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.

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.

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**).

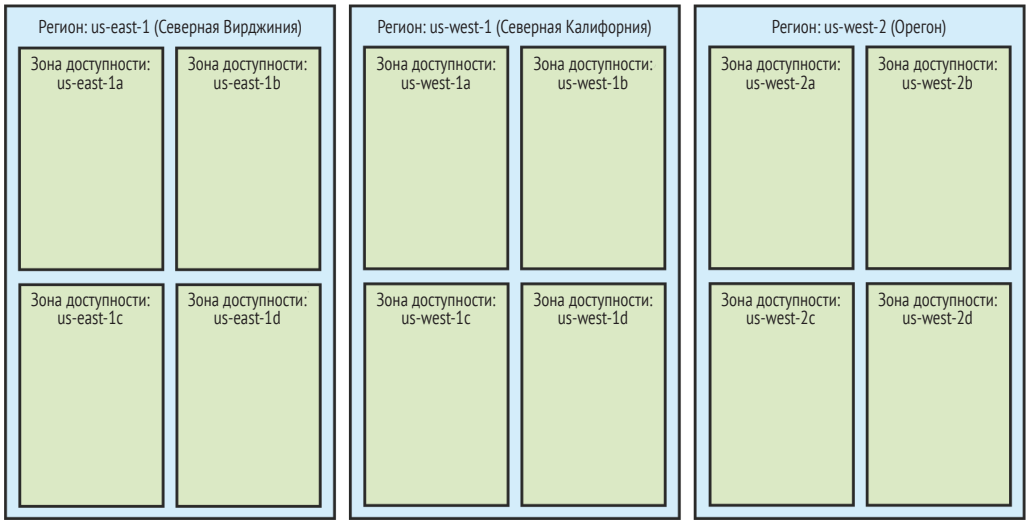


Рис. 1.1 ❖ AWS делит предлагаемые сервисы на регионы и зоны доступности. Регионы отображаются в географических областях, а зоны доступности обеспечивают дополнительную избыточность и изоляцию в пределах одного региона

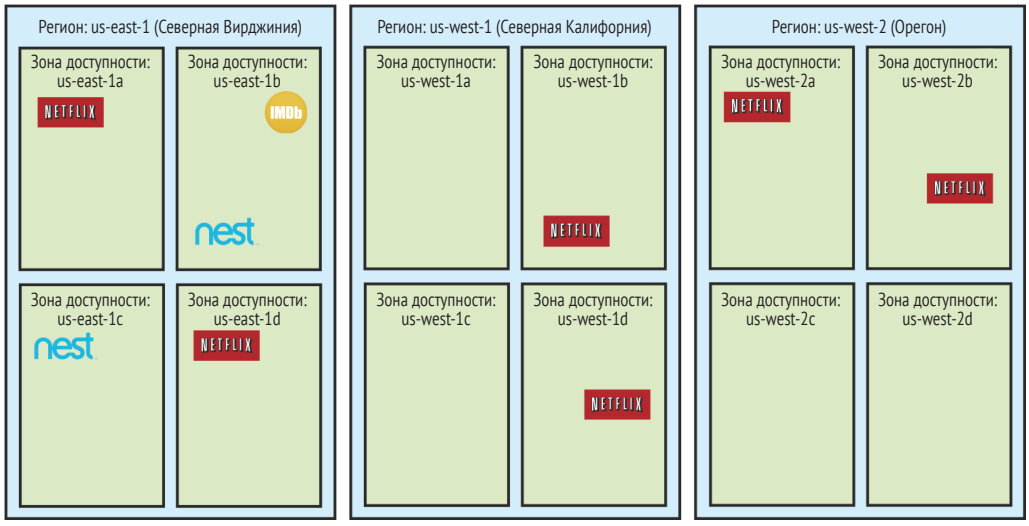


Рис. 1.2 ❖ Приложения, развернутые на AWS, могут быть развернуты в одной зоне доступности (IMDb) или в нескольких (Nest), но только в одном регионе, или в нескольких зонах доступности и нескольких регионах (Netflix), что обеспечивает различные профили устойчивости

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

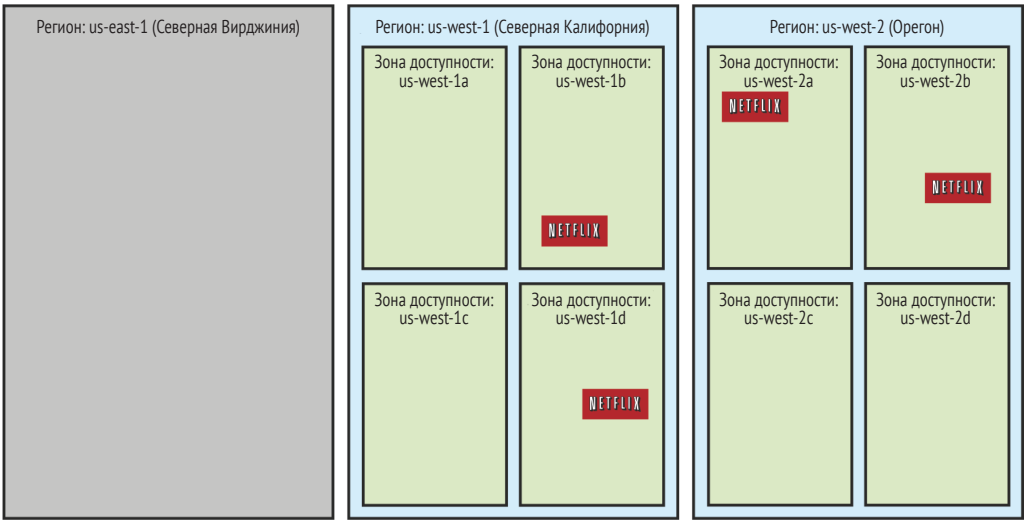


Рис. 1.3 ❖ Если приложения правильно спроектированы и развернуты, программное обеспечение может пережить даже обширный сбой, например сбой всего региона

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.

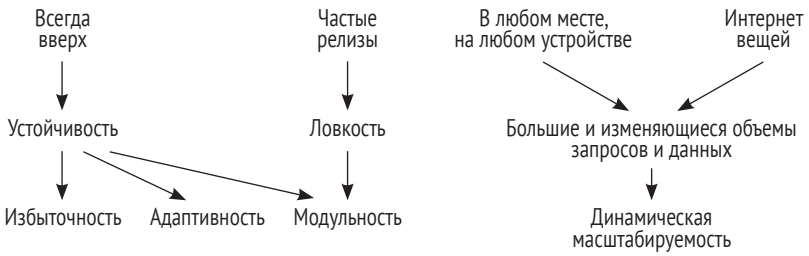


Рис. 1.4 ❖ Требования пользователей к программному обеспечению ведут разработку в направлении принципов архитектуры облачной среды

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.

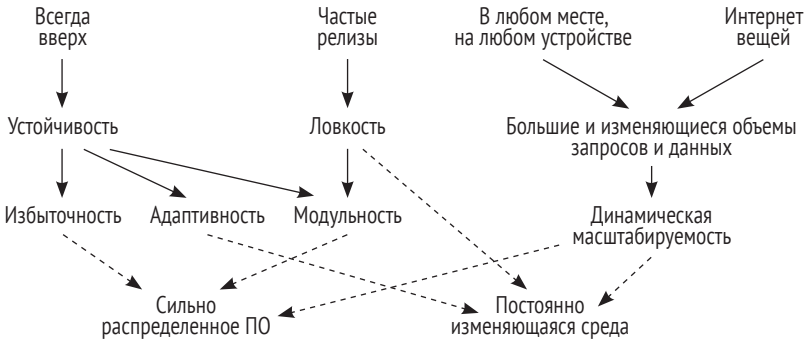


Рис. 1.5 ❖ Архитектурные и управленческие принципы приводят к основным характеристикам программного обеспечения для облачной среды: оно сильно распределено и должно работать в постоянно меняющейся среде, даже когда постоянно развивается

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.

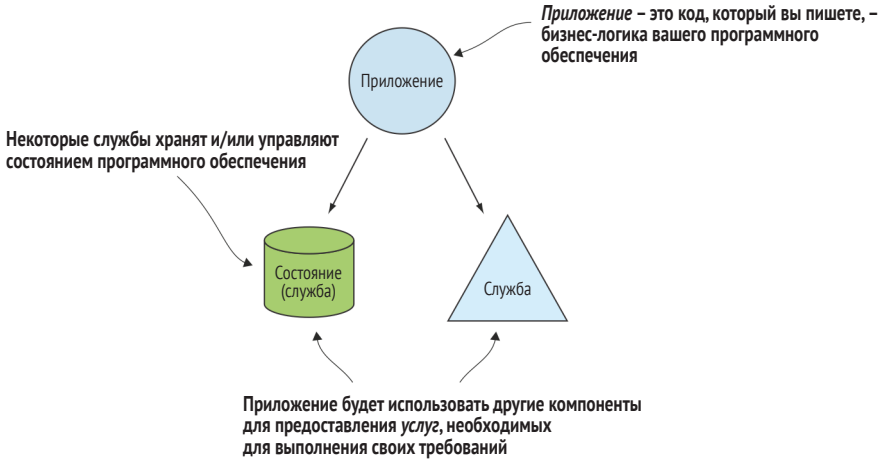


Рис. 1.6 ❖ Знакомые элементы базовой архитектуры программного обеспечения

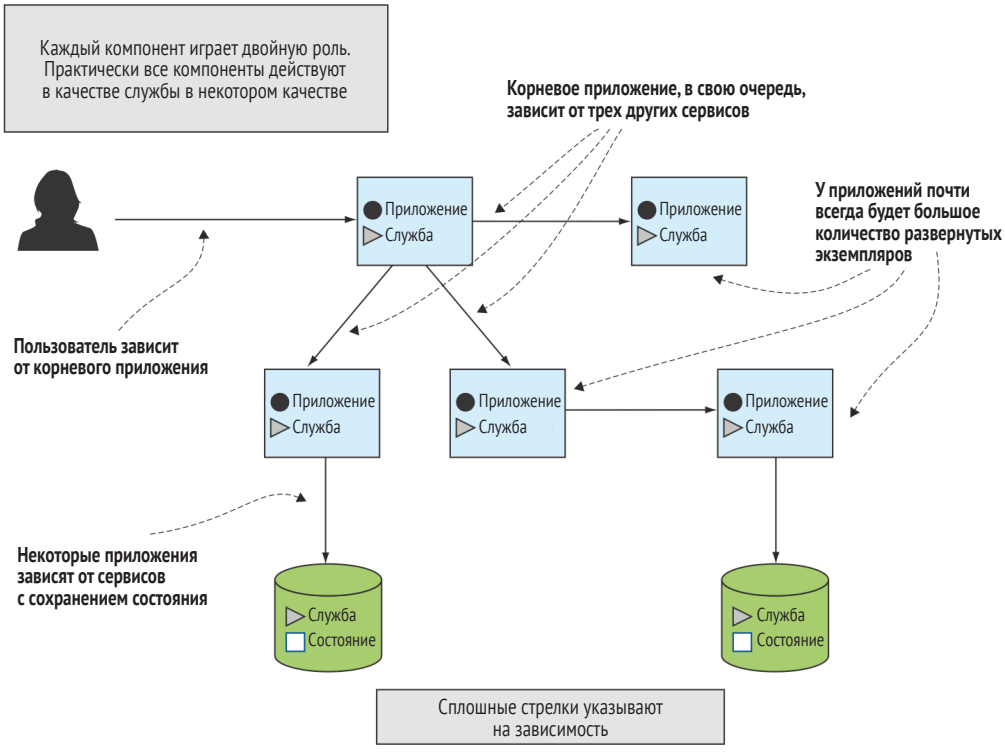


Рис. 1.7 ❖ Программное обеспечение для облачной среды использует знакомые концепции и обеспечивает экстремальное распространение с избыточностью повсюду и постоянные изменения

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.

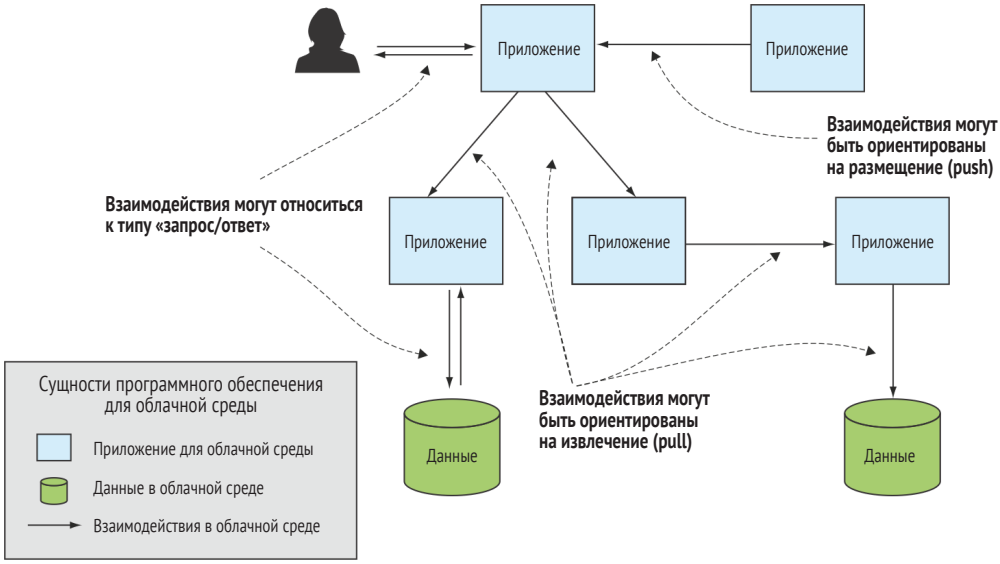


Рис. 1.8 ❖ Ключевые сущности в модели программного обеспечения для облачной среды: приложения, данные и взаимодействия

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.

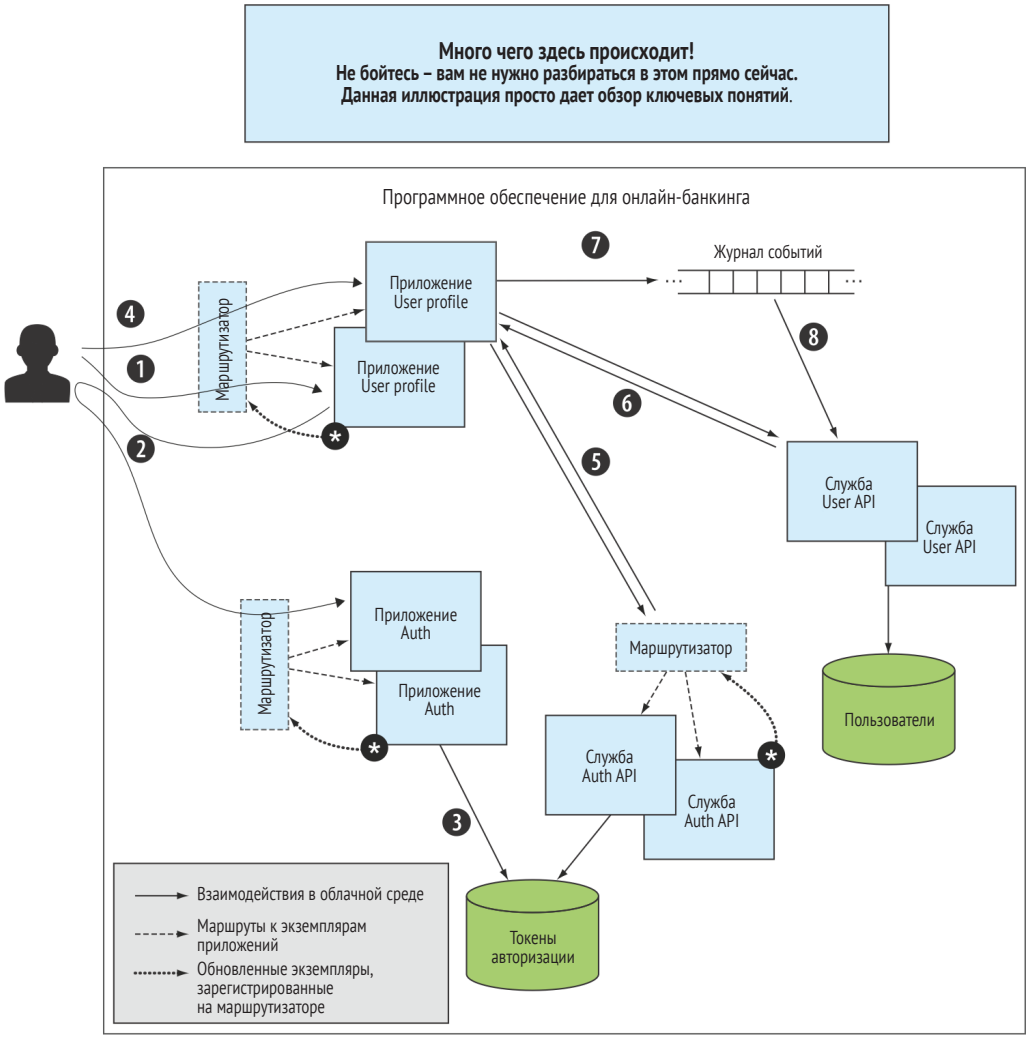


Рис. 1.9 ❖ Программное обеспечение для онлайн-банкинга представляет собой набор приложений и услуг передачи данных. Участвует множество типов протоколов взаимодействия

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.

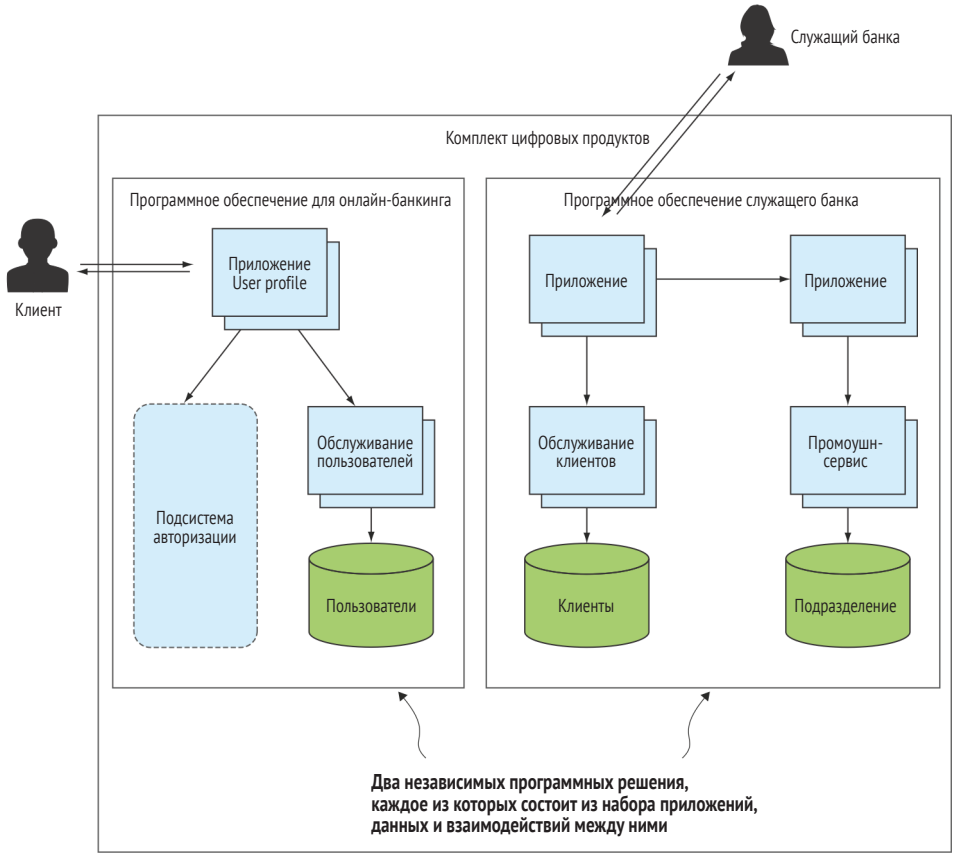


Рис. 1.10 ❖ То, что кажется пользователю единичным опытом взаимодействия с банком Волшебника, реализуется независимо разработанными и управляемыми программными ресурсами

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.

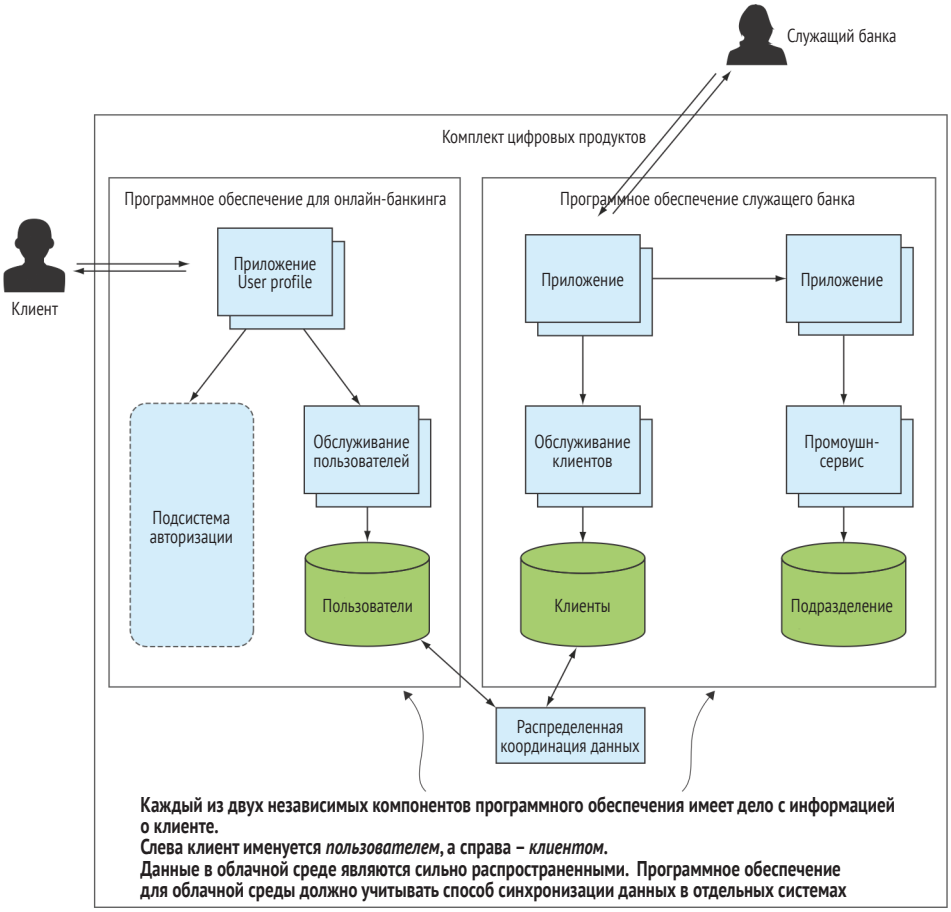


Рис. 1.11 ❖ Декомпозированная и слабо связанная структура данных требует методов для согласованного управления данными

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.

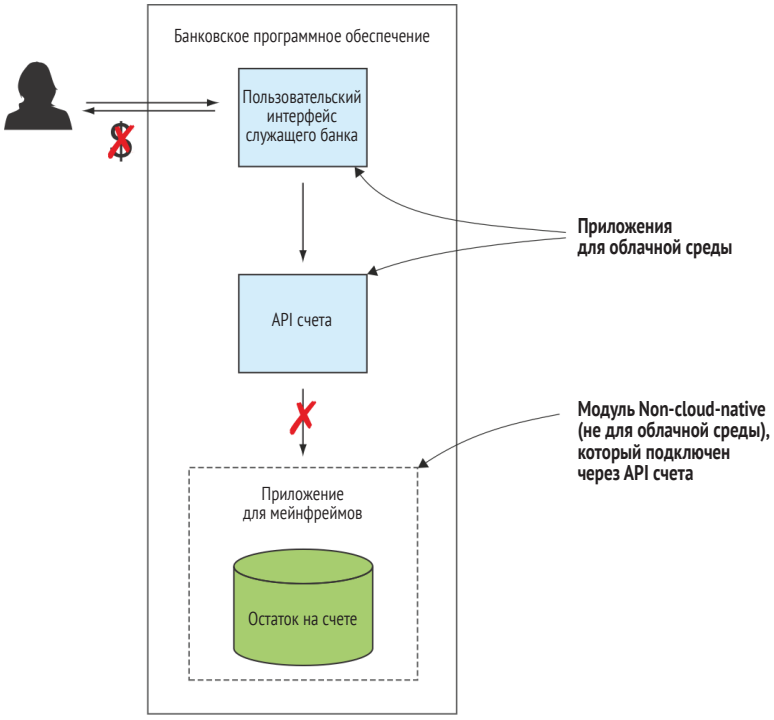


Рис. 1.12 ❖ Не рекомендуется распределять средства без доступа к источнику записи

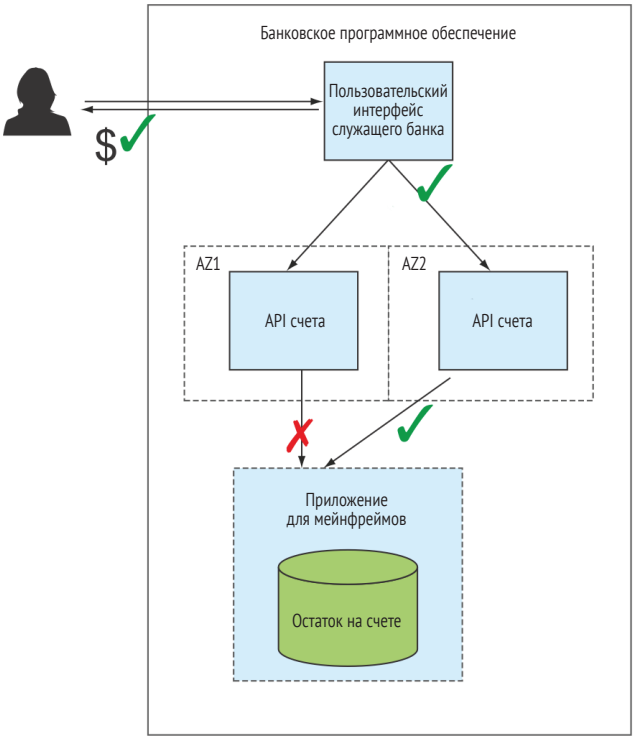
А теперь давайте применим несколько облачных шаблонов к частям этой системы. Например, если вы развернете множество экземпляров каждого микросервиса в многочисленных зонах доступности, сетевой раздел в одной зоне попрежнему разрешает доступ к данным мэйнфрейма через экземпляры сервисов, развернутые в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