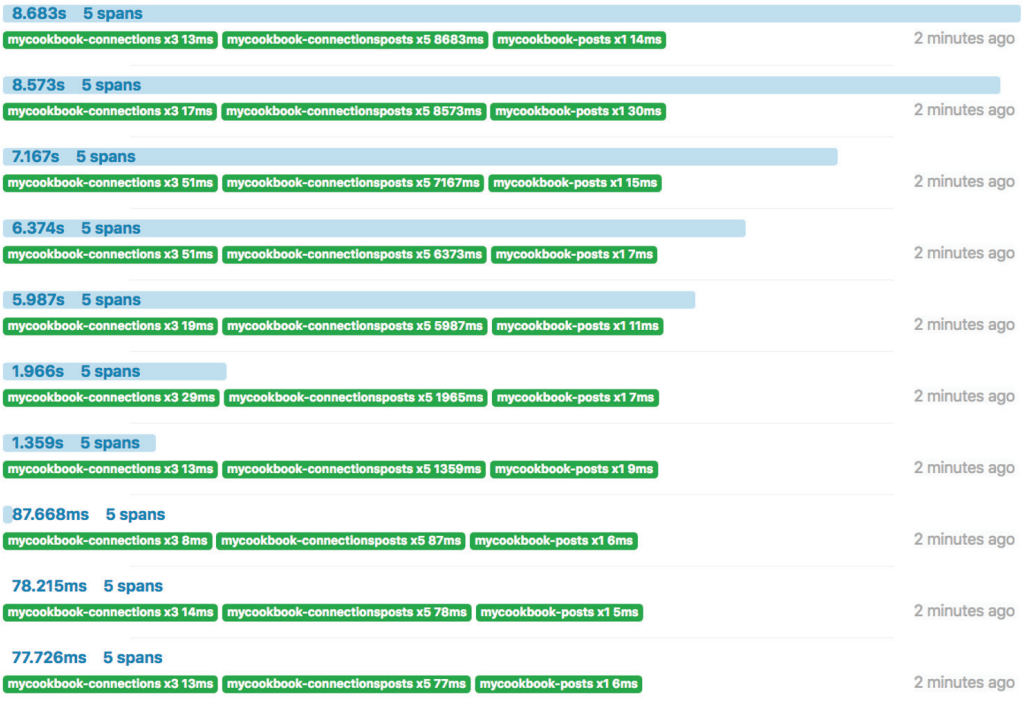


Рис. 11.14 ❖ Когда работа сети восстанавливается, нисходящие запросы снова выполняются успешно, но, судя по времени, потраченному на выполнение запросов Connections' Posts, можно увидеть запас по интенсивности трафика и сколько потребовалось времени для его рассеивания

Это показывает, что время, необходимое для обработки Connections' Posts, возвращается к нормальному состоянию после того, как трафик после шквала повторных отправок запросов рассеивается.

Используя методы трассировки распределенных приложений, вы получили ценное представление о том, как работает наше приложение для облачной среды. Вы смогли быстро увидеть, когда и где возникли ошибки, и смогли отследить маршрут до момента стабильности после устранения более раннего сбоя. А хорошая новость заключается в том, что при использовании какого-нибудь фреймворка вроде Spring Framework эти возможности легко добавить в реализацию.

11.3.3. Детали реализации

Напомню, что в основе трассировки распределенных приложений лежат два конкретных метода:

- вставка идентификаторов трассировки;

- плоскость управления, которая собирает метрики, включающие в себя эти идентификаторы, и использует их, чтобы соединить воедино вызовы связанных служб.

Эти две проблемы решаются с помощью включения двух зависимостей в файлы вашего проекта.

Листинг 11.4 ❖ Добавляется в файл pom.xml каждого из трех микросервисов

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
    <version>2.0.3.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
    <version>2.0.3.RELEASE</version>
</dependency>
```

Spring Cloud Sleuth обеспечивает генерацию и распространение идентификаторов трасс и отдельных вызовов. Включение в состав первой из предыдущих зависимостей приводит к тому, что эти значения включаются в файлы журнала, которые вы изучали ранее. Вторая зависимость добавляет доставку метрик на сервер Zipkin, адрес которого настроен в каждый из ваших сервисов, установив свойство `spring.zipkin.baseUrl`. Можно увидеть этот параметр наряду с частотой дискретизации в файлах развертывания Kubernetes каждой из служб. (Обратите внимание, что вы обращаетесь к службе Zipkin по имени; протокол обнаружения сервисов, встроенный в Kubernetes, помогает с фактической привязкой.)

Листинг 11.5 ❖ Добавляется в файл развертывания для каждого из трех микросервисов

```
- name: SPRING_APPLICATION_JSON
  value: '{"spring":{"zipkin":{"baseUrl":"http://zipkin-svc:9411/"}}}'
- name: SPRING_SLEUTH_SAMPLER_PERCENTAGE
  value: "0.01"
```

Я включила службу Zipkin в развертывание приложения, используемого в качестве примера, вместе с файлом zipkindeployment.yaml.

Вот и все. Верно – вам не нужно ничего менять в коде, чтобы активировать трассировку. Этим полностью занимается Spring Framework. Результат, который приносит данный тип трассировки, стоит такого уровня усилий и даже немного больше, если вы программируете на языке, который не обеспечивает такой же уровень поддержки. На момент написания этих строк существовали библиотеки Zipkin для Java, JavaScript, C #, Golang, Ruby, Scala, PHP, Python и других языков. Эта широко принятая технология также включает в себя и другие продукты, например Istio (см. раздел 10.3.2 в главе 10).

Резюме

- И метрики, и записи журнала должны быть предварительно извлечены из контекста времени выполнения, в котором выполняются наши сервисы, поскольку эти среды выполнения часто оказываются недоступны после того, как

служба столкнулась с проблемой или была обновлена. Среда исполнения наших сервисов должна рассматриваться как эфемерная.

- Агрегирование записей журнала из нескольких экземпляров службы важно для наблюдаемости. Решение, которое чередует во временном порядке записи из разных служб, обычно является предпочтительным.
- Сбор информации о наблюдаемости, журналов, метрик и данных трассировки эффективно реализуется в сайдкар-прокси, что позволяет приложению сосредоточиться на бизнес-логике и сконцентрировать эксплуатационные потребности в сервисной сетке.
- Хорошо зарекомендовавшие себя методы трассировки распределенных приложений и их реализации предоставляют ценную информацию о работоспособности и производительности ваших распределенных приложений.
- Многие шаблоны, описанные ранее в этой книге, используются в решениях, которые обеспечивают необходимую наблюдаемость. Конфигурация приложения, жизненный цикл приложения, обнаружение сервисов, шлюзы и сервисная сетка – все это задействовано.

Глава 12

Данные в облачной среде: разбиение монолитных данных

О чем идет речь в этой главе:

- почему каждому микросервису нужен кеш;
- использование событий для заполнения локальных хранилищ данных / кешей;
- использование обмена сообщениями в событийно-ориентированных системах;
- разница между обменом сообщениями и событиями;
- журнал событий и порождение событий.

Помните, какое определение я дала термину «cloud-native» в первой главе? Там я провела небольшой анализ, в ходе которого мы перешли от требований высокого уровня к современному программному обеспечению к набору из четырех характеристик: программное обеспечение для облачной среды является избыточным, адаптируемым, модульным и динамически масштабируемым (рис. 12.1). И на протяжении большей части книги вы изучали эти характеристики в контексте служб и взаимодействий, из которых состоит наше программное обеспечение. Но помните, что третьим объектом в ментальной модели, которую я также изложила в первой главе, были данные. Характеристики программного обеспечения для облачной среды в равной степени применимы и к уровню данных.

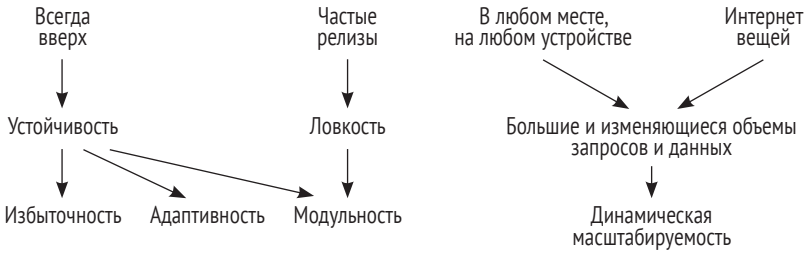


Рис. 12.1 ❖ Требования пользователей к программному обеспечению ведут разработку в направлении принципов архитектуры облачной среды. Что касается данных в облачной среде, мы обратим наше внимание на модульность и автономность

Взять, к примеру, избыточность. Хотя ценность наличия нескольких копий данных давно понятна, в прошлом часто использовались шаблоны, для того чтобы добиться этого, а иногда они возникали в результате эксплуатации. Например, в топологиях активного/пассивного развертывания активный узел обслуживает весь трафик чтения и записи, в то время как пассивный узел обновляется записью за кулисами. Если с активным узлом что-то случится, вся система может переключиться на пассивный узел. Современные сервисы данных для облачной среды (напомню, что в разделе 5.4.1 я говорила о специальных типах сервисов с сохранением состояния) обладают избыточностью, которая глубоко встроена в их многоузловые конструкции, и используют такие шаблоны, как Paxos и лидер/последователь для определения характеристик согласованности и доступности, которые они предоставляют.

Что касается масштабируемости, мы также увидели значительные изменения в шаблонах, которые правят бал. Традиционные базы данных чаще всего масштабировались вертикально, предоставляя все более крупные хосты и устройства хранения для удовлетворения растущих потребностей. Но, как вы видели в ходе прочтения, благодаря правилам горизонтальной масштабируемости для облачной среды большинство современных баз данных, таких как Cassandra, MongoDB и Couchbase, имеют эту модель, которая внедрена в ядро их систем. По мере увеличения объемов данных новые узлы можно присоединить к кластеру базы данных, и существующие данные и запросы будут перераспределены по всем старым и новым узлам.

Хотя мы могли бы достаточно долго и подробно изучать все четыре характеристики, изображенные в нижней части рис. 12.1, в последней главе я сосредоточу основное внимание на *модульности*. Очевидно, что объединение множества отдельных (микро)сервисов для формирования нашего программного обеспечения является центральным элементом архитектур облачной среды. Но слово *микросервис* немного вводит в заблуждение (поэтому я редко использовала его). Это побуждает нас уделять слишком много внимания размеру сервиса, вместо того чтобы обратить внимание на самое ценное, что он дает: автономность. Если все сделано правильно, микросервисы могут создаваться независимыми командами и могут управляться независимо (масштабироваться). Кроме того, они являются основным объектом, к которому применяется множество других шаблонов для облачной среды (предохранители, обнаружение сервисов), и они находятся на конечных точках взаимодействий.

Раньше у нас были монолитные приложения, которые мы теперь разложили на множество отдельных сервисов, что дало нам модульность. Так ли это? Эти монолитные приложения имели монолитные базы данных, и слишком часто мы встречаем новые конструкции для облачной среды, которые выглядят так, как показано на рис. 12.2. Мы разделили вычислительную часть нашего программного обеспечения на отдельные сервисы, но оставили централизованный монолитный уровень данных.

Эта простая диаграмма проясняет, что такие конструкции предлагают только иллюзию модульности; общая база данных создает переходные зависимости между независимыми службами. Например, если один сервис хочет изменить схему базы данных, он должен координировать действия со всеми другими службами, которые совместно используют какую-либо часть этой схемы базы данных.

Кроме того, благодаря общей базе данных многие отдельные службы конкурируют за одновременный доступ, создавая дополнительные проблемы. Увы, наши микроуслуги не так уж и автономны.

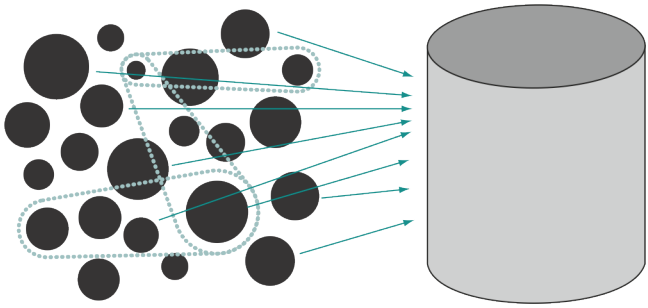


Рис. 12.2 ❖ Независимые микросервисы, которые совместно используют одну базу данных, не являются автономными

Как следует из названия главы, в конечном итоге наша цель состоит в том, чтобы разбить этот монолит данных. Возможно, вы уже слышали фразу, что каждый микросервис должен иметь свою собственную базу данных, но это может показаться опасным. Как вы будете поддерживать целостность своих данных в десятках или сотнях хранилищ? Как разные команды будут координировать действия в этой сложной сети? Точно так же, как разделение вычислительного монолита породило новые проблемы – проблемы, которые мы систематически решаем с помощью набора шаблонов, представленных в этой книге, – разрушение монолита данных тоже создает проблемы. И эти проблемы будут решаться с помощью шаблонов.

Многие в этой отрасли считают, что порождение событий является окончательным ответом на эти проблемы, и эта глава охватывает данную тему. Но как и со всеми другими шаблонами, описанными в этой книге, это не относится к категории «все или ничего». Я бы хотела отправиться с вами в путешествие, которое начинается с базовых, знакомых шаблонов проектирования, следуя по пути к порождению событий. Я надеюсь, что если буду освещать эту тему таким образом, это позволит вам не только лучше разбираться в ключевых элементах обработки данных в облачной среде, но и предоставит вам практический способ достичь этого поэтапно.

Я начну с кеширования – метода, который использовалася в течение некоторого времени и остается актуальным в архитектурах программного обеспечения для облачной среды; помните, что произошло, когда мы добавили кеш как часть возврата к предыдущей рабочей версии при изучении шаблонов устойчивости? Здесь мы еще раз посмотрим на кеширование. Затем я сделаю краткий обзор главы 4, где мы перевернули протокол «запрос/ответ» с ног на голову и вместо этого отправляли поступающие события через топологию своих сервисов. Я улучшу этот событийно-ориентированный дизайн, добавив журнал событий, а затем, в конце, познакомлю вас с концепцией порождения событий. По мере прохождения этих преобразований я буду использовать анализ успешных и неудачных запросов при различных условиях сбоя в качестве средства автономности, и вы увидите важную

роль, которую играет проектирование структуры данных в наших архитектурах облачной среды.

12.1. Каждому микросервису нужен кеш

Чтобы увидеть эффект, который дает кеширование, – а он выходит за рамки обычной производительности, – давайте начнем с варианта, где кеш еще не используется. На рис. 12.3 показан наш обычный пример, в котором служба Connections' Posts агрегирует контент из двух других наших служб: Connections, которая управляет пользователями и теми, на кого они подписаны, и Posts, которая управляет постами в блоге. На этой диаграмме мы используем знакомый протокол «запрос/ответ» для взаимодействия между этими службами.

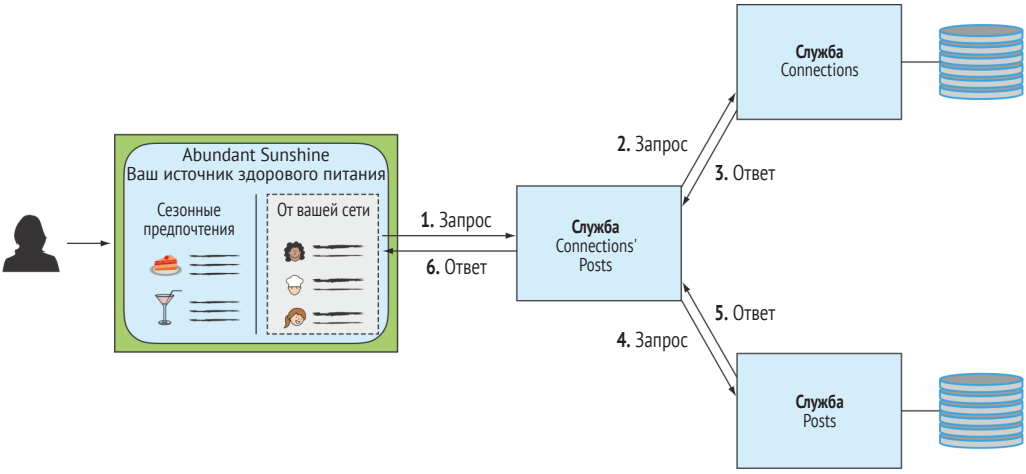


Рис. 12.3 ❖ Простая топология с агрегирующим сервисом. Обмен данными с зависимыми сервисами происходит через протокол «запрос/ответ»

Используя эту простую архитектуру в качестве основы для анализа, который идет в оставшейся части этой главы, давайте посмотрим на отказоустойчивость системы перед лицом частичных отключений. На рис. 12.4 показано четыре вычислительных компонента нашего программного обеспечения: многофункциональное веб-приложение и три сервиса, обозначенных горизонтальными линиями. Когда эти линии идут как одно целое, они обозначают время, когда служба доступна и дает результаты, а разрывы в этих линиях указывают на то, когда служба недоступна или работает неправильно.

Вы заметите, что для многофункционального веб-приложения или службы Connections' Posts я не указала время простоя. Это не значит, что они никогда не зависают. Скорее, я хочу сосредоточиться только на зависимостях между микросервисами и протоколом «запрос /ответ»; эти взаимодействия существуют только тогда, когда агрегирующий сервис вызывается и функционирует.

По вертикали поступают запросы от веб-приложения к службе Connections' Posts, которые затем потоком идут к зависимым службам. Когда служба Connections или Posts не работает, агрегирующий сервис не может сгенерировать результат.

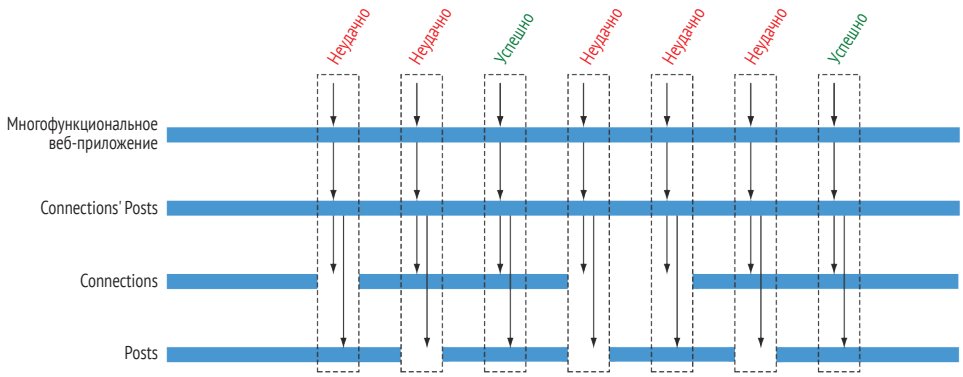


Рис. 12.4 ❖ Если одна из зависимых служб недоступна или не работает должным образом, служба Connections' Posts также не сможет сгенерировать результат

Теперь давайте добавим кеш в службу Connections' Posts (рис. 12.5). Этот кеш можно наполнить различными способами. При использовании *кеширования с параллельной выборкой* клиент (в данном случае служба Connections'Post) отвечает за реализацию протокола кеширования: когда необходимы данные, он проверяет значение в кеше и, если его нет, отправляет запрос в нисходящий сервис и записывает результат в кеш, прежде чем сам вернет результат. При *сквозном кешировании* служба Connections' Posts обращается только к кешу, и кеш реализует логику получения значения из нисходящего сервиса, когда это необходимо. Независимо от протокола значение сохраняется локально после успешного нисходящего запроса. Давайте теперь посмотрим, стала ли наша система более устойчивой после этого. На рис. 12.6 рядом со службой Connections' Posts добавлена еще одна горизонтальная полоса. Чтобы было попроще, предположим, что кеш доступен всегда, когда доступна служба Connections' Posts. Видно, что первоначально результаты запроса в точности совпадают с тем, что мы видели ранее: когда ни одна из нисходящих служб недоступна, агрегирующий сервис не сможет сгенерировать полный ответ. Но после того, как кеш наполнен, он изолирует потребителя, веб-приложение, от сбоев нисходящих служб.

Я хочу обратить ваше внимание на одну особенность на рис. 12.6. По сравнению со сценарием, изображенным на рис. 12.4, вы видите, что входящие запросы становятся успешными, как только каждая нисходящая служба будет доступна хотя бы один раз, и эти успешные запросы не обязательно приходят одновременно. Должно быть ясно, что уровень устойчивости напрямую связан со степенью автономности компонентов нашей системы.

Все это довольно убедительно, и, очевидно, вы достигли гораздо более высокого уровня устойчивости, когда сделали службу Connections' Posts более автономной, добавив кеш. Почему этого недостаточно? Дело в том, что все, что я здесь показала, – сильно упрощенное кеширование. Намек на это можно увидеть на рис. 12.6, где я указываю на то, что второй запрос *мог* быть успешным, *если* бы первый запрос загрузил данные, необходимые для второго. Откуда, например, вы знаете, что отсутствие каких-либо кешированных постов от пользователя Food52 означает, что на этом сайте нет новых, или что успешный запрос Posts, включая этот сайт, еще не получен? А если в кеше есть запись, как узнать, актуальна ли она?

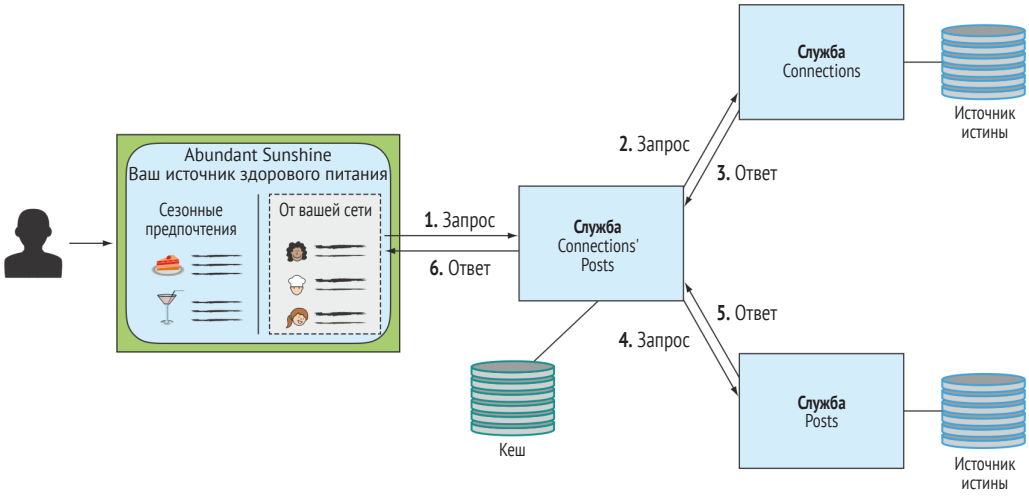


Рис. 12.5 ❖ В кеше, добавленном в службу Connections' Posts, будут храниться ответы, полученные в ходе успешных запросов к зависимым службам. Источник истины данных остается в базах данных, привязанных к каждой из нисходящих служб; локальное хранилище службы Connections' Posts не является официальной копией

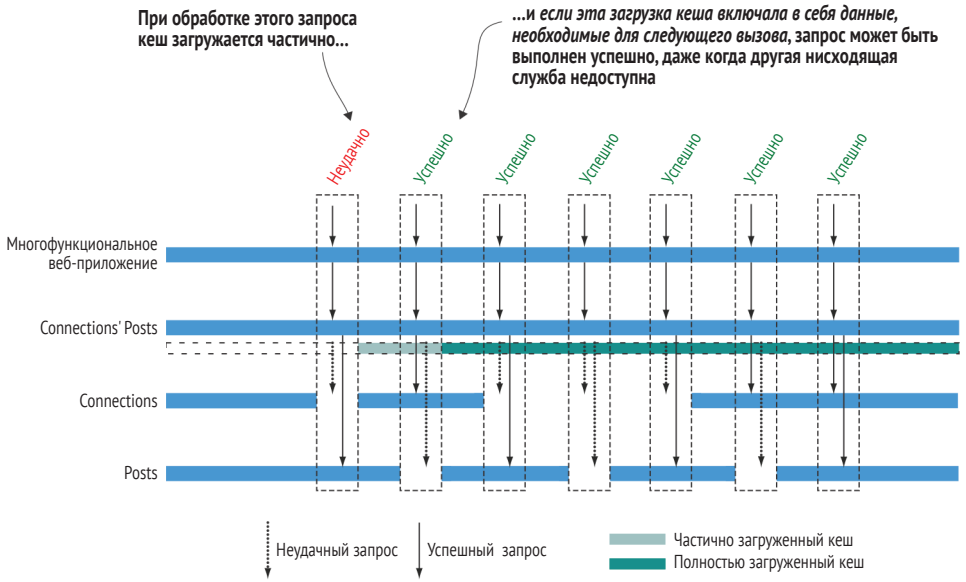


Рис. 12.6 ❖ Добавляя кеш в Connections' Posts, вы повышаете степень автономности службы и в результате получаете больше устойчивости

То, как мы традиционно использовали кеширование и даже мотивирующие факторы его применения, отличалось от наших нынешних целей его использования в микросервисных архитектурах. Часто используемые для достижения

определенного уровня прироста производительности данные, которые мы кешировали, как правило, были более статичными (например, изображения на вебсайтах и сопоставление почтовых индексов с городами), поэтому истечения срока действия кеша на базе таймера обычно было достаточно. Неудачное обращение к кеш-памяти было постоянным показателем того, что он еще не загружен, и часто при запуске использующего его приложения запускались какие-то процессы для разогрева или предварительной загрузки кеша. Попытка использовать кеш в этих сценариях, ориентированных на микросервисы, требует переосмысления шаблонов, а спереди и в центре – проблема свежести кеша. В идеале, когда в нисходящем сервисе что-то меняется, мы бы хотели, чтобы эти изменения отражались в нашем локальном хранилище данных как можно скорее.

Ага! Понимаете, к чему я веду?

12.2. Переход от протокола «запрос/ответ» к событийно-ориентированному подходу

Как я и предлагала, мы начали наше путешествие на пути к более подходящему способу обработки локального хранилища в главе 4, когда перевернули протокол «запрос/ответ» с ног на голову, перейдя к протоколу взаимодействия, управляемого событиями. Рисунок 12.7 выглядит аналогично диаграммам, которые вы видели ранее в этой главе, но с одним важным отличием: взаимодействие между Connections' Post и относящимися к ней службами инициируется с противоположной стороны. У Connections' Posts есть свое локальное хранилище, напоминающее кеш из предыдущего примера, но теперь оно обновляется каждый раз, когда одна из нисходящих служб транслирует изменение.

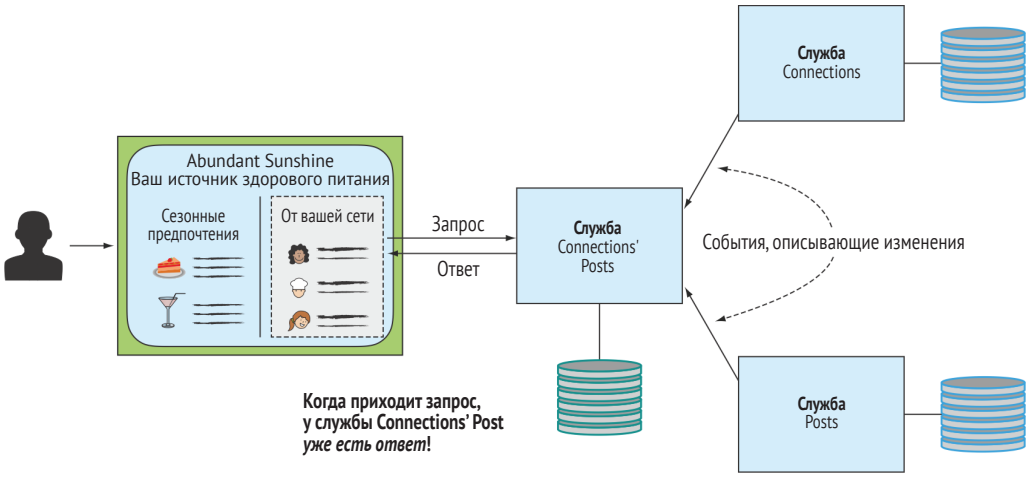


Рис. 12.7 ❖ Когда в нисходящих сервисах происходят изменения, события, описывающие эти изменения, отправляются заинтересованным сторонам, где их локальное хранилище обновляется

Больше не нужно беспокоиться об истечении срока действия кеша. Не нужно думать о том, действительно ли отсутствие данных означает, что таких данных

не существует. Предполагая, что механизм доставки нисходящих событий функционирует (и вы вскоре увидите, как мы обеспечиваем это), служба Connections' Posts может работать в своем собственном ограниченном контексте, не беспокоясь о том, что происходит в других местах системы. Вот красота!

Давайте посмотрим, как это влияет на устойчивость нашей системы. На рис. 12.8 приводится обновленный вариант наших предыдущих диаграмм. Теперь вы видите, что события, поступающие от служб Connections и Posts, отправляются в Connections' Posts, а когда поступает запрос с веб-страницы, не важно, работают зависимые службы или нет; у Connections' Posts есть свое локальное хранилище, и она работает автономно. Но что произойдет, если она будет недоступна, когда у одной из остальных служб есть событие для доставки? Согласно реализации, приведенной в главе 4, это событие будет потеряно. Конечно, мы можем использовать другие шаблоны, которые изучали, такие как повторная отправка запроса, чтобы компенсировать часть сбоев, но в некоторых случаях это не сработает. Конечный результат будет плохим. Службе Connections' Posts кажется, что все в порядке. Она вернет результат, основанный на данных в своем локальном хранилище, именно так, как она и должна сделать. Она и понятия не имеет, что эти данные не точные. А дальше – хуже. Обычно это не тот случай, когда событие представляет интерес только для одной стороны; наоборот, многим другим объектам есть дело до того, например если пользователь в системе меняет свое имя. Если это событие не доходит до множества предполагаемых участников, несоответствия могут распространяться по всей вашей системе, как показано на рис. 12.9.

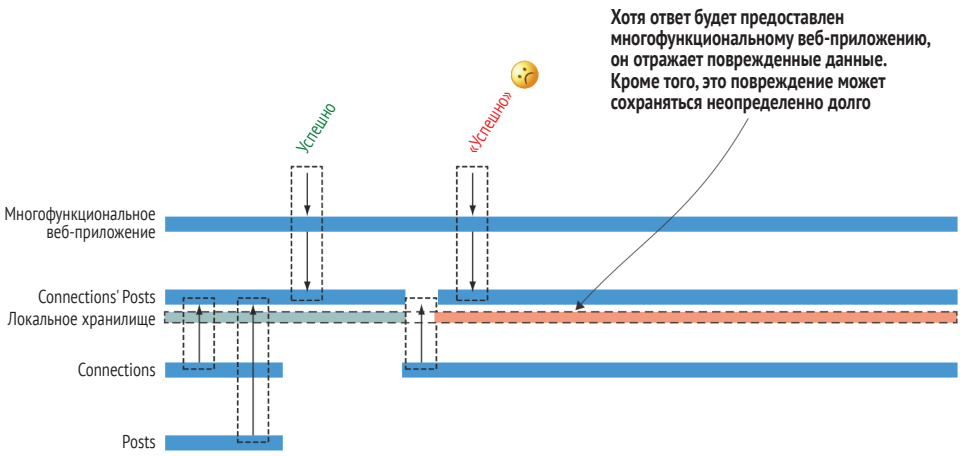


Рис. 12.8 ❖ Событийно-ориентированный подход позволяет службе Connections' Posts работать абсолютно автономно, используя данные в локальном хранилище. Однако пропущенные события могут повредить хранилищу, «успешно» возвращая неверные результаты

По сути, местные хранилища, возможно многие из них, сейчас испорчены. Хуже того, они могут оставаться такими до бесконечности! Событийно-ориентированный подход обещал устранить вопрос свежести кеша, но потерпел неудачу.

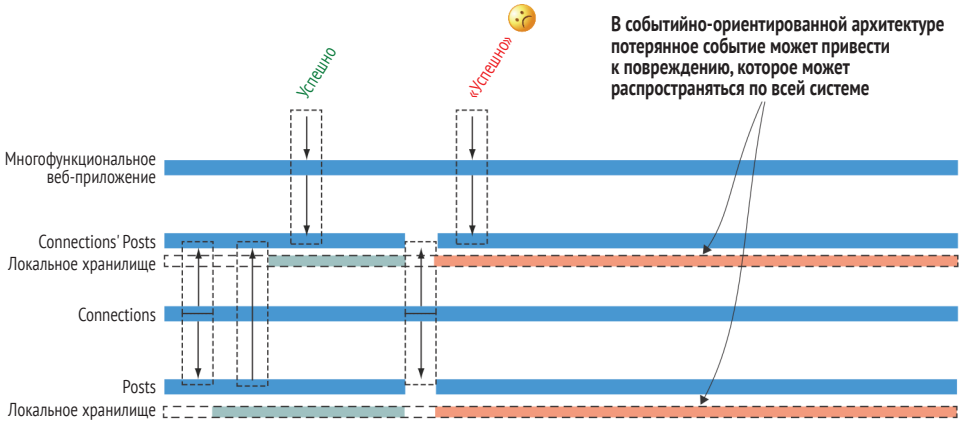


Рис. 12.9 ❖ События обычно представляют интерес для многих сторон, поэтому если они не доставляются должным образом, это может привести к распространению несоответствий по всей системе

Помните тот комментарий относительно механизма доставки нисходящих событий? Нет? Тогда продолжим.

12.3. Журнал событий

Нужно исключить необходимость в доступности службы Connections' Posts при отправке события, о котором она печется. Да, вы угадали – вы будете использовать систему асинхронного обмена сообщениями. Служба Connections, вместо того чтобы отправлять события непосредственно в Connections' Posts, будет доставлять эти события в систему, ответственную за их доставку. Конечно, ваше программное обеспечение будет зависеть от доступности этой системы обмена сообщениями, но сосредоточие семантики обмена сообщениями в системе, специально разработанной для этой цели, не только обеспечивает последовательную реализацию шаблонов, но и позволяет сосредоточить усилия по обеспечению устойчивости в структуре обмена сообщениями, а не на обширной сети взаимосвязанных служб. Если система обмена сообщениями сама по себе спроектирована облачным способом с избыточностью и динамическим масштабированием, она будет надежной. На рис. 12.10 показан журнал событий, добавленный в вашу программную архитектуру.

Давайте посмотрим, как это влияет на устойчивость программного обеспечения в целом. На рис. 12.11 добавлен еще один компонент: журнал событий. Поскольку службы Connections and Posts генерируют события, они отправляются в журнал, и заинтересованные в событии стороны будут забирать их и обрабатывать. Обратите внимание, что первое сообщение, отправленное службой Connections, забрали службы Posts и Connections' Post. Однако второе событие, генерируемое службой Posts, идет только в Connections' Posts. А событие, которое возникает, когда потребители не могут дать мгновенный ответ, сохраняется в журнале событий (по крайней мере)[1], пока все заинтересованные стороны не примут его, даже если

[1] Оставайтесь с нами – я вернусь к фразе «по крайней мере», когда буду говорить о порождении событий.

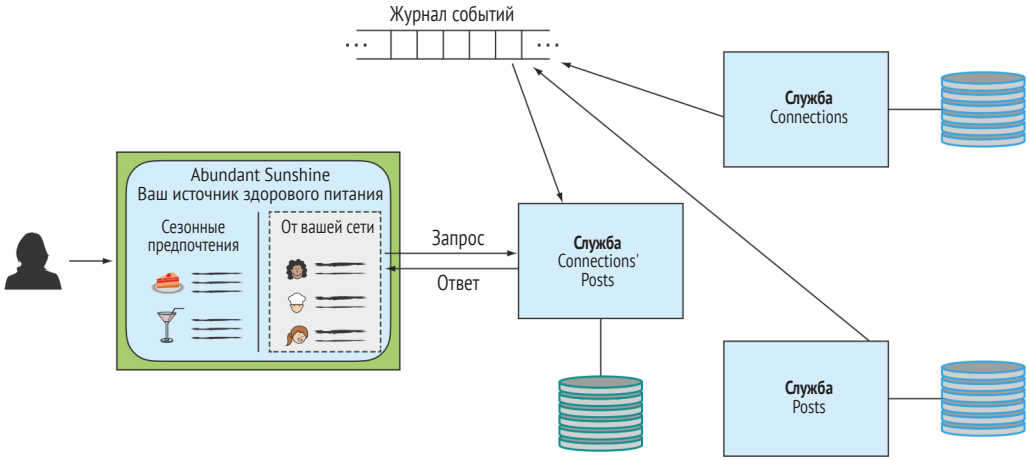


Рис. 12.10 ❖ Добавив журнал событий, вы теперь полностью отделили каждую из трех служб друг от друга. События производятся и потребляются из журнала событий, который отвечает за сохранение событий до тех пор, пока они больше не понадобятся

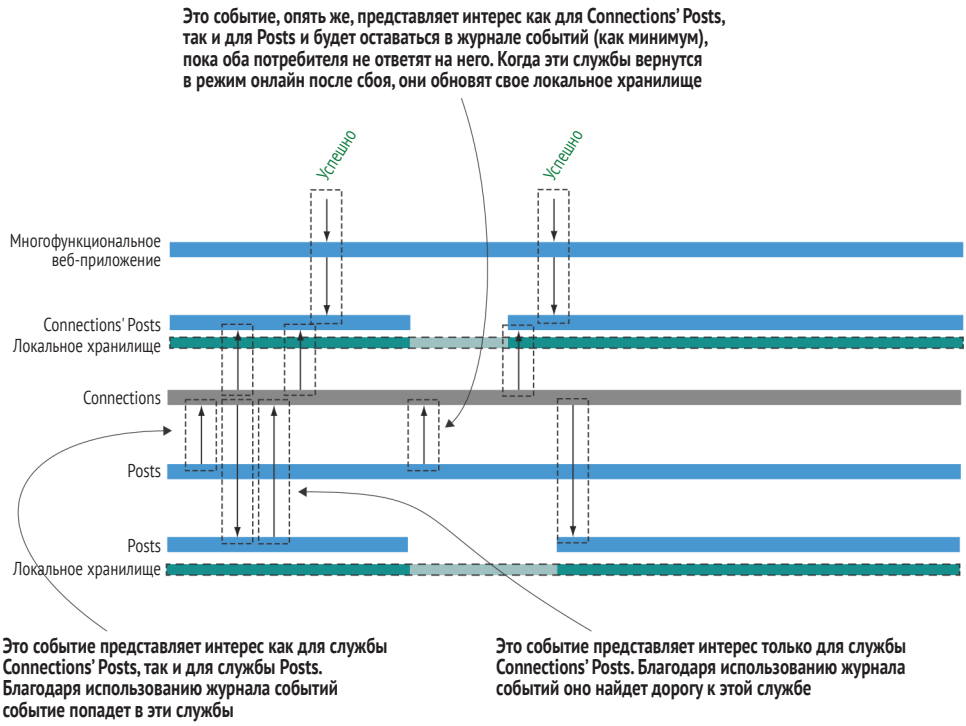


Рис. 12.11 ❖ События создаются и потребляются из журнала событий, который поддерживает эти события, пока они необходимы. Используя журнал событий, вы полностью отделяете сервисы друг от друга

эти потребители подключаются к сети в совершенно разное время. В результате мы видим, что доступность агрегирующего сервиса является сильной, даже когда происходят сбои во всей системе.

Одно замечание по поводу согласованности: ранее вы видели случай, когда из-за пропущенного события ваш кеш мог испортиться. Возможно, даже здесь, в период между производством события и его потреблением, локальное хранилище потребителя устарело. Когда я буду говорить о порождении событий далее в этой главе, мы рассмотрим эту проблему.

Подключение к структуре обмена сообщениями обычно происходит через клиентскую библиотеку

Обратите внимание, что создание и потребление сообщений из кода обычно выполняется с помощью какой-то клиентской библиотеки, которая часто реализует некоторые из шаблонов устойчивости, которые вы изучали. Например, если журнал событий не доступен. С помощью первого запроса клиентская библиотека может повторить попытку. А если установить соединение по-прежнему не удается, код библиотеки может вызвать протокол обнаружения сервисов, чтобы найти альтернативные конечные точки для контакта.

Разработчикам не нужно беспокоиться о деталях протокола. Вместо этого им нужно только указать тип уровня обслуживания, который им нужен от взаимодействия. Например, если реализация требует гарантированной доставки событий (программное обеспечение должно сообщить об ошибке, если сообщение нельзя доставить), это будет указано через API клиента. С другой стороны, если реализация может допустить потерю события, это тоже можно указать.

Давно пора взглянуть на какой-нибудь код, вам так не кажется?

12.3.1. Давайте посмотрим, как это работает: реализация событийно-ориентированных микросервисов

Реализация, приведенная в этой главе, основана на главе 4, где я поставила под сомнение нашу естественную склонность к подходу «запрос/ответ», предложив вместо этого событийно-ориентированное решение. Но тогда у вас еще была тесная связь между сервисами, как описано в разделе 12.2, когда клиент отправлял события непосредственно потребителям. Эта реализация показана на рис. 12.7.

Здесь я добавляю журнал событий, тем самым ослабляя связи между сервисами. Я в основном двигаюсь в направлении описания, приведенного на рис. 12.10, но когда мы перейдем к детальному рассмотрению, пример будет несколько сложнее. Служба Connections' Post – не единственная, кто интересуется сгенерированными событиями. Служба Posts также проявляет к ним интерес. Эта несколько более сложная топология позволяет нам исследовать детали более всеобъемлющим образом.

Служба Posts отвечает за управление постами в блоге, позволяя создавать новые посты и публиковать события, когда эти новые посты появляются. Посты, которые она хранит, включают в себя заголовок, текст и дату публикации, а также идентификатор пользователя, который опубликовал пост. В нашей системе

служба Connections отвечает за управление пользователями, а также связями между ними. Например, чтобы пользователь мог изменить свои данные, имя пользователя или собственное имя, запросы в вашем коде будут ссылаться на него только по идентификатору. Но идентификатор пользователя – это деталь реализации, внутренний идентификатор, который не должен утекать через какой-либо API. Поэтому, используя API, вы обращаетесь к пользователям через их имя (username). Например, при добавлении нового поста вы отправляете запрос с помощью метода POST в службу Posts с полезной нагрузкой, которая выглядит примерно так:

```
{
  “username”:”madmax”,
  “title”:”I love pho”,
  “body”:”Yesterday I made my mom a beef pho that was very close to what I
  ➥ ate in Vietnam earlier this year ...”
}
```

Пользователь с именем `madmax` опубликовал пост о супе фо, который он недавно приготовил. Однако когда вы сохраняете его в базе данных, вам не нужно сохранять имя пользователя, потому что если Макс позже изменит его, у вас возникнут проблемы с поиском его более ранних постов. Таким образом, вы сохраняете пост с идентификатором этого пользователя, как показано в приведенном ниже выводе SQL-запроса:

```
mysql> select * from cookbookposts.post;
+----+----------------------+---------------------+-------------+---------+
| id | body                 | date                | title       | user_id |
+----+----------------------+---------------------+-------------+---------+
|  1 | Yesterday I made...  | 2018-10-30 11:56:05 | I love pho  |       2 |
...
```

Из-за этого уровня косвенности службе Posts необходимо сопоставить идентификаторы пользователей с их именами.

Служба Posts не управляет пользователями (это делает служба Connections), но для того, чтобы оставаться отделенной от службы Connections, она должна хранить в своей базе данных актуальную копию некоторых данных, которыми управляет служба Connections, а именно взаимосвязанность идентификатора с именем пользователя. Вы видите вот такую таблицу, полученную с помощью приведенного ниже SQL-запроса:

```
mysql> select * from cookbookposts.user;
+----+-----------+
| id | username  |
+----+-----------+
|  1 | cdavisafc |
|  2 | madmax    |
|  3 | gmaxdavis |
+----+-----------+
```

Итак, на рис. 12.12 службы Connections и Posts генерируют события, а службы Posts и Connections' Posts потребляют их. На этой диаграмме также видны хранилища данных, связанные с каждой из ваших служб. Службы Connections и Posts владеют данными, которые хранятся в источнике истины, а службы Posts и Connections' Posts хранят копии данных, принадлежащих другому сервису, в базах данных локальных хранилищ. Эти локальные базы данных обновляются через обработку событий.

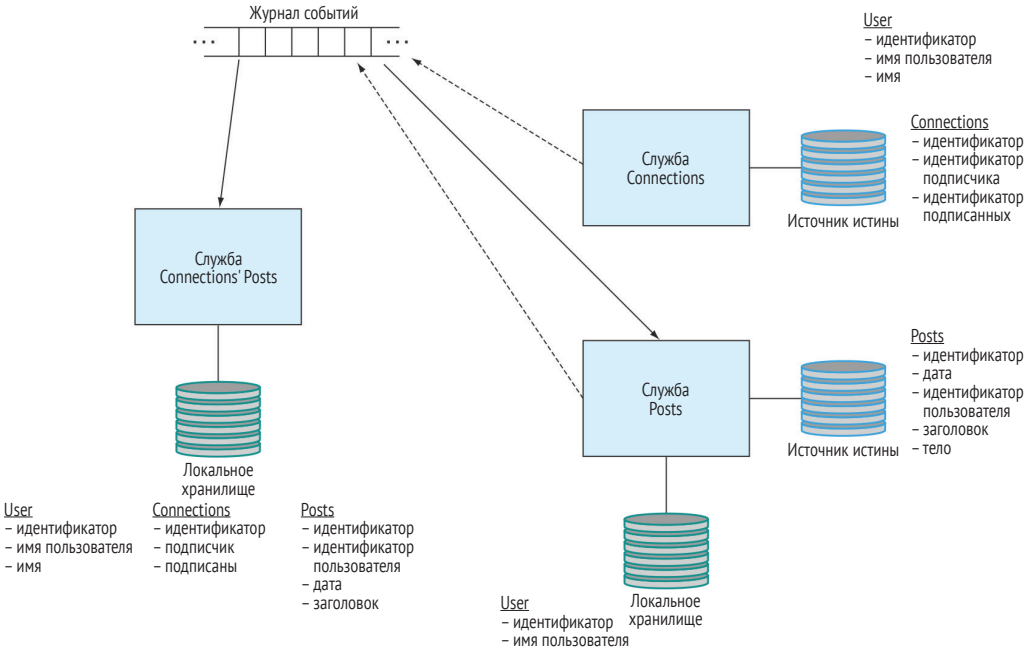


Рис. 12.12 ❖ Полная топология событий и данных нашей реализации. Службы Posts и Connections обслуживают «собственные» данные (источник истины) и генерируют события (пунктирные стрелки), когда в этих данных происходят изменения. Службы Posts и Connections' Posts хранят данные, которые им не принадлежат (локальное хранилище), и обновляют эти данные, обрабатывая события (сплошные стрелки)

Таблица 12.1 резюмирует роли, которые каждая служба играет в общей архитектуре программного обеспечения.

Обращаясь теперь к проекту, вы можете увидеть предыдущие детали, отраженные в структуре каталога и пакета. Поскольку вы сами можете все увидеть, позвольте мне перейти к настройке этого проекта.

Предполагая, что вы уже клонировали репозиторий, скачайте этот тег:

```
git checkout eventlog/0.0.1
```

Теперь вы можете перейти в каталог этой главы:

```
cd cloudnative-eventlog
```

Таблица 12.1. Роли служб в нашей событийно-ориентированной программной архитектуре

Роль	Служба Connections	Служба Posts	Служба Connections' Posts
Источник истины – выделенная база данных для хранения данных, принадлежащих службе	Эта служба управляет пользователями и связями между пользователями	Эта служба управляет постами	
Производитель записи и событий – реализации конечных точек HTTP обновляют источники истины и доставляют события в Kafka	Генерирует пользовательское событие каждый раз, когда пользователь создается, обновляется или удаляется. Генерирует событие подключения каждый раз, когда соединение создается или удаляется	Генерирует событие поста каждый раз, когда создается пост	
Обработчик событий – подписывается на события и соответственно обновляет базы данных локального хранилища		Обновляет базы данных локального хранилища, когда пользователи создаются, обновляются или удаляются	Обновляет локальное хранилище при создании, обновлении или удалении пользователей, подключений и постов
Локальное хранилище – выделенная база данных для хранения данных, не принадлежащих этой службе		Эта служба хранит сопоставления имени пользователя с идентификатором пользователя	Эта служба хранит данные о пользователях и соединениях, а также публикует сводные данные о постах
Чтение – реализации конечных точек HTTP, которые обслуживают доменные объекты службы	Пользователи. Соединения	Посты	Посты, сделанные пользователями, на которых вы подписаны

На рис. 12.13 показаны ключевые части структуры каталогов проекта, как описано в предыдущей таблице:

- API чтения реализуются в виде контроллеров, которые расположены в пакетах с суффиксом `read`. Все три сервиса поддерживают эти API;
- любые данные источника истины реализуются с помощью классов JPA, расположенных в пакете с суффиксом `sourceoftruth`. Этот пакет существует для служб Posts и Connections;
- API записи, поддерживающие HTTP-методы, такие как POST, PUT и DELETE, реализуются в виде контроллеров, которые расположены в пакетах с суффиксом `write`. В дополнение к сохранению данных в источнике истины службы эти контроллеры создают события в нашей структуре обмена сообщениями. Этот пакет существует для служб Posts и Connections;
- потребители событий реализованы в пакетах с суффиксом `eventhandlers`. Этот пакет существует для служб Posts и Connections' Posts;

- локальное хранилище данных, полученных посредством событий, реализовано с помощью классов JPA, расположенных в пакете с суффиксом `localstorage`. Этот пакет существует для служб Posts и Connections' Posts.

На рис. 12.14 представлен еще один способ просмотра микросервисов, структурированных таким образом, – на примере сервиса Posts. Данные поста (дата, название, содержимое и идентификатор автора) принадлежат службе и хранятся в источнике истины. Данные сопоставления идентификатора пользователя и его имени – лишь копия данных, принадлежащих другой службе, и хранятся в базе

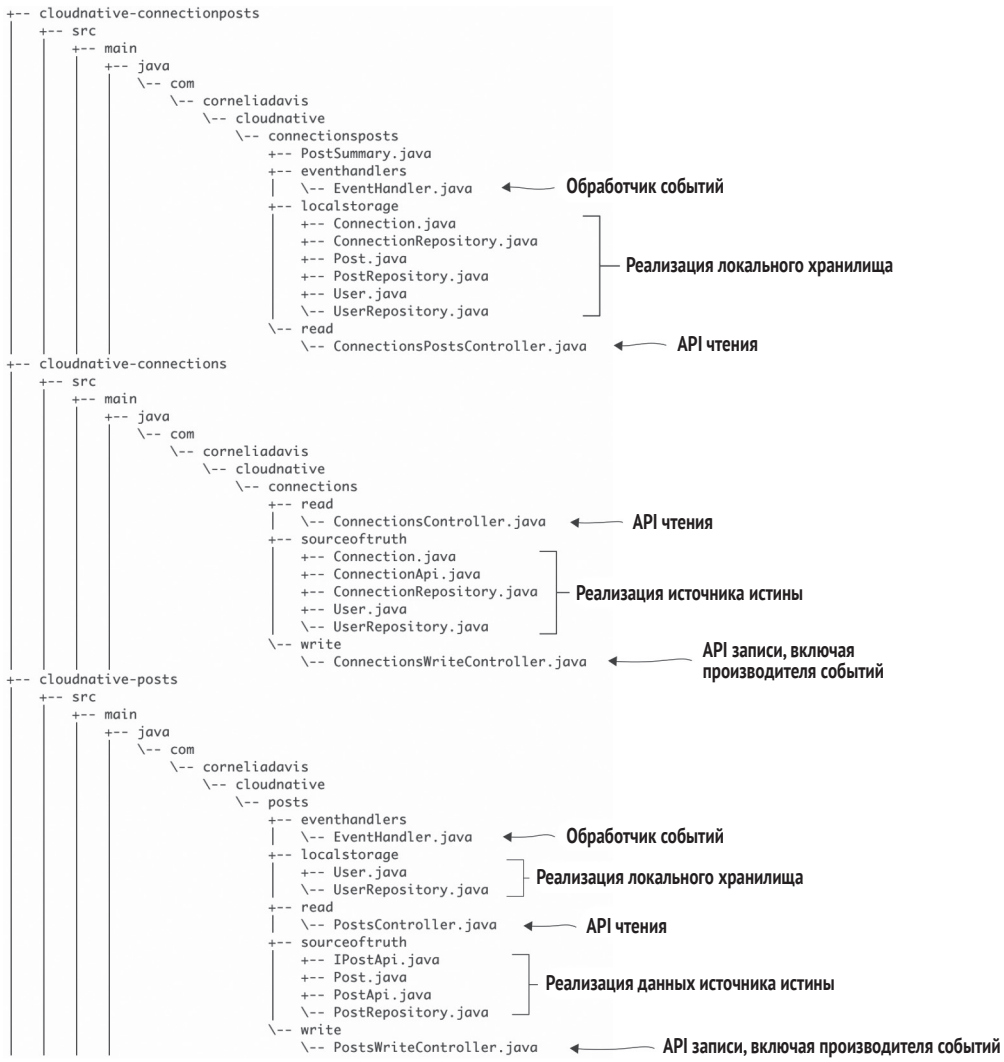


Рис. 12.13 ❖ Структура каталогов трех наших микросервисов, показывающая организацию кода, производящего и потребляющего события, а также API-интерфейсы чтения и «собственное» и «локальное» хранилища данных

данных локального хранилища. У нас есть контроллер записи, который выполняет два шага; он одновременно сохраняет данные в источнике истины и доставляет события в журнал событий. Обработчик событий выполняет одну задачу, потребляя события, представляющие интерес, и сохраняет данные в базе данных локального хранилища. И наконец, контроллер чтения выполняет одну задачу, запрашивая обе базы данных для создания списков поста.

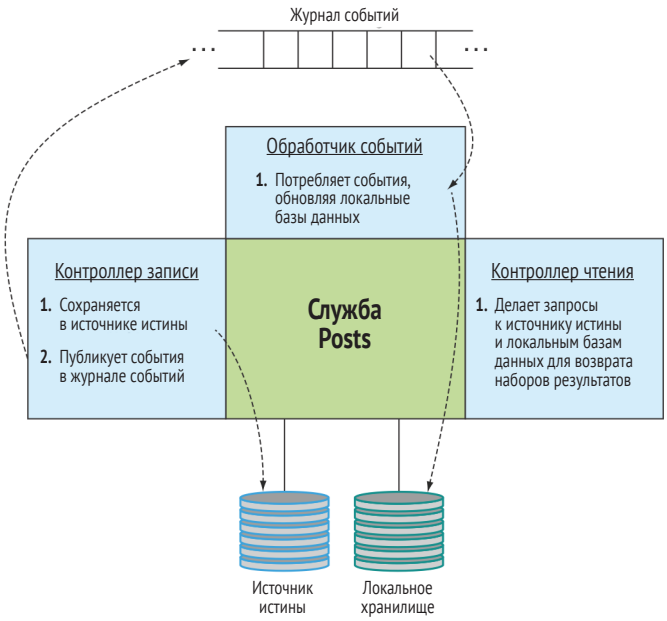


Рис. 12.14 ❖ Служба реализует контроллер записи, отвечающий за изменения в сущностях первого класса службы, обработчика событий, отвечающего за ведение локальной базы данных с данными неосновных сущностей, которые тем не менее являются частью ограниченного контекста, и контроллер чтения, поддерживающий запросы через эти базы данных

Разобравшись с этой структурой, давайте более подробно рассмотрим несколько частей реализации, начиная с контроллера записи службы Connections. Код в приведенном ниже листинге выполняется при создании нового пользователя.

Листинг 12.1 ❖ Метод из ConnectionsWriteController.java

```
@RequestMapping(method = RequestMethod.POST, value="/users")
public void newUser(@RequestBody User newUser,
                    HttpServletResponse response) {

    logger.info("Have a new user with username " + newUser.getUsername());

    // Сохраняем этого пользователя в нашей базе данных
    userRepository.save(newUser);          <-- Хранит данные в источнике истины

    // Отправляем событие в Kafka
    UserEvent userEvent =
```

```
    new UserEvent("created",
                  newUser.getId(),
                  newUser.getName(),
                  newUser.getUsername()); ◄——— Доставляет событие изменения
    kafkaTemplate.send(«user», userEvent);
}
```

Сначала новый пользователь сохраняется в источнике истины, а затем событие отправляется в журнал событий (из комментария видно, что вы используете Kafka – я расскажу о Kafka подробнее, когда мы перейдем к запуску этого примера). Вы отправляете пользовательское событие, которое фиксирует, что пользователь был создан с определенным идентификатором, именем и именем пользователя, и доставляете это событие в тему под названием `user`. Я подробнее расскажу о темах (и очередях) совсем скоро, а сейчас просто рассматривайте их как темы, с которыми вы знакомы по предыдущим реализациям, управляемым сообщениями. *Тема* – это просто канал, на который доставляются сообщения и из которого они потребляются.

Давайте сравним это с кодом из событийно-ориентированной реализации из главы 4 в листинге 12.2 (этот код можно найти в модуле `cloudnative-eventlog` , посмотрев тег репозитория `eventlogstart`).

Листинг 12.2 ❖ Метод из ConnectionsWriteController.java

```
@RequestMapping(method = RequestMethod.POST, value="/users")
public void newUser(@RequestBody User newUser,
                    HttpServletResponse response) {

    logger.info("Have a new user with username " + newUser.getUsername());

    // Сохраняем этого пользователя в нашей базе данных;
    userRepository.save(newUser);       ◄——————————— Сохраняет данные

    // Сообщаем заинтересованным сторонам об этом новом пользователе;
    // Необходимо уведомить службу Posts о новых пользователях;
    try {                                                    Отправляет событие в службу
                                                             Posts
        RestTemplate restTemplate = new RestTemplate(); ◄———
        restTemplate.postForEntity(postsControllerUrl+"/users",
                                   newUser, String.class);
    } catch (Exception e) {
        // Сейчас мы ничего не делаем - когда мы добавим журнал событий, все будет в порядке
        logger.info("problem sending change event to Posts");
    }

    // Необходимо уведомить службу Connections'Posts о новых пользователях;
    try {                                                    Отправляет событие в службу
                                                             Connections' Posts
        RestTemplate restTemplate = new RestTemplate(); ◄———
        restTemplate.postForEntity(connectionsPostsControllerUrl+"/users",
                                   newUser, String.class);
    } catch (Exception e) {
        // Сейчас мы ничего не делаем - когда мы добавим журнал событий, все будет в порядке
        logger.info("problem sending change event to ConnsPosts");
    }
}
```

Фрагмент этого кода показывает, что после сохранения пользователя в базе данных события доставлялись как в службу Posts, так и в службу Connections' Posts. С введением журнала событий вы смогли удалить эту связь – производителю событий не нужно знать что-либо о потребителях, – и данный код также, очевидно, намного проще, а следовательно, гораздо легче в обслуживании.

Давайте теперь обратимся к реализации потребителя событий. В приведенном ниже листинге показана часть кода обработчика событий для службы Connections' Posts.

Листинг 12.3 ❖ Методы из EventHandler.java

```
@KafkaListener(topics=»user»,
               groupId = “connectionspostsconsumer”,
               containerFactory = “kafkaListenerContainerFactory”)
public void userEvent(UserEvent userEvent) {

    logger.info(“Posts UserEvent Handler processing - event: “ +
                userEvent.getEventType());

    if (userEvent.getEventType().equals(“created”)) {

        // Делаем обработчика событий идемпотентным.
        // Если пользователь уже существует, ничего не делаем;
        User existingUser
            = userRepository.findByUsername(userEvent.getUsername());
        if (existingUser == null) {

            // Сохраняем запись в локальном хранилище;
            User user = new User(userEvent.getId(), userEvent.getName(),
                                 userEvent.getUsername());
            userRepository.save(user);

            logger.info(“New user cached in local storage “ +
                        user.getUsername());
            userRepository.save(new User(userEvent.getId(),
                                         userEvent.getName(),
                                         userEvent.getUsername()));
        } else
            logger.info(“Already existing user not cached again id “ +
                        userEvent.getId());
    } else if (userEvent.getEventType().equals(“updated”)) {
        // ... Обработка обновленного события;
    }

}

@KafkaListener(topics=»connection»,
               groupId = “connectionspostsconsumer”,
               containerFactory = “kafkaListenerContainerFactory”)
public void connectionEvent(ConnectionEvent connectionEvent) {

    // ... Обработка изменений в подключениях - кто на кого подписан;
    // Это созданные и удаленные события;
}

@KafkaListener(topics=”post”,
```

```
            groupId = "connectionspostsconsumer",
            containerFactory = "kafkaListenerContainerFactory")
public void postEvent(PostEvent postEvent) {

    // ... Обработка изменений в постах - т. е. новые посты;
}
```

В этой реализации есть несколько интересных моментов:

- из этого и предыдущего кодов, где показан производитель событий, видно, что детали взаимодействия с журналом событий абстрагированы от разработчика. Включение зависимости Spring Kafka в файл POM отрисовывает в клиенте Kafka, что позволяет разработчику использовать простой API для обозначения тем и других деталей, чтобы события можно было легко доставлять и принимать;
- обработчик событий службы Connections' Posts принимает сообщения из трех тем. Эти темы организуют события изменений для пользователей, подключений и постов. Хотя производитель просто предоставил название темы, потребители также должны предоставить `groupId`, который контролирует прием сообщений;
- в этом и показанном ранее коде производителя можно увидеть ссылки на схему событий: `UserEvent`, `ConnectionEvent` и `PostEvent`. События, публикуемые в журнал событий и принимаемые оттуда, должны иметь формат, а производитель и потребитель должны знать детали.

В этом коде можно увидеть комментарий по поводу того, чтобы сделать логику потребителя для события, созданного пользователем, идемпотентной. В распределенной системе гарантированная доставка событий ровно один раз может быть сложной и дорогостоящей с точки зрения производительности. Когда вы делаете сервисы идемпотентными, это снимает с вас часть этого бремени.

Мы подробно рассмотрим каждую из этих тем, но сначала давайте запустим код.

Настройка

В последний раз я отсылаю вас к инструкциям по настройке для запуска примеров, приведенным в предыдущих главах. В этой главе нет новых требований для запуска образца.

Вы будете получать доступ к файлам в каталоге cloudnative-eventlog, поэтому в окне терминала перейдите в этот каталог.

И, как я уже писала в предыдущих главах, я предварительно собрала образы Docker и сделала их доступными в Docker Hub. Если вы хотите собрать исходный код Java и образы Docker и перенести их в свой собственный репозиторий образов, обратитесь к предыдущим главам (самые подробные инструкции приведены в главе 5).

Запуск приложения

Этот пример можно запустить на небольшом кластере Kubernetes, поэтому если хотите, то можете использовать Minikube или что-то похожее. Если у вас еще есть работающие примеры из предыдущей главы, очистите их. Для этого запустите предоставленный мной скрипт:

```
./deleteDeploymentComplete.sh all
```

После этого будут удалены все экземпляры служб Posts, Connections и Connections' Posts, а также любые работающие службы MySQL, Redis или SCCS. Если в вашем кластере Kubernetes работают другие приложения и службы, вы также можете удалить некоторые из них; просто убедитесь, что у вас хватает емкости. В предыдущих главах удаление сервисов было необязательным, но в этом случае я рекомендую вам удалитьих все. Некоторые из них здесь не используются (SCCS), и я бы хотела, чтобы вы использовали свежий экземпляр MySQL, поскольку топология базы данных здесь довольно сильно отличается.

Существует небольшая зависимость порядка старта в том, что после создания сервера MySQL вам необходимо создать в нем действительные базы данных. Итак, сначала запустите журнал событий:

```
./deployServices.sh
```

Как только база данных MySQL будет готова, что можно увидеть, выполнив команду `kubectl get all`, вы создадите базы данных, используя интерфейс командной строки `mysql`:

```
mysql -h $(minikube service mysql-svc --format "{{.IP}}")
➥ -P $(minikube service mysql-svc --format "{{.Port}}") -u root -p
```

Пароль – `password`, и после входа в систему выполните три эти команды:

```
create database cookbookconnectionsposts;
create database cookbookposts;
create database cookbookconnections;
```

Обратите внимание, что сейчас вы создаете базы данных для каждой из служб; до сих пор вы всегда создавали только одну базу данных. Это означает, что вы реализовывали именно то, что было изображено на рис. 12.2!

Теперь вы можете запустить микросервисы, выполнив этот скрипт:

```
./deployApps.sh
```

Все готово, и вы можете загружать данные. Но подождите. Если вы все время запускали примеры, то, вероятно, заметили, что прежде я никогда не просила вас загружать данные. Почему сейчас?

Причина состоит в том, что вы разбили базу данных на части и у вас есть источники истины и базы данных локального хранилища, и я хочу, чтобы вы использовали ту обработку событий, которую встроили в свои службы, для загрузки данных во все эти хранилища. И я хочу, чтобы вы посмотрели, что происходит, когда эти данные загружаются, поэтому передавайте журналы из каждого своего микросервиса, по одному в трех разных окнах терминала, с помощью этой команды (помните, что имена модулей можно получить с помощью команды `kubectl get all`):

```
kubectl logs -f po/<name of your pod instance>
```

Если вы посмотрите в конец журнала службы Posts, то увидите эту строку:

```
o.s.k.l.KafkaMessageListenerContainer : partitions assigned:[user-0]
```

И если вы посмотрите в конец журнала службы Connections' Posts, то увидите это:

```
o.s.k.l.KafkaMessageListenerContainer : partitions assigned:[post-0]
o.s.k.l.KafkaMessageListenerContainer : partitions assigned:[connection-0]
o.s.k.l.KafkaMessageListenerContainer : partitions assigned:[user-0]
```

Это показывает, что служба Posts прослушивает события в теме `user`, а служба Connections' Posts прослушивает события в каждой из тем `user`, `connection` и `post`. Ага, вы видите, что наша топология обработки событий оживает! Теперь загрузите данные и увидите этот поток в действии. Выполните эту команду:

```
./loadData.sh
```

Если вы посмотрите в этот файл, то увидите, что я создала те же образцы данных, которые вы использовали все это время. Посмотрите еще раз на журналы.

Журнал службы Connections показывает именно те записи, которые вы ожидаете увидеть, – создание трех пользователей и соединений между ними:

```
...ConnectionsWriteController : Have a new user with username madmax
...ConnectionsWriteController : Have a new user with username gmaxdavis
...ConnectionsWriteController : Have a new connection: madmax is following cdavisafc
...ConnectionsWriteController : Have a new connection: cdavisafc is following madmax
...ConnectionsWriteController : Have a new connection: cdavisafc is following gmaxdavis
```

Вывод журнала службы Posts немного интереснее; вы видите, что здесь вызывается слушатель для обработки трех пользовательских событий и для создания трех пользователей в вашем локальном хранилище пользователей:

```
...EventHandler : Posts UserEvent Handler processing - event: created
...EventHandler : New user cached in local storage cdavisafc
...EventHandler : Posts UserEvent Handler processing - event: created
...EventHandler : New user cached in local storage madmax
...EventHandler : Posts UserEvent Handler processing - event: created
...EventHandler : New user cached in local storage gmaxdavis
```

Позже в том же журнале вы видите сообщения о создании постов:

```
...PostsWriteController : Have a new post with title Cornelia Title
...PostsWriteController : find by username output
...PostsWriteController : user username = cdavisafc id = 1
...PostsWriteController : Have a new post with title Cornelia Title2
...PostsWriteController : find by username output
...PostsWriteController : user username = cdavisafc id = 1
...PostsWriteController : Have a new post with title Glen Title
...PostsWriteController : find by username output
...PostsWriteController : user username = gmaxdavis id = 3
```

В реализацию средства записи службы Posts я включила сообщение журнала, в котором показан поиск идентификатора пользователя по имени пользователя, последнее из которых предоставляется как часть поста. Конечно, этот поиск происходит в нашей базе данных локального хранилища пользователей.

Наконец, просматривая журналы службы Connections' Posts, вы видите то, что, как я надеюсь, ожидаете увидеть, – сообщения о том, что события `user`, `connection` и `post` обрабатываются, а данные сохраняются в ваших локальных базах данных:

```
...EventHandler. : Posts UserEvent Handler processing - event: created
...EventHandler. : New user cached in local storage cdavisafc
```

```
...EventHandler. : Posts UserEvent Handler processing - event: created
...EventHandler. : New user cached in local storage madmax
...EventHandler. : Posts UserEvent Handler processing - event: created
...EventHandler. : New user cached in local storage gmaxdavis
...EventHandler. : Creating a new connection in the cache:2 is following 1
...EventHandler. : Creating a new connection in the cache:1 is following 2
...EventHandler. : Creating a new connection in the cache:1 is following 3
...EventHandler. : Creating a new post in the cache with title Max Title
...EventHandler. : Creating a new post in the cache with title Cornelia Title
...EventHandler. : Creating a new post in the cache with title Cornelia Title2
...EventHandler. : Creating a new post in the cache with title Glen Title
...EventHandler. : Posts UserEvent Handler processing - event: created
```

Давайте выполним еще несколько команд, чтобы посмотреть на них в действии.

Теперь, когда вы запрашиваете у службы Connections' Posts посты пользователей, на которых я подписана:

```
curl $(minikube service \
         --url connectionsposts-svc)/connectionsposts/cdavisafc | jq
```

вы видите это:

```
[
  {
    «date»: «2019-01-22T01:06:19.895+0000»,
    “title”: “Chicken Pho”,
    “usersName”: “Max”
  },
  {
    “date”: “2019-01-22T01:06:19.985+0000”,
    “title”: “French Press Lattes”,
    «usersName»: «Glen»
  }
]
```

Давайте теперь изменим имя пользователя. Используя приведенный ниже код, измените имя пользователя с `cdavisafc` на `cdavisafcupdated`:

```
curl -X PUT -H “Content-Type:application/json” \
      --data ‘{“name”:”Cornelia”,”username”:”cdavisafcupdated”}’ \
      $(minikube service --url connections-svc)/users/cdavisafc
```

Еще раз, давайте посмотрим на журналы. Во-первых, обратите внимание, что каждый из журналов Posts и Connections' Posts обработал это событие. Каждый журнал показывает вот такую запись:

```
...: Updating user cached in local storage with username cdavisafcupdated
```

Теперь запросы к службе Connections' Posts и службе Posts (напомню, что все три службы реализуют контроллеры чтения) отражают это изменение:

```
curl $(minikube service --url \
   connectionsposts-svc)/connectionsPosts/cdavisafc
curl $(minikube service --url \
   connectionsposts-svc)/connectionsPosts/cdavisafcupdated
curl $(minikube service --url posts-svc)/posts
```

Я рекомендую вам изучить топологию потока событий, выполнив дополнительные команды, просмотреть журналы и увидеть результаты. Напомню, что в Posts и Connections также есть API-интерфейсы чтения, включая конечные точки, которые позволяют вам видеть каждую из коллекций объектов, которыми они управляют:

```
curl $(minikube service --url connections-svc)/users
curl $(minikube service --url connections-svc)/connections
curl $(minikube service --url posts-svc)/posts
```

На рис. 12.15 топология событий представлена более подробно. На ней можно увидеть различных производителей, потребителей и темы, и то, как они связаны друг с другом.

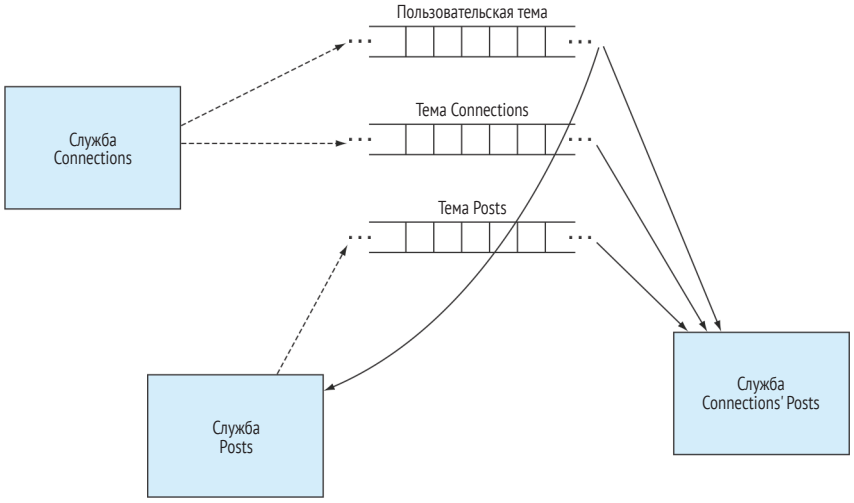


Рис. 12.15 ❖ Топология событий определяет темы, которые будут содержать события, а также производителей и потребителей каждой из них. Служба Connections производит события пользователей и подключений. Служба Posts производит события постов и потребляет события пользователей. Служба Connections' Posts потребляет события пользователей, подключений и постов

Последний комментарий, прежде чем я перейду к еще трем архитектурным темам журнала событий: вы, вероятно, обратили внимание на команду `sleep` в скрипте `loadData.sh`. Да, это выглядит немного приевшимся, но помогает избежать состояния гонки, когда команда `curl`, чтобы создать пост, добирается до службы Posts до того, как события для создания пользователя будут обработаны этой службой. Вы можете обработать этот сценарий иным способом, например выполнив простую повторную отправку запроса из кода, отправив событие в тему «retry» для повторной попытки обработки в более позднее время или даже потерпев неудачу при создании поста. Анализ этих вариантов и выбор правильного – интересная тема, но она выходит за рамки этой главы. Взгляните на ссылку, которую я привела в конце данной главы.

Хорошо, теперь вернемся к трем темам, которые я поставила в очередь для дальнейшего обсуждения:

- различные типы каналов обмена сообщениями и целесообразность их применения в программном обеспечении для облачной среды;
- полезная нагрузка события;
- ценность идемпотентных сервисов.

12.3.2. Что нового в темах и очередях?

Те из вас, кто работал или знаком с основами систем обмена сообщениями в прошлом, возможно, с системами JMS (Java Messaging Service), знают об очередях и темах. Но если это было какое-то время назад, позвольте мне пошевелить вашу память. У обеих абстракций есть издатели и подписчики, действующие субъекты, которые создают сообщения в именованный канал и принимают сообщения оттуда (тема или очередь), но способы обработки сообщений – как структурой обмена сообщениями, так и потребителями – различаются.

В случае когда может быть несколько подписчиков, одно сообщение будет обрабатываться только *одним* из подписчиков. Кроме того, если на момент появления сообщения в канале доступных подписчиков нет, оно будет храниться до тех пор, пока подписчик не примет его. После этого сообщение исчезнет.

В случае с *темами*, у которых также может быть несколько подписчиков, одно сообщение будет отправляться *всем* подписчикам в момент его создания. Сообщение будет обработано любым числом потребителей, даже если оно будет равно нулю. Потребители будут получать сообщения, только если они прикреплены к теме на момент создания сообщения. Если потребитель не подключен к теме при создании сообщения, он никогда его не увидит. Если в теме нет подписчиков, доставленные туда сообщения просто исчезнут. Можно рассматривать тему как предоставление дополнительной функции маршрутизации сообщений. На рис. 12.16 изображены обе очереди и темы.

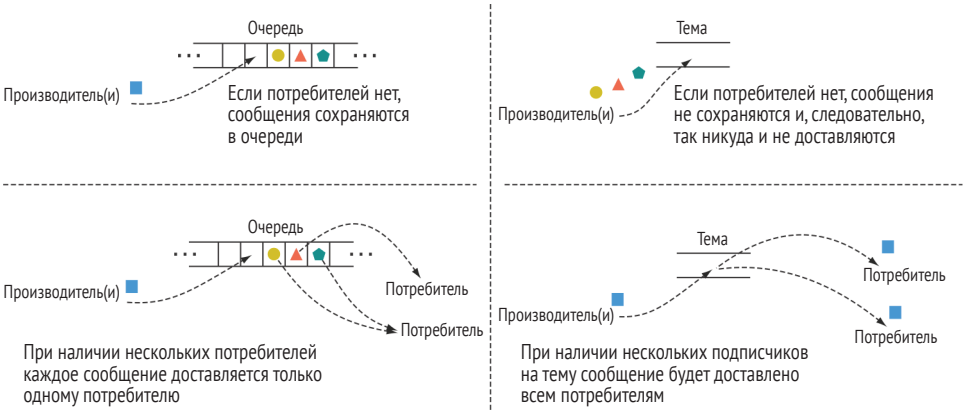


Рис. 12.16 ❖ Системы обмена сообщениями, в основном стандартизированные с помощью JMS, обеспечивают поддержку очередей и тем. Очереди сохраняют сообщения до тех пор, пока каждое из них не будет потреблено. Темы не сохраняют никаких сообщений, а вместо этого просто доставляют их всем доступным подписчикам

Хотя в новом *журнале событий* используются некоторые из тех же понятий, что и во времена JMS (брокеры, производители, потребители и даже темы), семантика вокруг некоторых из них несколько иная, что может привести к совершенно другому поведению. С точки зрения разработчика, брокер выглядит практически так же, как и раньше; это канал для подключения производителей и потребителей к журналу событий. Однако внедрение современной системы регистрации событий, такой как Apache Kafka, обычно более ориентировано на облачную среду, что позволяет брокерам начинать или прекращать свое существование по мере изменения инфраструктуры и/или масштабирования кластера журнала событий. Роль производителя также почти такая же, как и раньше; он будет подключаться к журналу событий через брокера и доставлять события.

Когда речь заходит о темах и потреблении, все меняется. Потребители указывают на интерес к событиям, которые публикуются по определенной теме. Но то, как возникают события, когда существует множество потребителей, становится интересным. Вы уже достаточно видели в наших архитектурах для облачной среды, что у нас будет много разных микросервисов, на каждом из которых будет развернуто несколько экземпляров. Шаблоны потребления журнала событий оптимизированы для этого варианта использования.

> **ПРИМЕЧАНИЕ** Нам нужна семантика, которая позволяет различным микросервисам потреблять одно и то же событие, но иметь только один экземпляр каждого процесса.

В Kafka эта семантика поддерживается двумя абстракциями: *темой* и *идентификатором группы*. Любой микросервис, который заинтересован в определенном событии, создаст слушателя для темы, которая несет событие, и любое новое событие будет доставляться *всем* слушателям – всем микросервисам. `groupId` используется, чтобы гарантировать, что только один экземпляр определенного микросервиса получает сообщение; оно будет доставлено только одному из всех экземпляров, которые совместно используют один и тот же идентификатор группы.

Самый простой способ разобраться в этом – контекст нашего приложения, которое мы используем в качестве примера.

Напомню, что события, которые фиксируют создание нового пользователя или изменения существующего, должны потребляться как службой Posts, так и службой Connections' Posts. Поэтому обе эти службы будут слушать в теме `user`. Но затем, поскольку вам нужно, чтобы только один экземпляр службы Posts и только один экземпляр службы Connections' Posts обрабатывал событие, вы устанавливаете `groupId` одинаково для всех экземпляров Posts и то же самое делаете для всех экземпляров службы Connections' Posts, как показано здесь – в обработчике событий Posts:

```
@KafkaListener(topics="user", groupId="postsconsumer")
public void listenForUser(UserEvent userEvent) {
    ...
}
```

и в обработчике событий Connections' Posts:

```
@KafkaListener(topics="user", groupId = "connectionspostsconsumer",
               containerFactory = "kafkaListenerContainerFactory")
public void userEvent(UserEvent userEvent) {
    ...
}
```

По сути, тема событий – это нечто вроде гибрида старой темы и очереди старой очереди; тема событий действует как тема для потребителей с разными идентификаторами групп, а также как очередь для потребителей, у которых один и тот же идентификатор группы. Эта топология потребления событий показана на рис. 12.17.

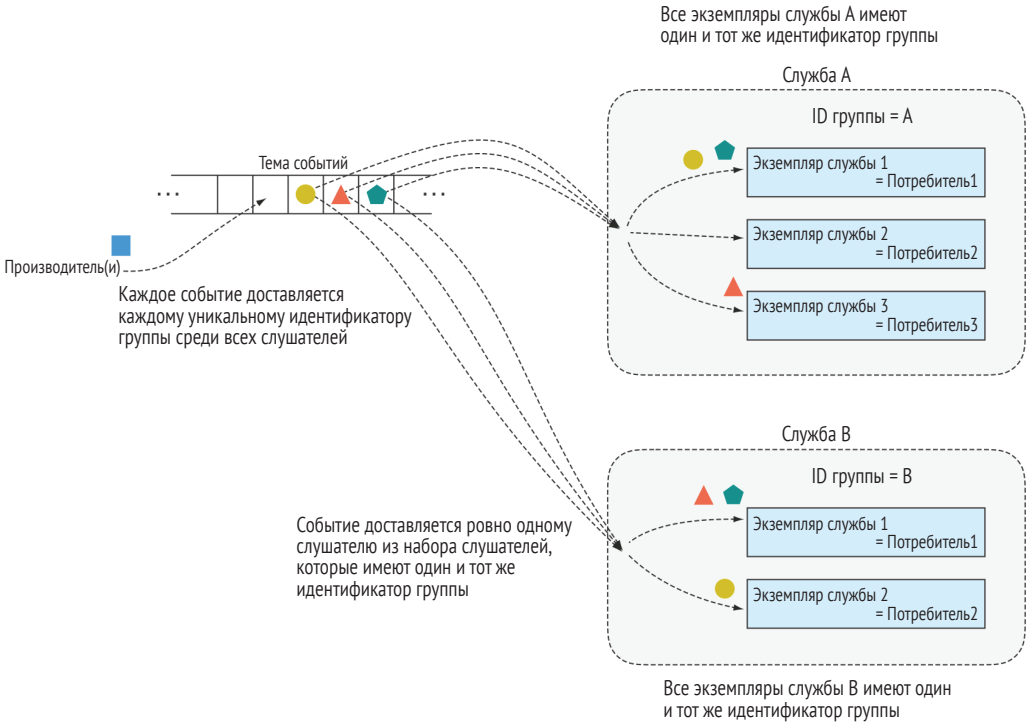


Рис. 12.17 ❖ Темы событий в Kafka обладают поведением, которое представляет собой гибрид тем JMS и очередей. Как и в старых темах, события доставляются всем уникальным групповым идентификаторам – здесь несколько подписчиков. Но в пределах набора слушателей с одинаковым идентификатором группы поведение напоминает очередь, доставляя событие одному из слушателей

Тема журнала событий отличается от темы обмена сообщениями другим способом, и в ее названии содержится подсказка, а именно слово «журнал». Будь то журнал приложения, направляемый в Splunk, или журнал записи в базу данных, мы обычно рассматриваем журналы как нечто, где хранятся записи, возможно, бессрочно. Сохранение журнала событий является фундаментальным принципом нашей архитектуры на базе журналов событий. События хранятся для любых потребителей, которые могут быть заинтересованы в них, даже если потребитель не подключен во время создания события. Вы уже встречались с этим во время анализа устойчивости и результирующих характеристик автономности сервисов ранее в этой главе; на рис. 12.11 вы видели, что стороны, заинтересованные в событии, могут выбрать их через некоторое время после того, как они будут созданы.

Здесь я хочу довести концепцию до предела, предела, в котором, как я утверждаю, на самом деле вовсе нет ничего странного. События в журнале событий сохраняются для любого потребителя, который может во всякий момент в будущем заинтересоваться им, *даже для потребителей, о которых мы и не думали и которые еще не существуют*! Верно, мы хотим, чтобы события были поблизости, чтобы при появлении у нас идеи относительно того, что мы хотим сделать (изучите модель машинного обучения, чтобы придумать предложения для кулинарных блогов, которые вам нравятся, основываясь на истории пользователей, связей и постов за последние два года), вы могли бы разработать это программное обеспечение, и оно смогло бы потреблять все события, представляющие интерес. Сохранение двухлетней истории событий поддерживает нынешние инновации.

Да, это означает, что нам потребуется много места для хранения, но затраты на хранение снизились до такой степени, что им можно управлять, и полученная возможность значительно превышает его. Я не сомневаюсь, что практически любая организация должна подумать о создании и сохранении таких типов журналов событий, как для насущных потребностей, о которых я рассказывала в этой главе, а также для тех, что относятся к будущему.

Я хочу быть откровенной, говоря об одной последней вещи, когда речь идет о темах, хотя она уже едва ощутимо проявилась в наших обсуждениях: каждый потребитель поддерживает свой курсор в журнале событий / теме. *Курсор* – это место в журнале событий, где они потребляли предыдущие события. В результате каждый потребитель может обрабатывать события по собственному расписанию – даже те воображаемые потребители, о которых я говорила минуту назад! И здесь мы возвращаемся к понятию автономности. Вспомните семантику темы обмена сообщениями – все потребители должны были находиться в сети в тот самый момент, когда было создано сообщение. Тесная связь!

Когда потребители могут управлять собственными курсорами, это позволяет использовать интересные шаблоны. Например, допустим, вы потеряли базу данных службы Connections' Posts. Поскольку все таблицы были сгенерированы из данных журнала событий, чтобы восстановить службу Connections' Posts, вам нужно всего лишь настроить новую пустую базу данных, установить курсор в начало журнала событий и выполнить повторную обработку всех событий. Нет необходимости в коде из основной ветви разработки, который отличается от кода восстановления после сбоя; результаты будут предсказуемо такими же.

Так ли это? Как это часто бывает, ответ на такие открытые вопросы звучит так: как посмотреть. Давайте ознакомимся с содержанием следующего раздела.

12.3.3. Полезные данные события

Здесь я хочу коснуться некоторых аспектов, связанных с полезными данными событий, но начну с того, что я имела в виду, когда задала вопрос о предсказуемости минуту назад. Вы хотите иметь возможность воссоздавать свое состояние, например состояние службы Connections'Posts, полностью из сообщений в журнале событий. Все данные, необходимые для получения этого состояния, должны быть указаны в событиях – ссылки на внешние источники не допускаются.

Позвольте мне проиллюстрировать это на примере. Скажем, когда служба Posts публикует событие в теме, чтобы сэкономить место в журнале событий, вы решаете не включать сюда текст поста в блоге, а только URL-адрес поста. Все потребите-

ли знают, что это формат события, поэтому если им нужно обработать тело поста, возможно, чтобы провести анализ тональности текста, они могут получить тело, перейдя по ссылке. Хотя изначально это может сработать, что произойдет, если пост будет удален? Повторная обработка журнала событий позже не позволит вам сгенерировать тот же результат, что был у вас первоначально. Это подводит нас к первому правилу.

ПОЛЕЗНЫЕ ДАННЫЕ СОБЫТИЯ, ПРАВИЛО № 1 Событие, которое публикуется в журнале событий, должно быть описано *в полном объеме*.

Я знаю, о чем вы думаете: «Я ни за что не помещу изображения с высоким разрешением в журнал событий!» И я не спорю – вы не должны этого делать. Я полагаю, что асинхронная обработка изображений не подходит для того типа рабочего процесса, управляемого событиями, который я только что описала. Да, вы хотите, чтобы происходила какая-то обработка, например в результате отображения изображения в определенном месте файловой системы. Вы захотите поместить уведомление об этом новом изображении в тему (или, скорее, в очередь), но я предлагаю, чтобы то, что помещено в тему, URL-адрес, по которому можно найти изображение, было скорее сообщением, нежели событием. Я утверждаю, что если полезные данные не являются полным представлением события, из которого можно получить состояние, то пусть это будет сообщение. Конечно, вполне приемлемо продолжать использовать шаблоны передачи сообщений, даже если вы используете шаблоны регистрации событий в других частях своей системы; все это вопрос выбора правильного шаблона для работы.

Теперь давайте подробно рассмотрим детали события, в частности схему. События имеют структуру, и важно, чтобы потребители знали эту структуру и могли надлежащим образом обрабатывать полезные данные. Вопрос в том, кто отвечает за эту структуру. В начале 2000-х годов одним из самых популярных ответов был этот: сервисная шина предприятия (ESB). Идея заключалась в том, что будет определяться каноническая модель данных и, если хотите, «устанавливаться» в сервисную шину. Производители затем преобразуют свое сообщение в каноническую модель как часть входа в ESB, а потребители аналогичным образом выполняли преобразование из канонической модели в собственный выход. У сервисных шин были целые фреймворки, касающиеся производства и управления каноническими моделями и преобразованиями. Все это было очень тяжелым и очень неповоротливым.

Я стала рассматривать ESB как незначительное постепенное развитие централизованной, канонической, корпоративной базы данных. Посмотрите на рис. 12.2. Все «независимые» микросервисы на этом рисунке вообще-то не очень независимы, когда привязаны к одной централизованной базе данных. Оказывается, что перемещение этой централизации в структуру обмена сообщениями не сделало наши системы менее связанными, чем они были раньше. Вместо этого нам нужно, чтобы производители могли доставлять события в той форме, которая является естественной для их области. Потребители, которые в любом случае должны были адаптироваться к канонической модели в прошлом, будут адаптироваться к модели от производителя.

ПОЛЕЗНЫЕ ДАННЫЕ СОБЫТИЯ, ПРАВИЛО № 2 Для журнала событий не существует канонической модели событий. Производители контролируют формат данных событий, которые они поставляют, а потребители адаптируются к этому.

Самые наблюдательные из вас заметят, что в проекте cloudnative-eventlog есть еще один модуль под названием cloudnative-eventschemas, и он немного напоминает централизованную модель данных. Сознаюсь. Виновна. Я создала этот модуль и использовала его для событий во всех трех наших микросервисах только для удобства, чтобы наш проект, используемый в качестве примера, был как можно проще. В действительности у служб Connections, Posts и Connections' Posts должны быть собственные репозитории, и каждая служба должна определять схемы генерируемых ими событий и схемы данных, представляющих интерес в событиях, создаваемых другими службами.

И это подводит меня к последнему пункту, который я хочу отметить, говоря о полезных данных событий: схемами для этих полезных данных нужно управлять. Производители, которые «владеют» схемами событий, которые они производят, должны адекватно описывать *и версионировать* эти схемы, а описания должны быть доступны для всех заинтересованных сторон в официальном порядке. Вот почему в Confluent, компании, предоставляющей коммерческое предложение на базе Apache Kafka, есть реестр схем[1]. Реестр схем также можно использовать для сериализации и десериализации событий – полезные данные не должны быть в формате JSON в журнале событий. Подводя итог, просто потому что мы не централизуем схему в единую каноническую модель, не означает, что у нас тут дикий, дикий запад; там по-прежнему нужен порядок.

> **ПОЛЕЗНЫЕ ДАННЫЕ СОБЫТИЯ, ПРАВИЛО № 3** Все события, опубликованные в журнале событий, должны иметь связанную схему, к которой могут обращаться все заинтересованные стороны, и схемы должны иметь версии.

То, о чем я рассказала здесь, лишь поверхностно раскрывает тему полезных данных событий, но я надеюсь, что этот рассказ адекватно выделил некоторые ключевые принципы и их отличие от систем на базе сообщений, с которыми вы, возможно, знакомы из прошлого. Как со всем остальным в системах для облачной среды, основное внимание уделяется автономности, а не централизации; эволюции, а не прогнозированию; адаптации, а не строгости.

12.3.4. Идемпотентность

Теперь давайте вернемся к комментарию в коде обработчика событий в службе Connections' Posts:

```
// make event handler idempotent.
```

На мгновение давайте подумаем о потоке, который происходит в трех наших микросервисах, без осложнений, связанных с их распределением. Новые пользователи, соединения и посты поступают в службы Connections and Posts, которые, в свою очередь, доставляют события в журнал событий. Потребители берут эти события и обрабатывают их, выполняя записи в соответствующие локальные базы данных. Это простой поток. Но мы знаем, что многое может пойти не так, как только мы распределим компоненты.

Для начала у производителя могут возникнуть проблемы с доставкой события в журнал событий. Скажем, производитель отправляет событие в журнал, но не

[1] Реестр схем от компании Confluent предоставляет множество сервисов, которые поддерживают как создателей, так и потребителей событий: **http://mng.bz/GWAM**.

получает подтверждение того, что оно было получено. В этом случае он попробует снова. Со второй попытки он получает подтверждение, и дело сделано. Но учтите: когда первая попытка записи в журнал не была подтверждена, возможно, что фактически запись была осуществлена, но подтверждение было потеряно до того, как было получено производителем. В этом случае со второй попытки второе событие вполне могло бы быть помещено в журнал.

Вы можете избежать дублирования записей журнала различными способами (например, большинство структур обмена сообщениями поддерживает семантику *exactly once* (ровно один раз)), но это может быть затратно с точки зрения производительности. В результате семантика *at-least-one* (хотя бы раз) является предпочтительной, что эффективно переносит ответственность за дедупликацию на потребителя.

Вот тут вступает в дело идемпотентность. *Идемпотентная* операция – это операция, которую можно применить один или *несколько* раз, при этом результат будет тот же. Например, при создании новой записи в локальной базе данных проверка на предмет того, была ли она уже создана, и отсутствие действий в этом случае делают операцию идемпотентной. Операция удаления обычно идемпотентна, потому что если вы удаляете объект один раз или более одного раза, в итоге состояние остается тем же – объект удаляется. Обновление сущностей также обычно идемпотентно.

Если вы пишете потребителя, который не идемпотентен, это накладывает ограничения на способ его использования. Например, он навязывает семантику *exactly once* (или *at most once*) в протоколе доставки события. Если, с другой стороны, вы пишете потребителя, который идемпотентен, это позволяет применять его в более широком диапазоне вариантов использования.

ПОЛЕЗНЫЕ ДАННЫЕ СОБЫТИЯ, ПРАВИЛО № 1 Если это вообще возможно, делайте операции вашего потребителя событий идемпотентными.

12.4. Порождение событий

Уф. Довольно много информации – много нового, о чем стоит поразмыслить, и множество новых способов применения знакомых концепций. И все же мы пока еще не закончили. Однако прежде чем приступить к обсуждению последней темы, я хочу напомнить вам, с чего мы начали и к чему мы стремимся в ходе этого обсуждения.

12.4.1. Путешествие еще не окончено

Эта глава посвящена *данным* в облачной среде. По сути, мы говорим о разрушении монолита данных, потому что без этого компонентизация вычислительного монолита дает нам только иллюзию автономности. Микросервис, имеющий ограниченный контекст, является для нас естественным местом, где мы также определяем порог для хранения данных, поэтому первым шагом является то, что каждый микросервис получает свою собственную базу данных. Вопрос в том, как эти локальные хранилища данных наполняются. Один из вариантов – это кеширование, и мы видим мгновенную выгоду в автономности и результирующей устойчивости. Но кеширование работает гораздо лучше для редко меняющегося контента, чем для контента, который меняется часто и ассоциирован с моими микросерви-

сами, что влечет за собой множество сложностей для поддержания данных в актуальном состоянии. Это приводит нас к событийно-ориентированному подходу, где изменения активно распространяются через сеть сервисов, образующих наше программное обеспечение. Но события можно делать способом, который слишком тесно связывает различные сервисы друг с другом, поэтому мы используем знакомый шаблон «издатель/подписчик». Но вместо того чтобы называть это *обменом сообщениями*, мы называем это *журналом событий*, потому что, помимо других важных различий, журнал событий следует рассматривать как нечто, что сохраняет записи в течение неопределенного времени, и все потребители могут прокладывать себе путь к журналу таким способом, который соответствует их потребностям. Это показано на рис. 12.18.

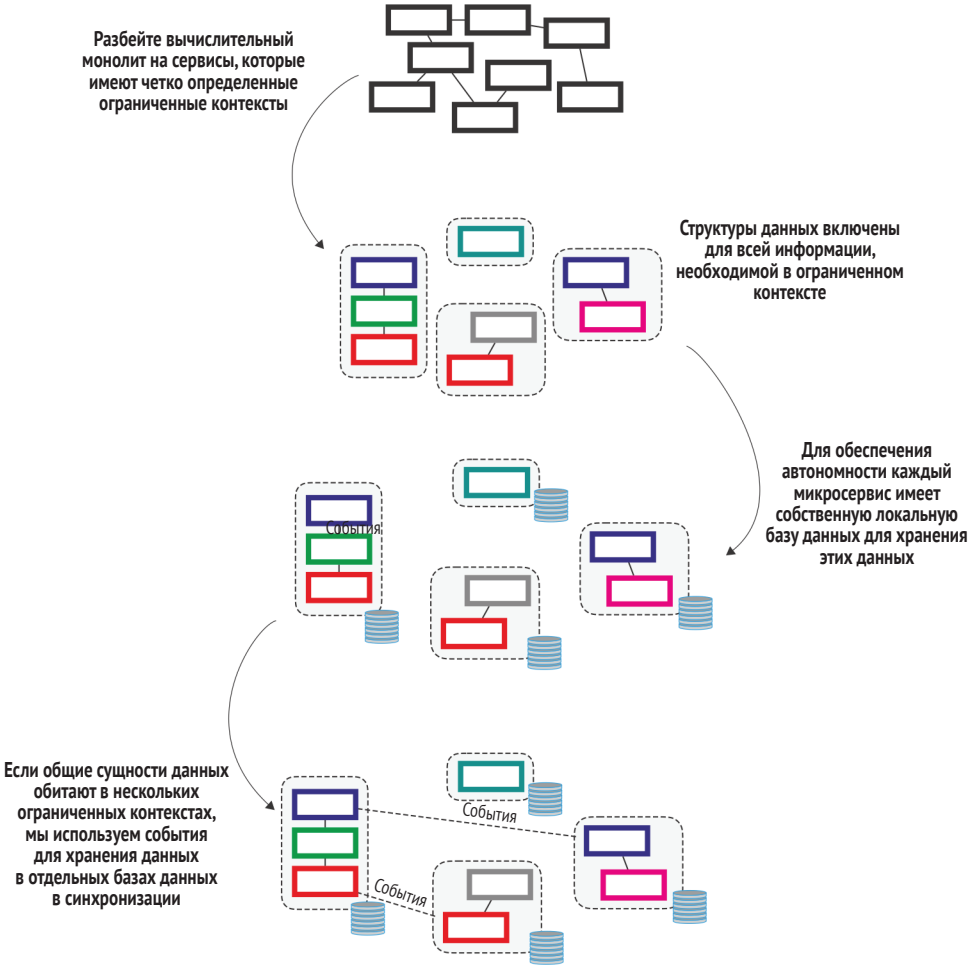


Рис. 12.18 ❖ Наше получение данных для облачной среды начинается с разделения монолита данных таким образом, чтобы данные поддерживали ограниченный контекст микросервисов. Микросервисы получают собственные базы данных, и, где это необходимо, данные сохраняются согласованными в распределенной структуре данных посредством обработки событий

Несмотря на то что, на мой взгляд, кажется разумной прогрессией, которая решает все больше и больше проблем распространения данных с помощью нашего программного обеспечения для облачной среды, я надеюсь, что у вас будет хотя бы смутное чувство дискомфорта. Существуют и более сложные пограничные случаи, чем те, например, о которых я говорила в предыдущем разделе «Идемпотентность». И краткий обзор по обработке полезных данных мог бы оставить больше вопросов, чем ответов. Но над архитектурой, которая у меня получилась, нависает еще большее облако с вопросом: *кто какими данными «владеет»?*

12.4.2. Источник истины

Выяснить в нашем приложении, использующемся в качестве примера, кому какие данные принадлежат, довольно просто. Служба Connections владеет пользователями и соединениями, а служба Posts – постами. Но в более реалистичной обстановке ответить на этот вопрос гораздо сложнее. «Владеет» ли программное обеспечение, работающее в филиале банка, записями клиентов, или это приложение для мобильного банкинга? Или, возможно, это веб-приложение для VIP-клиентов?

Вопрос в том, что является источником истины для каждого элемента данных в вашей системе? Когда в адресе электронной почты клиента существует несоответствие в разных приложениях и хранилищах данных, какой вариант является правильным?

Что, если я скажу вам, что это *не* приложение? И здесь мы подходим к объяснению названия этого раздела: *источником истины для всех данных является хранилище / журнал событий*.

Вот что стоит за термином *порождение событий*. Основной шаблон порождения событий состоит в том, что все источники сеансов записи данных создаются *только* в хранилище событий, а во всех других хранилищах просто хранятся проекции или снимки состояния, полученные из событий, сохраненных в журнале событий. Вспомните рис. 12.14; на нем было показано три интерфейса для службы Posts: контроллер чтения, контроллер записи и обработчик событий. Контроллер записи делал две вещи, сохранял новые посты в базе данных Posts и доставлял событие в журнал событий для любого другого микросервиса, которого это событие интересовало. В то время я намеренно не говорила о том, что происходит, когда запись успешно попадает в базу данных. Но тогда получается, что событие не было успешно доставлено, или мы не знаем, было ли оно доставлено успешно, в журнал событий. О боже, еще один проблемный случай.

В некотором смысле проблема заключается в том, что двухэтапный коммит не работает в этом гетерогенном и распределенном наборе компонентов. Я не могу создать транзакцию вокруг записи в базу данных и доставки события. Решение таково: вместо того чтобы координировать две операции, делайте только что-то одно. Рисунок 12.19 – это обновленная версия рис. 12.14, где показано, что:

- контроллер записи отвечает только за доставку событий в журнал событий. Снова посмотрев на рис. 12.14, вы видите, что контроллер записи отвечал за две вещи – хранение данных и доставку событий. Теперь он отвечает только за одну из этих функций, и это дает более надежную систему;

- обработчик событий отвечает только за чтение событий и их проецирование в локальное хранилище данных;
- контроллер чтения отвечает только за возврат результатов в зависимости от состояния в локальном хранилище данных.

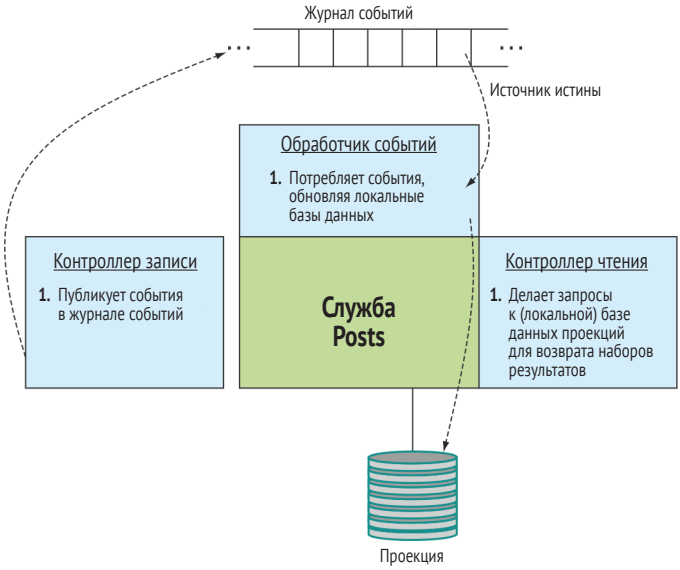


Рис. 12.19 ❖ Контроллер записи только выполняет запись в журнал событий, который теперь является источником истины. В локальной базе данных микросервиса хранятся только проекции, полученные из данных журнала событий; это реализовано в обработчике событий

Обратите внимание, что на этой диаграмме кажется, что некоторые данные в локальном хранилище обновляются обходным путем. Событие попадает в хранилище событий, а затем выходит из этого хранилища в хранилище проекций. Но вместо того чтобы считать это недостатком, я призываю вас рассматривать контроллер записи как контроллер, отделенный от службы Posts. Вот почему я нарисовала окошко отдельно от службы Posts; каждая часть сервиса делает только что-то одно, и мы достигаем желаемых результатов через композицию. То, что может показаться неэффективным, компенсируется соответствующим повышением надежности нашей системы.

Рисунок 12.20 – это обновленная версия рис. 12.12, отражающая событийно-ориентированный подход. Вы видите результирующее упрощение в топологии данных. В некоторых службах больше нет источников истины и локальных/проекционных баз данных, в то время как в других они есть; у всех сервисов есть только хранилища проекционных данных, и средства для поддержания этих хранилищ в актуальном состоянии также единообразны. Изящная конструкция более прочная и надежная.

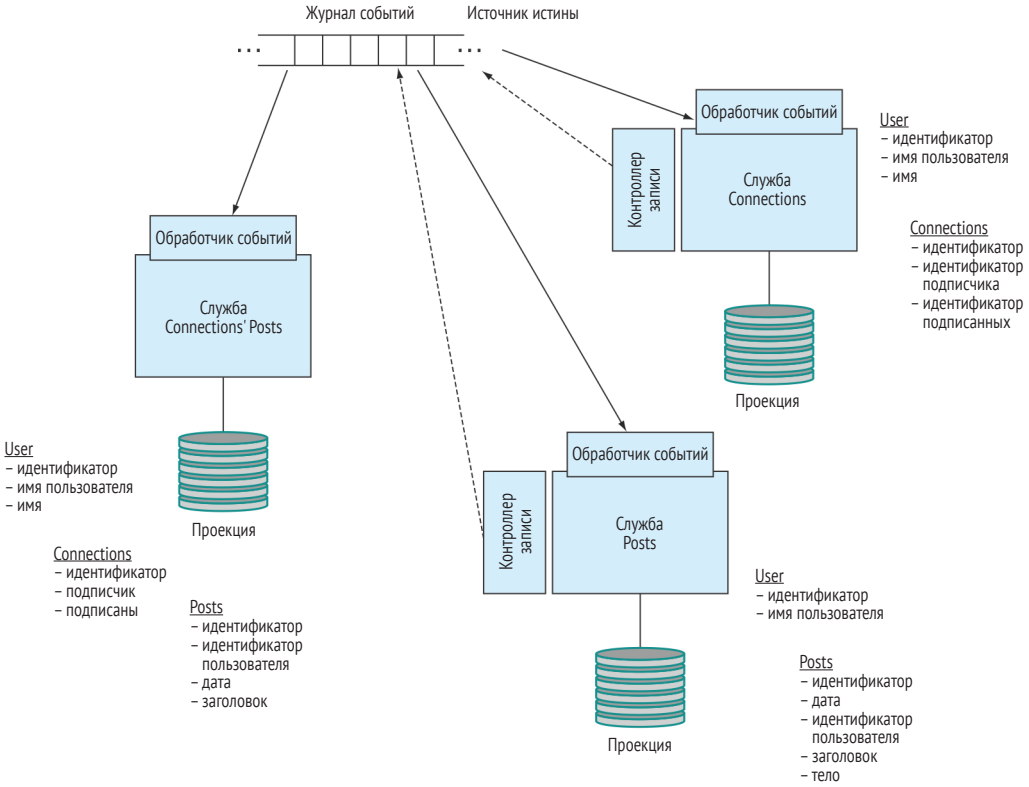


Рис. 12.20 ❖ Журнал событий является единственным источником истины. Все локальные хранилища микросервисов – это проекции, полученные из событий в журнале событий обработчиками событий.

Локальные хранилища микросервисов – всего лишь проекции

12.4.3. Давайте посмотрим, как это работает: реализация порождения событий

Давайте посмотрим на изменения в коде, которые переносят нашу более раннюю реализацию на базе журнала событий в вариант, использующий порождение событий. Скачайте этот тег из репозитория Git:

```
git checkout eventlog/0.0.2
```

Как уже говорилось ранее, мы упростили набор ролей, которые может выполнять любой данный сервис. Это отражено в табл. 12.2; непротиворечивость шаблонов в разных сервисах очевидна.

Поскольку это самый сложный из всех сервисов, давайте рассмотрим службу Posts более подробно.

Во-первых, вы увидите, что все функциональные возможности базы данных были объединены из пакетов `localstorage` и `sourceoftruth` в единый пакет, который я назвала *projectiontorage*; все данные теперь хранятся в хранилище проекций. Это не значит, что у вас может быть только одно хранилище проекций. Вы могли бы, например, проецировать события в реляционную базу данных для поддерж-

ки реляционных запросов и проецировать события в графовую базу данных для поддержки графовых запросов. Акцент в этом рефакторе заключается в том, что теперь существует только управление проекционными данными.

Таблица 12.2. Роли служб в нашей архитектуре программного обеспечения, основанной на событиях

Роль	Служба Connections	Служба Posts	Служба Connections' Posts
Производитель записи и событий – реализации конечных точек HTTP обновлять базы данных истины и доставлять события в Kafka	Генерирует пользовательское событие каждый раз, когда пользователь создается, обновляется или удаляется. Генерирует событие подключения каждый раз, когда соединение создано или удалено	Создает событие сообщения в любое время, когда создается сообщение	
Обработчик событий – подписывается на события и соответственно обновляет базы данных локального хранилища	Вызывается каждый раз, когда в журнале событий появляется событие пользователя или подключения	Вызывается в любое время, когда сообщение или пользовательское событие появляется в журнале событий	Вызывается каждый раз, когда в журнале событий появляется событие пользователя, подключения или публикации
База данных проекции – выделенная база данных для хранения всех данных ограниченного контекста сервиса	Хранит данные пользователя и соединения	Хранит подмножество пользовательских данных	Хранит подмножества данных пользователя, соединения и публикации
Чтение – реализации конечных точек HTTP, которые обслуживают доменные объекты для обслуживания	Пользователи. Соединения	Сообщения	Посты, сделанные пользователями, сопровождаемые человеком

Во-вторых, контроллер чтения практически не изменился, за исключением рефакторинга хранилища данных, который я только что описала.

В-третьих, в листинге 12.4 видно, что контроллер записи больше не выполняет запись в локальное хранилище, а только отправляет событие создания или обновления в журнал событий. Вы также заметите, что я переместила генерацию идентификатора из базы данных, потому что хочу назначить идентификатор при создании события, а не при создании проекции.

Листинг 12.4 ❖ Метод из PostsWriteController.java

```
@RequestMapping(method = RequestMethod.POST, value="/posts")
public void newPost(@RequestBody PostApi newPost,
                    HttpServletResponse response) {

    logger.info("Have a new post with title " + newPost.getTitle());

    Long id = idManager.nextId();
    User user = userRepository.findByUsername(newPost.getUsername());
```

```
    if (user != null) {
        // Отправка события нового поста;
        PostEvent postEvent
          = new PostEvent("created", id, new Date(), user.getId(),
                          newPost.getTitle(), newPost.getBody());
        kafkaTemplate.send("post", postEvent);
    } else
        logger.info("Something went awry with creating a new Post - user with username "
            + newPost.getUsername() + " is not known");
}
```

И наконец, в обработчик событий добавлен новый метод обработки событий поста. По сути, он содержит логику, которую вы удалили из исходного контроллера записи; в следующем листинге вы сохраняете данные поста в хранилище проекций.

Листинг 12.5 ❖ Метод из EventHandler.java

```
@KafkaListener(topics=»post»,
               groupId = "postsconsumer",
               containerFactory = "kafkaListenerContainerFactory")
public void listenForPost(PostEvent postEvent) {

    logger.info("PostEvent Handler processing - event: "
                + postEvent.getEventType());

    if (postEvent.getEventType().equals("created")) {
        Optional<Post> opt = postRepository.findById(postEvent.getId());
        if (!opt.isPresent()){
            logger.info("Creating a new post in the cache with title "
                        + postEvent.getTitle());
            Post post = new Post(postEvent.getId(),
                                 postEvent.getDate(),
                                 postEvent.getUserId(),
                                 postEvent.getTitle(),
                                 postEvent.getBody());
            postRepository.save(post);
        } else
            logger.info("Did not create already cached post with id "
                        + existingPost.getId());
    }
}
```

Аналогичные изменения были внесены в службу Connections, но для службы Connections' Posts никаких изменений не потребовалось, поскольку у нее не было данных об источнике записи или контроллера записи, который нужно было запустить. После изучения кода я рекомендую вам запустить скрипт `deployApps.sh` для обновления развертывания до этой последней реализации. Выполните несколько примеров, чтобы увидеть, как это работает (создайте пользователя, подключения и посты, удалите подключение), просмотрите журналы и прочитайте результирующее состояние, используя контроллеры чтения каждой службы:

```
curl $(minikube service --url connections-svc)/users
curl $(minikube service --url connections-svc)/connections
```

```
curl $(minikube service --url posts-svc)/posts
curl $(minikube service --url connections-svc)/connectionsPosts/<username>
```

Хотя подход на базе порождения событий, возможно, отличается от того, как вы рассматриваете реализацию служб, таких как примеры, приведенные в этой книге, он является важной частью архитектуры микросервисов. Это шаблон, который позволяет вам избавиться от проекционной базы данных и всех ее резервных копий, а также восстановить состояние программного обеспечения путем воспроизведения логов, чтобы сгенерировать их заново. Это позволяет микросервисам, которые ранее могли быть подчинены иерархии глубоких зависимостей, функционировать автономно, даже в случае сбоев в сети или других инфраструктурах. Это позволяет командам развивать свои сервисы без необходимости идти в ногу с другими сервисами. Этот шаблон является важным инструментом для создания программного обеспечения, устойчивого к изменениям.

12.5. Это лишь поверхностное знакомство

Несмотря на то что мы рассмотрели в этой главе множество вопросов, есть еще много вещей, о которых нужно говорить, когда речь заходит о данных в облачной среде:

- мы не говорили о разделении журнала событий, которое необходимо для масштабирования. Когда у вас есть 10 млн пользователей, вам нужно будет объединить их в подмножества. Вы будете делать это, группируя пользовательские события по первой букве их фамилии, или будете использовать какой-либо иной параметр?
- мы мало говорили об упорядочении событий, что важно для использования журнала событий, чтобы получить проекции состояния. Технологии журнала событий реализуют сложные алгоритмы для обеспечения правильного упорядочения и могут иногда сообщать производителю, что он не может записать событие из-за несоблюдения определенных ограничений упорядочения; производитель событий должен это учесть;
- мы не говорили о том, как развивать схемы событий или методы, такие как разрешение схем (поддерживаемое Apache Avro), которые позволяют старым событиям олицетворять новые;
- мы не говорили о практике создания периодических снапшотов хранилищ проекционных данных, чтобы, если вам нужно будет снова создать хранилище проекций из журнала, вам не нужно будет возвращаться к началу времени.

Тема данных в облачной среде настолько увлекательна, что заслуживает отдельной книги, и я хочу порекомендовать вам кое-что. В 2017 году Мартин Клеппманн, специалист в области информатики и один из создателей Apache Kafka, опубликовал книгу *Designing Data-Intensive Applications* (O'Reilly Media). Я всячески поддерживаю этот проект!

Резюме

- Когда вы предоставляете микросервису базу данных для хранения данных, необходимых для выполнения своей работы, он получает значительную выгоду в плане автономности. В результате ваша система как единое целое будет более устойчивой.

- Хотя во многих случаях это гораздо лучше, чем ничего, использование кеширования для заполнения этой локальной базы данных сопряжено с трудностями. Кеширование – это не шаблон, который хорошо работает для часто меняющихся данных.
- Упреждающая передача изменений данных в эти локальные хранилища данных с помощью событий – гораздо лучший подход.
- В качестве основы этого метода используется знакомый шаблон «издатель/подписчик», хотя сущности, которые мы производим и потребляем, являются событиями, а не сообщениями.
- Создание журнала событий в качестве единственного источника истины для данных со всеми локальными сервисными базами данных, содержащими только проекции, обеспечивает согласованность таким образом, чтобы это работало в сильно распределенной, постоянно меняющейся среде, в которой работает наше программное обеспечение для облачной среды.

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Корнелия Дэвис

Шаблоны проектирования для облачной среды

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Беликов Д. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 31,53. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**