Оглавление

ГЛАВА 1

Навстречу архитектуре микросервисов

Цель книги — помочь вам создать рабочую архитектуру микросервисов. Здесь вы найдете надежные и проверенные советы по созданию ПО. Они основаны на опыте реальных практикующих специалистов, как на примерах успешных реализаций, так и на тех, которые могли быть реализованы лучше. Мы превратили эти уроки в модель, которая, как мы надеемся, поможет вам быстрее освоиться с вашей системой.

В последнее время популярность создания программного обеспечения в виде микросервисов резко возросла. В начале 2010-х годов термин *«микросервисы»* был введен для описания нового стиля архитектуры ПО. Приложения, реализованные в этом стиле, создаются с помощью небольших независимых компонентов, которые работают вместе. С тех пор темпы внедрения стиля микросервисов резко возросли. Стартапы, крупные и мелкие компании изучают и внедряют микросервисные архитектуры. Растущая экосистема инструментов, сервисов и решений в данной области свидетельствует о ее широкой популярности. Когда мы писали эту книгу, специалисты Allied Market Research (https://oreil.ly/cugsz) предсказали, что мировой рынок микросервисных архитектур вырастет до 8,07 млрд долларов США в 2026 году с нынешних 2,07 млрд долларов США. Такие цифры указывают на большой интерес к микросервисам и их широкое внедрение.

Для многих реализация микросервисных архитектур оказалась нелегкой задачей. Дело в том, что внедрить систему микросервисов непросто. Заставить множество независимых частей работать вместе — куда сложнее, чем может показаться. Затраты на управление, сопровождение, поддержку и тестирование в такой системе суммируются. В крупных масштабах эти затраты могут стать непомерно высокими. Если вы не будете осторожны, то сложность управления системой может привести к тому, что микросервисы окажутся плохой затеей.

Но преимущества микросервисов оправдывают риски. Хорошо реализованные микросервисы позволяют развивать программное обеспечение быстрее и безопаснее, если рассматривать достаточно масштабную систему. Скорость и безопасность изменений дают бóльшую гибкость, что способствует процветанию и вашего бизнеса, и вашей организации.

Чтобы раскрыть все эти преимущества, нужно создать правильную архитектуру для поддержки сервисов. Важно снизить системные затраты, не снижая ценности независимых сервисов. Чтобы создать такую архитектуру, нужно заранее принять важные решения, касающиеся методов, процессов, команд, технологий и инструментов. Все это должно работать слаженно, чтобы сформировать новое оптимизированное целое.

Хороший способ построить подобную систему — эволюционное развитие. Для начала можно принять несколько простых решений и учиться и расти по ходу дела. Фактически большинство пионеров в этой области пришли к микросервисам благодаря многочисленным экспериментам. Они не ставили перед собой цель создать приложение на основе микросервисов. Они получили их в результате непрерывного процесса оптимизации и совершенствования.

Чтобы начать с нуля и двигаться к цели шаг за шагом, нужно время. Но у вас есть ценное преимущество — возможность использовать опыт ваших предшественников, чтобы помочь себе быстрее выстроить свою систему. Начните ее создание с подбора шаблонов, методов и инструментов, сочетание которых привело к успеху. Затем оптимизируйте систему в соответствии с целями и ограничениями вашей организации.

В этой книге мы описали решения, формирующие прочную основу для создания микросервисов. Но прежде чем углубиться в детали, рассмотрим важный вопрос: что именно подразумевается под «микросервисами»?

Что такое микросервисы

Не существует общепринятого официального, канонического определения *микросервисов*. Хорошей отправной точкой является статья Джеймса Льюиса и Мартина Фаулера о микросервисах 2014 года (https://oreil.ly/guhCP). Авторы описывают микросервисы как:

> «подход к разработке единого приложения в виде набора небольших сервисов, работающих в отдельных процессах и взаимодействующих с применением упрощенных механизмов. [...] построенных вокруг бизнес-потребностей и развертываемых независимо с помощью полностью автоматизированного механизма».

Суть статьи Льюиса и Фаулера — в описании набора из девяти характеристик, которыми обладают микросервисы. Список начинается с основной микросервисной характеристики: *компонентное представление с помощью сервисов*, что означает разбиение приложения на более мелкие сервисы. От него авторы переходят к широкому спектру возможностей. Они обосновывают необходимость организационного и управленческого проектирования с учетом особенностей *организации, связанных с бизнес-потребностями* и *децентрализованным управлением*. Авторы намекают на DevOps и методы гибкой разработки, когда внедряют *автоматизацию инфраструктуры* и *продукты, а не проекты*. В дополнение авторы определяют несколько ключевых принципов архитектуры, таких как *умные конечные точки и тупые каналы* (smart endpoints and dumb pipes), *проектирование с учетом сбоев* и *эволюционное проектирование*.

Каждая из этих характеристик заслуживает внимания, и мы рекомендуем вам прочитать их статью, если вы еще этого не сделали. Вместе эти характеристики образуют целостное решение с очень большим набором задач, куда включаются технологии, инфраструктура, инжиниринг, внедрение, управление, структура команды и культура.

Для сравнения предлагаем еще одно определение микросервисов из книги *Microservice Architecture*, написанной Иракли Надареишвили, Ронни Митрой, Мэттом Макларти и Майком Амундсеном (O'Reilly) (https://learning.oreilly.com/library/view/microservice-architecture/9781491956328).

> «*Микросервис* — это независимо развертываемый компонент с ограниченной областью действия, поддерживающий взаимодействия посредством обмена сообщениями. *Микросервисная архитектура* — это стиль проектирования высокоавтоматизированных, эволюционирующих программных систем, состоящих из микросервисов, ориентированных на потребности».

Оно похоже на определение Льюиса и Фаулера, но здесь особое внимание уделяется ограниченным областям, функциональной совместимости и взаимодействиям посредством сообщений. В нем также проводится различие между микросервисами и архитектурой, которая их обеспечивает.

Это всего лишь два примера из множества определений микросервисов. Большинство определений в целом схожи, но каждое из них немного отличается по своей направленности.

В мире технологий названия и определения важны, поскольку позволяют нам простым языком объяснить сложные понятия. В этом случае слово «микросервисы» помогает нам описать *стиль* архитектуры ПО, обладающий тремя основными конструктивными чертами.

1. Архитектура приложения в основном состоит из машинно-вызываемых «сервисов», доступных в сети.

2. Размеры (или границы) сервисов — важный критерий проектирования, на который влияют факторы времени выполнения, времени разработки и персонала.
3. Программная система, организация и способ работы оптимизированы для достижения цели.

Это довольно общий набор конструктивных черт. Например, в нем не упоминаются организационные стили, конкретные инструменты или архитектурные принципы, которые следует использовать. Не определены и какие-либо формальные шаблоны или практики. Но он дает нам достаточно характеристик, чтобы идентифицировать систему микросервисов при встрече с ней.

Правда в том, что практически любую систему на базе API можно назвать микросервисной архитектурой, если очень постараться. Основное внимание должно быть сосредоточено на цели вашей системы. Мы думаем, что вопрос о том, зачем создаются микросервисы, гораздо важнее, чем вопрос о том, что они собой представляют. Несмотря на то что микросервисы обладают множеством потенциальных преимуществ, мы считаем, что лучшая причина создавать программное обеспечение с применением этого подхода — снижение затрат на координацию (взаимодействие).

Сокращение затрат на координацию

Компании по всему миру с успехом внедряют микросервисные архитектуры. Почти все практикующие специалисты, с которыми мы беседовали, сообщали об увеличении скорости доставки программного обеспечения. Мы считаем, что улучшение обусловлено фундаментальным преимуществом микросервисов: снижением затрат на координацию.

Следует отметить, что существует множество способов увеличить скорость разработки ПО, и организация программного обеспечения в виде микросервисов — лишь один из вариантов. Например, можно быстро создать систему, выполнив работу поверхностно и накопив «технический долг» (https://oreil.ly/PBMHU), с которым предполагается разобраться позже. Или же можно меньше внимания уделить стабильности и безопасности и просто выпустить свой продукт в мир. В отдельных ситуациях и для некоторых предприятий это разумные подходы.

Но в системах, разрабатываемых, в частности, для финансового, медицинского и государственного секторов, непозволительно ради повышения скорости идти на компромисс с безопасностью. И все же рынок требует от этих отраслей более высокой скорости, как и от любых других. Именно здесь микросервисы могут проявить себя во всей красе. Они предлагают архитектурный подход, позволяющий увеличить скорость разработки без ущерба для безопасности. И дают возможность делать это в широком масштабе.

Проблема затрат на координацию

Создание сложного программного обеспечения — тяжелая работа. В кино блестящий программист может за пару бессонных ночей героически создать продукт, способный изменить мир. В реальной жизни для получения качественного результата требуется много людей и уйма времени. Несколько команд, работающих над сложным проектом, обычно реализуют разные части общей системы, следуя своим планам и совершая шаги независимо. Периодически эти части необходимо интегрировать, чтобы устранить зависимости, для чего независимые команды должны координировать свою работу (рис. 1.1).

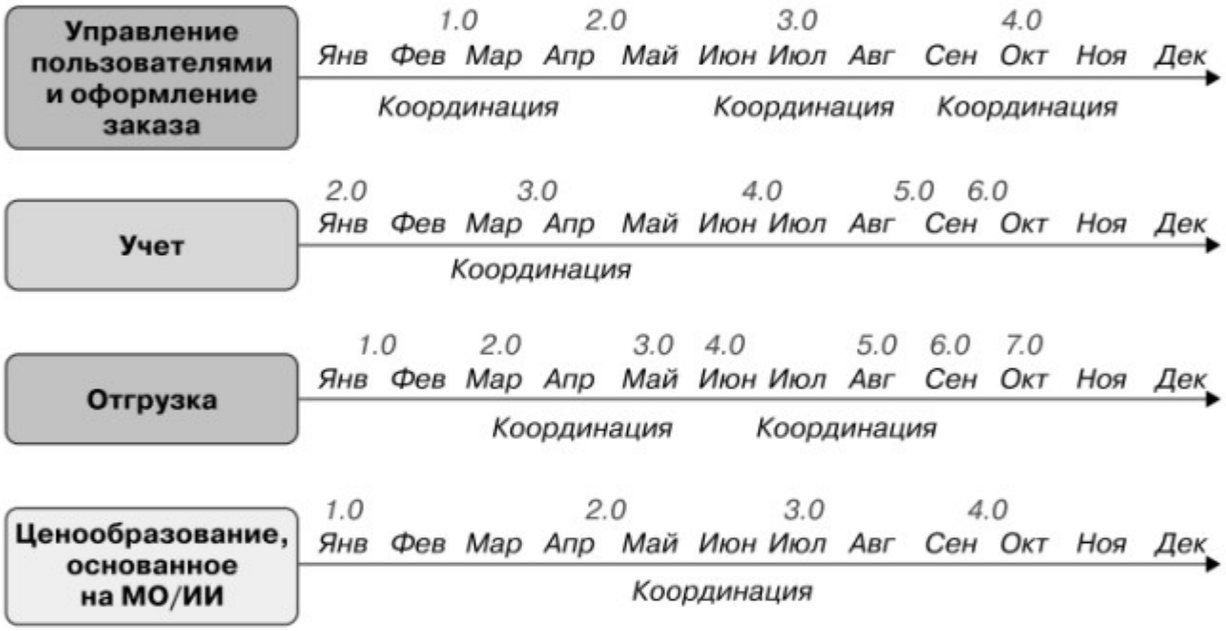

Рис. 1.1. Примерная временная шкала сложного проекта с координационными точками соприкосновения

Представьте, что Джейн — руководитель группы, отвечающей за бухгалтерский учет. Ее команда только что закончила спринт и зависит от компонента, который разрабатывается командой, отвечающей за модуль отгрузки, во главе с Тайроном. Поскольку их проектные планы независимы, вполне возможно, что его команда на самом деле не закончила реализацию необходимого компонента. На данный момент у Джейн есть два варианта: либо дожидаться готовности компонента (отдавая приоритет безопасности и жертвуя скоростью) и провести надлежащее интеграционное тестирование, либо положиться на согласованный контракт взаимодействий между ее компонентом и компонентом Тайрона, предполагая, что его команда выпустит обновленную версию компонента точно в срок. Выбрав второй вариант, Джейн сможет работать без остановки, увеличивая скорость работы своей команды, но потенциально

ставя под угрозу общую безопасность системы, поскольку интеграционное тестирование не проводилось на самом раннем возможном этапе и было сделано предположение об «удачном пути»[1].

Любой руководитель в организации со множеством задействованных команд регулярно сталкивается с этим выбором: игнорировать затраты на координацию и сохранить динамику или признать необходимость координации и замедлиться. Как правило, мы выбираем тот или иной вариант, интуитивно оценивая соотношение риска и выгоды. Но в целом в достаточно сложной системе, когда такой выбор необходимо делать часто, возникает очень заметное противоречие между скоростью и безопасностью.

Поскольку нас беспокоят затраты на координацию, то было бы неплохо иметь систему, специально разработанную так, чтобы *минимизировать* их. Чтобы вместо выбора того или иного пути команды вообще не сталкивались с необходимостью выбора. Хотелось бы иметь проект, требующий минимум координации между автономными командами, работающими над небольшими изолированными задачами. Именно это предлагает микросервисная архитектура.

Важно понимать, что работа по созданию хороших микросервисов направлена на минимизацию координации. Создавать сложные распределенные системы, такие как микросервисные архитектуры, непросто, и в моменты сомнений мы всегда должны спрашивать себя: «Решение, с которым я сталкиваюсь, снизит затраты на координацию моих команд или нет?» Правильный ответ будет гораздо очевиднее, если мы рассмотрим решения с точки зрения затрат на координацию.

В конечном счете микросервисы стали популярными, потому что помогают бизнесу добиться успеха. Современные организации испытывают невероятное давление, которое требует чаще и быстрее адаптироваться, меняться и совершенствоваться. Важно инвестировать в технологическую архитектуру, целенаправленно разработанную для изменения скорости и безопасности в масштабах крупной организации. Микросервисы позволяют компаниям, работающим в сложных областях, быть такими же гибкими, как более простые и небольшие компании, и продолжать использовать мощь и возможности своего фактического размера. Это невероятно привлекательно, и рост количества их внедрений доказывает это, однако преимущества не даются бесплатно. Требуется много и сосредоточенно работать и принимать важные решения, чтобы создать микросервисную архитектуру, которая может раскрыть эту ценность.

---

[1] От англ. happy path — сценарий по умолчанию, в котором отсутствуют исключительные ситуации или ошибки. — *Примеч. пер.*

Сложные компромиссы

Одно из самых больших препятствий, с которыми сталкиваются начинающие разработчики микросервисов, — огромная разветвленность системы. На первых порах можно сосредоточиться на создании небольших ограниченных сервисов. Но очень скоро придется создавать правильную инфраструктуру, модели данных, фреймворки, командные модели и процессы для их поддержки. Это большая область, которую нужно охватить, и решение всего спектра вопросов может привести к особым проблемам. Ниже описаны три серьезные проблемы проектирования, с которыми обычно сталкиваются архитекторы и инженеры микросервисов.

- *Длительные циклы обратной связи*. Одна из больших проблем — сложность оценки эффективности решений в системе микросервисов. Решения, принимаемые сегодня, могут повлечь проблемы, которые проявятся гораздо позже. Например, на старте вы можете решить использовать общую библиотеку, чтобы упростить реализацию взаимодействий между сервисами. Но со временем может выясниться, что поддержание ее в актуальном состоянии во всех микросервисах и командах оказывается огромной проблемой. Суть в том, что трудно оценить, как повлияет принимаемое вами решение, пока не возникнут сложности. Это затрудняет оценку и выбор вариантов.
- *Слишком много движущихся частей*. Система микросервисов — это сложная адаптивная система. Каждая часть системы некоторым образом влияет на другие ее части. Когда все они собираются воедино, возникает новое, непредсказуемое поведение системы. Если вы когда-либо внедряли новый инструмент или процесс в организации, то, вероятно, сталкивались с этим. Одни команды реагируют на новые стимулы и немедленно меняются, другим нужны помощь и поддержка, чтобы адаптироваться. Однако, несмотря ни на что, почти всегда приходится сталкиваться с последствиями работы людей и принятых ими решений. Например, команды, внедряющие инструменты контейнеризации Docker, неизбежно адаптируют свой цикл разработки и выпуска, приводя его в соответствие с принятой моделью развертывания контейнеров. Иногда эти последствия можно предусмотреть, но часто приходится иметь дело с непредвиденными последствиями вносимых изменений. Это затрудняет проектирование систем микросервисов. Трудно предсказать конкретные последствия вносимых изменений, поэтому, внедряя новую архитектурную модель, мы рискуем принести больше вреда, чем пользы.
- *Аналитический паралич*. Сложность проектируемой системы вкупе с длительными циклами обратной связи обнажает трудности построения микро-

сервисной архитектуры. Решения, которые приходится принимать, имеют большое влияние и трудно поддаются оценке. Это может привести к бесконечным спекуляциям, обсуждениям и оценке архитектурных решений из-за страха создать неправильную систему. Вместо того чтобы создать систему, способную удовлетворить требования бизнеса, мы оказываемся в подвешенном состоянии, пытаясь смоделировать бесконечные последствия нашего выбора. Это состояние широко известно как *аналитический паралич*. Мешает также наличие в Интернете множества страшных историй, советов в стиле «наклейки на бампер» и противоречивых рекомендаций по созданию микросервисных архитектур.

В конечном счете реальная сложность построения микросервисной архитектуры заключается в организации работы с большой, сложной системой, охватывающей обширную область. Но спешим вас утешить: это не уникальная проблема. В книге мы представим и используем набор практик и шаблонов, разработанных для микросервисных архитектур. Мы также покажем и реализуем инструменты, помогающие внедрять эти практики и делающие работу по созданию системы микросервисов проще, безопаснее, дешевле и быстрее.

Обучение на практике

На данный момент мы установили, что микросервисы могут помочь быстрее доставлять программное обеспечение без ущерба для безопасности. Но мы также определили, что путь к хорошей микросервисной архитектуре труден и сопряжен с необходимостью принятия сложных решений. Многие из успешных разработчиков микросервисов, с которыми мы беседовали, создавали свои системы путем постоянных повторений и улучшений. Прежде чем понять, как создать работающую систему, они часто создавали неудачные архитектуры.

Будь у вас неограниченное время, вы могли бы создать отличную микросервисную архитектуру исключительно путем экспериментов, используя разные организационные модели, пробуя все методологии и создавая микросервисы различных размеров. Пока есть возможность оценивать результаты, можно продолжать совершенствовать систему. Проведя достаточное количество экспериментов, вы создадите систему, удовлетворяющую именно вашим требованиям, а также получите большой опыт создания микросервисных систем.

Однако, скорее всего, у вас ограниченный запас времени. Как же вы будете накапливать опыт, необходимый для создания более совершенных микросервисов?

Чтобы помочь решить эту проблему, мы разработали образцовую модель микросервисов. В процессе ее создания мы принимали решения о структуре команды, процессах, архитектуре, инфраструктуре, инструментах и технологиях. Мы постарались охватить широкий спектр тематических областей. Наши решения базируются на опыте создания систем микросервисов для крупных организаций. Если вы последуете нашим рекомендациям, то к концу этой книги создадите простую систему микросервисов в облачной архитектуре.

В работе мы использовали вымышленную, значительно упрощенную систему бронирования авиабилетов. Наша простая система бронирования авиабилетов будет предоставлять две функции: предоставление информации о рейсах, доступной только для чтения, и бронирование мест.

Наша цель — помочь вам как можно быстрее создать свою первую систему микросервисов. По опыту, процесс создания реальной системы — лучший способ получить истинное представление о проделанной работе и ключевых решениях. Мы не ждем, что вы согласитесь со всеми нашими решениями. На самом деле подвергать сомнению решения, которые мы приняли за вас, — важная часть пути обучения! Надеемся, модель, которую мы создадим вместе, — лишь первая из многих ваших будущих систем микросервисов.

Модель приобретения навыков Дрейфуса

Начать обучение, следуя инструкциям, — проверенный и верный путь к приобретению опыта. Согласно статье FiveStage Model of Adult Skill Acquisition Стюарта и Хьюберта Дрейфусов (https://oreil.ly/vs3ao), первый этап включает в себя следование конкретным рекомендациям для получения знаний и наработки собственного опыта.

Модель микросервисов «От архитектуры до релиза»

Область применения микросервисных архитектур довольно широка. К сожалению, мы не можем рассмотреть все в одной книге, но постарались охватить тематические области, которые наиболее актуальны для системы микросервисов и больше всего влияют на успех. Перечислим, что мы будем рассматривать в модели микросервисов «От архитектуры до релиза».

- *Структура команды.* Мы приступим к работе в главе 2, начав с людей, которые должны участвовать в создании системы микросервисов. Раскроем проблемы эффективного построения команды и назовем главные факторы, влияющие на координацию микросервисов. Кроме того, представим коман-

ды, которые будем использовать в наших примерах, и инструмент Team Topologies («Топологии команд»), который поможет их создать.

- *Структура микросервисов*. После создания команд мы представим в главе 3 SEED(S)-процесс. Этот процесс проектирования позволяет создавать микросервисы, которые удовлетворяют потребности пользователей и потребителей и отличаются удобными интерфейсами и поведением. Затем, в главе 4, мы рассмотрим проблему определения правильных границ для наших микросервисов. Мы также представим некоторые важные концепции предметно-ориентированного проектирования и будем использовать процесс, называемый Event Storming, для выбора «правильного размера» наших сервисов.
- *Структура данных*. Данные — один из самых сложных аспектов проектирования микросервисов. В главе 5 мы посмотрим, какие свойства данных необходимо учитывать в системе микросервисов, представим концепцию независимости данных и заложим основу архитектуры данных для нашего проекта.
- *Облачная платформа*. Наша реализация микросервисов будет построена на облачной инфраструктуре. В главе 6 мы представим и реализуем принципы неизменяемой инфраструктуры и инфраструктуры как кода (infrastructure as code, IaC) в качестве основы для инфраструктуры микросервисов. Кроме того, мы представим облачную платформу AWS и создадим конвейер CI/CD на основе GitHub Actions. Затем, в главе 7, с помощью этого конвейера мы спроектируем и реализуем инфраструктуру микросервисов на базе AWS — она будет включать сеть, кластер Kubernetes и средство развертывания GitOps.
- *Разработка микросервисов*. Настроив инфраструктуру, мы погрузимся в задачи по разработке микросервисов. В главе 8 мы сначала опишем принципы и инструменты, необходимые для достижения успеха. Затем в главе 9 реализуем два независимых разнородных микросервиса для нашего приложения.
- *Релиз и внесение изменений*. В главе 10 мы соберем все вместе и развернем один из созданных нами микросервисов на облачной платформе. Для этого воспользуемся набором технологий, включая DockerHub, Kubernetes, Helm и Argo CD. Наконец, после выпуска рассмотрим систему в целом в главе 11.

Разработанная нами модель основана на пяти руководящих принципах, входящих в манифест «Приложение двенадцати факторов» (https://12factor.net). Если вам интересно, можете прочитать о руководящих принципах нашей модели в репозитории книги на GitHub (https://oreil.ly/MicroservicesUpandRunning).

Надеемся, этот краткий обзор помог вам получить представление о масштабах нашей модели и примере приложения. К концу книги мы реализуем полноценную систему. Но, чтобы добраться туда, нам придется принять много решений. Итак, в первую очередь нам понадобится определить способ отслеживания действительно важных решений.

Решения, решения...

В разработке ПО решения имеют большое значение. Профессиональным инженерам-программистам и архитекторам много платят за решения, которые они принимают, и за задачи, которые они решают. Качество ПО и бизнес-результаты, которые обеспечивают эти люди, зависят от качества этих решений.

Но решения не всегда легко принимать. К тому же они не всегда верны. Мы принимаем решения, наилучшие при имеющемся объеме информации, уровне опыта и таланта. Когда любая из этих переменных меняется, наши решения тоже должны меняться. Одни решения верны в определенное время, но устаревают, когда меняются технологии, люди или ситуации. Другие не были хорошими с самого начала. В любом случае нам нужен способ фиксировать важные решения, чтобы со временем мы могли их переоценить и улучшить.

Чтобы удовлетворить эту потребность, мы воспользуемся инструментом под названием *реестр архитектурных решений* (architecture decision record, ADR). Мы точно не знаем, кто изобрел термин ADR и когда он был впервые использован, но идея документирования проектных решений существует давно. Проблема заключается в том, что большинство людей не тратят на это время. По нашему опыту, ADR — чрезвычайно полезный инструмент, позволяющий прояснить принимаемые решения.

Каждая запись в реестре решений должна включать четыре важных компонента.

- *Контекст*. В чем заключается проблема, которую мы пытаемся решить? Какие ограничения имеются? Запись о принятом решении должна давать краткие ответы на эти вопросы. Благодаря этому мы сможем понять обоснование решения и почему его, возможно, потребуется обновить.
- *Альтернативы*. Решение не является таковым, если нет выбора, который нужно сделать. Хорошая запись о решении должна описывать возможные альтернативы. Это поможет нам лучше понять контекст и «пространство выбора» в момент принятия решения.
- *Выбор*. В основе решения лежит выбор. Каждая запись решения должна документировать сделанный выбор.

- *Влияние*. Решения имеют последствия, и записи решений должны документировать самые важные из них. Какие достоинства и недостатки имеет решение? Как выбор решения повлияет на рабочие подходы или на другие решения?

Вы можете оформлять записи в реестре решений так, как вам нравится: в виде текстовых файлов, использовать инструмент управления проектами или хранить их в электронной таблице. Содержание важнее, чем формат и инструментарий. Главное, чтобы каждая запись содержала четыре компонента, перечисленные выше.

В нашем демонстрационном проекте мы будем использовать формат, называемый *упрощенной записью архитектурного решения* (lightweight architectural decision record, LADR). Формат LADR, предложенный Майклом Нейгардом (https://oreil.ly/_mVoC), позволяет кратко описать принятое решение. Познакомимся с LADR поближе, создав запись вместе.

Если вы решите работать с другим форматом, отличным от LADR, то воспользуйтесь большим перечнем форматов ADR и шаблонов, которые предлагает Джоэл Паркер Хендерсон (https://oreil.ly/T3Tc-).

Создание упрощенной записи архитектурного решения

Первым ключевым решением, которое мы опишем, будет выполнение записей решений. Проще говоря, создадим ADR, в которой укажем, что намерены фиксировать принимаемые решения. Как упоминалось ранее, мы будем использовать формат LADR. Главное его преимущество — простота. Мы будем сохранять описание решений в простых текстовых файлах, а поскольку это текстовые файлы, то сможем управлять записями наших решений так же, как управляем исходным кодом.

Записи LADR оформляются на языке разметки Markdown (https://oreil.ly/oRyx0), который обеспечивает элегантный и простой способ написания документации. Самое замечательное в Markdown, что он не затрудняет чтение текста и большинство популярных инструментов знают, как его визуализировать. Например, Confluence, GitLab, GitHub и SharePoint могут представлять текст в формате Markdown и в виде удобочитаемого документа.

Чтобы создать первый документ LADR на основе Markdown, откройте текстовый редактор и создайте новый документ. Первое, что мы сделаем, — определим структуру.

Добавьте следующий текст в свой файл LADR:

```
# OPM1: Использовать ADRS для документирования решений

## Статус
Принято

## Контекст

## Решение

## Следствия
```

Это ключевые разделы записи о принятом решении. Символы # в начале строк — это маркеры Markdown, подсказывающие анализатору, что эти строки представляют собой заголовки. Обратите внимание: мы присвоили этому решению название, кратко описывающее его. Кроме того, название решения начинается со странной аббревиатуры OPM1. Это всего лишь краткий код формы, который поможет нам обозначить и понять, к какой части системы относится решение. В данном случае OPM1 указывает, что это первое наше решение и оно связано с операционной моделью (OPerating Model).

Подзаголовок `Статус` дает понять, на какой стадии жизненного цикла это решение находится. Например, если вы разрабатываете новое решение, по которому необходимо получить согласие, то первоначально ему можно присвоить статус `Предложено`. Или, если вы рассматриваете возможность изменения существующего решения, можете изменить его статус на `На рассмотрении`. В нашем случае мы уже приняли решение, поэтому установили статус `Принято`.

В разделе `Контекст` описывается проблема, ограничения и предпосылки для принятого решения. В данном случае мы постарались отразить необходимость регистрации важных решений и объяснить, почему это важно. Добавьте следующий текст (или его аналог) в раздел `Контекст` вашей записи:

```
## Контекст
Архитектура микросервисов сложна, и потребуется принять множество решений. Нам
понадобится способ отслеживать важные принимаемые решения, чтобы можно было
пересмотреть и переоценить их в будущем. Мы предпочли бы использовать простой
текстовый формат описания решений, чтобы не пришлось устанавливать какие-либо
новые программные средства.
```

Описав контекст, можно переходить к описанию фактического принятого нами решения, перечислению некоторых из рассмотренных альтернатив, а также обоснованию выбора в пользу LADR. Добавьте следующий текст в раздел `Decision`, чтобы задокументировать этот факт:

```
## Решение
Мы решили использовать формат упрощенной записи архитектурных решений
Майкла Найгарда — LADR. Этот текстовый формат достаточно легковесный,
чтобы удовлетворить наши запросы. Мы будем хранить каждую LADR в отдельном
текстовом файле и управлять файлами как исходным кодом. Мы также
рассмотрели следующие альтернативные решения:
* инструменты управления проектами (отвергнуто, поскольку мы бы не хотели
устанавливать дополнительные инструменты);
* неформальное ведение записей или сарафанное радио (ненадежно).
```

Осталось задокументировать следствия. В нашем случае одним из ключевых следствий является необходимость тратить время на документирование решений и управление записями. Зафиксируем это следующим образом:

```
## Следствия
* Нам потребуется создавать записи решений для ключевых решений.
* Понадобится инструмент для управления файлами записей решений
как исходным кодом.
```

Вот и все, что нужно для оформления LADR. Это невероятно полезный способ документирования мыслей, к тому же он имеет дополнительное преимущество, заставляя принимать взвешенные и обдуманные решения. По мере создания нашего примера приложения бронирования авиабилетов мы будем вести реестр принятых нами ключевых решений. Однако для экономии времени вместо создания полноценных записей о принятии решений мы будем отмечать их вот такими примечаниями.

КЛЮЧЕВОЕ РЕШЕНИЕ: ПРИМЕНЯТЬ ADR ДЛЯ ОТСЛЕЖИВАНИЯ РЕШЕНИЙ

Мы будем создавать записи в реестре ADR для фиксации ключевых решений, принятых при проектировании и создании нашей системы.

Полные версии записей о принятии решений вы найдете в репозитории GitHub для этой книги (https://github.com/implementing-microservices/ADRs).

Резюме

В этой главе мы познакомили вас с некоторыми основополагающими концепциями. Дали общее определение системы микросервисов как набора трех ключевых характеристик. Обозначили сокращение затрат на координацию как главное преимущество микросервисов. Кроме того, мы обсудили, как

сложность и аналитический паралич создают проблемы для пользователей микросервисов.

Чтобы помочь решить эти проблемы, мы разработали модель микросервисов «От архитектуры до релиза», которая ускорит процесс обучения разработчиков. Рассмотрели аспекты модели и темы, которые обсудим. Наконец, представили концепцию реестра архитектурных решений (ADR), который планируем использовать далее в книге.

Теперь, закончив обзорную часть, перейдем к созданию системы. В главе 2 мы начнем знакомиться с особенностями организации микросервисов и уделим особое внимание координации деятельности команд.