Содержание

Введение

Как всегда, технологический сектор находится в гуще важных перемен. Вот лишь некоторые примеры, связанные с возможностями интернета: сети с программной поддержкой и предоставление программного обеспечения как услуги (Software as a Service, SaaS). Благодаря этим инновациям возник большой спрос на платформы и архитектуры, способные улучшить процесс разработки и развертывания приложений. И небольшие, и крупные компании нуждаются во фреймворках и архитектурах, упрощающих процесс обновления их приложений и позволяющих чаще выводить на рынок новые версии.

Эти и многие другие перемены остаются пока достаточно новыми, и все же за время их существования успело появиться и исчезнуть множество технологий и фреймворков. Однако наиболее успешные остаются и продолжают помогать совершенствовать мир программного обеспечения, позволяя разработчикам – нам с вами – создавать новые и обновлять существующие приложения с еще большей гибкостью, чем прежде. К двум таким успешным технологиям относятся микросервисы и контейнеры. По сравнению с широко используемым монолитным подходом к разработке и развертыванию приложений, микросервисы упрощают эти процессы, особенно в крупных проектах, требующих совместной работы нескольких групп и все более длинного кода. В таких проектах даже небольшое изменение в коде может вызвать серьезные задержки. Современные микросервисы способны объединять большие объемы кода, обеспечивая гибкость и масштабируемость разработки и развертывания приложений, и все это в рамках проверенной парадигмы.

Когда я впервые начал знакомиться с микросервисами, существовало несколько ценных интернет-ресурсов, таких как microservices.io Криса Ричардсона (Chris Richardson) и martinfowler.com Джеймса Льюиса (James Lewis) и Мартина Фаулера (Martin Fowler), но почти не было книг, доходчиво объясняющих, почему технический директор или руководитель группы разработчиков должен (или не должен) перейти к использованию микросервисов. На рынке явно наблюдался пробел. Чем больше я овладевал предметом, тем больше думал: «Почему бы мне самому не попробовать восполнить этот пробел?» Вскоре я начал обдумывать идею написания своей собственной книги.

Эта книга для вас?

Я писал эту книгу, ориентируясь на две аудитории специалистов. В первую входят студенты, дизайнеры и архитекторы с опытом разработки программного обеспечения и конструирования систем. Даже если вы знакомы с микросервисами и/или контейнерами, эта книга, вероятно, первая на вашем пути, полностью посвященная им. Ее цель – не только представить исчерпывающий обзор обозначенных тем, но и дать достаточный объем информации, чтобы помочь решить, стоит или не стоит использовать эти технологии в вашем конкретном случае. Те из вас,

кто уже имеет практический опыт работы с микросервисами и/или контейнерами, возможно, захотят перелистнуть части I и II и погрузиться прямо в часть III, демонстрирующую развернутый пример центра обслуживания, написанный в соответствии со стандартными методологиями создания сервис-ориентированных архитектур (Service-Oriented Architecture, SOA). Данный пример показывает, как преобразовать монолитную архитектуру приложения в архитектуру на основе микросервисов и как в общую картину вписываются контейнеры Docker. Я думаю, что это глубокое погружение окажется для вас достаточно полезным и интересным, чтобы вызвать желание продолжить самостоятельное исследование мира микросервисов и контейнеров.

Вторая аудитория моих потенциальных читателей – это люди, рассматривающие данную тему с точки зрения бизнеса (руководители или менеджеры проектов, которым интересно познакомиться с основами). Возможно, вы прочитали интригующую статью в блоге о микросервисах. Может быть, вы давно присматривались к этому решению, но не могли найти хорошую книгу, описывающую последовательность шагов, которые нужно выполнить. Вероятно, вы подслушали, как инженеры обсуждают контейнеры Docker, и теперь хотите узнать больше, чтобы принять участие в последующих обсуждениях. Независимо от причин, побудивших вас обратиться к этой книге, по сути, она представляет собой букварь, полный простых для понимания примеров и содержащий минимальное количество жаргона, и послужит идеальным руководством для любого менеджера, ищущего новые пути увеличения эффективности разработки и развертывания приложений.

Эта книга для всех, кто:

- стремится повысить эффективность разработки промышленного программного обеспечения в своей организации;
- предполагает перейти к использованию микросервисов и контейнеров Docker и хочет понять, чем они отличаются от SOA;
- желает получить представление о микросервисах и Docker, чтобы обрести новые навыки, пользующиеся спросом на рынке.

Проще говоря, эта книга для тех, кто хочет узнать больше о микросервисах и контейнерах Docker. Я надеюсь, что вы один из них!

Зарегистрируйте свою копию книги «Микросервисы и контейнеры» на сайте InformIT, чтобы получить доступ к обновлениям и/или исправлениям по мере их появления. Для этого откройте в браузере страницу informit.com/register и выполните вход или создайте новую учетную запись. Введите код ISBN книги (9780134598383) и щелкните кнопку **Submit** (Отправить). На вкладке **Registered Products** (Зарегистрированные продукты) найдите ссылку **Access Bonus Content** (Дополнительные материалы) рядом с этим продуктом и перейдите по ней, чтобы получить доступ к любым имеющимся дополнительным материалам. Если вы хотите получать уведомления об эксклюзивных предложениях на новые издания или обновления, установите флажок, чтобы сообщить о своем желании.

Об авторе

Парминдер Сингх Кочер (Parminder Singh Kocher) родился и вырос в Индии. Вот уже более двух десятилетий он занимается созданием программных систем корпоративного класса. С 2005 г. работает в Cisco Systems, где в свое время руководил платформой Cisco Managed Services (CMS) и возглавлял несколько проектов программного обеспечения. В настоящее время занимает пост технического директора платформы Cisco Networking Academy и возглавляет инженерные группы, отвечающие за разработку платформы доступа следующего поколения в 180 странах. Помимо степеней бакалавра и магистра в области информатики, автор имеет степень магистра в области администрирования, полученную в Школе бизнеса имени Хэнкамера в Бэйлоре (Baylor Hankamer School of Business), и сертификат руководителя в области стратегии и инноваций, полученный в Школе менеджмента Слоана Массачусетского технологического института (MIT Sloan School of Management). Живет в городе Остин, расположенном в штате Техас, со своей женой и тремя детьми.

Часть I

МИКРОСЕРВИСЫ

Глава 1

Введение в микросервисы

Технологии меняются и влияют на развитие промышленности, которая, в свою очередь, предъявляет новые сложные требования к технологиям. Менее чем за 2 десятилетия мы перешли от эры коммутируемых модемов со скоростью 56 Кбит до 100-гигабитных сетей Ethernet. С увеличением скорости передачи данных выросли требования к скорости работы программного обеспечения, для разработки которого были созданы более совершенные и высокоуровневые языки программирования. Аналогично в области систем мы перешли от мейнфреймов к высокоскоростным серверам, а затем воплотили на этих серверах облачные технологии и технологии виртуализации. Теперь все чаще в разговорах специалистов стал встречаться термин «контейнеризация», обозначающий новую технологию, помогающую более эффективно использовать ресурсы.

Попутно появились новые парадигмы, такие как «модель – представление – контроллер» (Model – View – Controller, MVC), шаблоны интеграции корпоративных приложений (Enterprise Integration Patterns, EIP) и сервис-ориентированные архитектуры (Service-Oriented Architectures, SOA). В настоящее время в техническом мире активно обсуждаются архитектуры на основе микросервисов. Давайте попробуем выяснить, почему.

Что такое «микросервис»?

Микросервис – это независимый, автономный ресурс, спроектированный как отдельный выполняемый файл или процесс и взаимодействующий с другими микросервисами через стандартные, но легковесные межпроцессные связи, такие как протокол передачи гипертекста (HTTP), веб-службы RESTful (построенные на архитектуре репрезентативной передачи состояния – Representational State Transfer, REST), очереди сообщений и т. п. Уникальность микросервисов обусловлена тем, что каждая из них разрабатывается, тестируется, развертывается и масштабируется независимо от других микросервисов.

Идея использования микросервисов основана на лучших принципах разработки программного обеспечения, в том числе таких, как слабая взаимозависимость, высокая масштабируемость и ориентированность на службы.

Что подразумевается под словами «автономный ресурс»? А подразумевается под ними, что каждый микросервис выполняет ровно одну функцию, которая ведет себя одинаково для всех потребителей. Возьмем, к примеру, службу управления заказами, которая только обрабатывает заказы и больше ничего (даже уведомлений не отправляет). Но она может вызвать другой микросервис, отвечающий за отправку уведомлений об обработке. Такое разделение функций обеспечивает достаточную гибкость, т. к. каждый микросервис можно развивать, поддерживать, масштабировать, расширять и замещать независимо от других микросервисов.

Согласно этому определению, приложение на основе микросервисов – это просто группа из нескольких независимых и автономных микросервисов, каждый из которых реализует четко определенную функцию и для обеспечения общей функциональности приложения взаимодействует с другими микросервисами через четко определенные протоколы. Эту парадигму можно описать как архитектуру, в которой каждый микросервис выполняется с помощью отдельного процесса.

Возможно, вам интересно узнать, чем приложения на основе микросервисов отличаются от монолитных приложений на основе сервис-ориентированной архитектуры (SOA). Разница в том, что в монолитном приложении, известном также как *монолитная реализация*, все службы упакованы в один большой выполняемый файл, или файл WAR.

Рассмотрим простой пример: веб-приложение калькулятора. В монолитном приложении все операции калькулятора (сложение, вычитание и т. д.) могут быть написаны как отдельные функции в программе, причем для выполнения своей операции одна функция может вызывать другую непосредственно. Все они действуют в пределах одного процесса и взаимодействуют друг с другом посредством стандартного механизма вызова подпрограмм. Примерная конструкция такой программы показана на рис. 1.1.

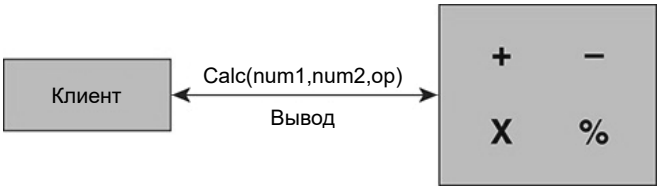

Рис. 1.1 ❖ Простая программа-калькулятор с монолитной архитектурой

Это очень простое приложение, реализация которого на основе микросервисов была бы излишеством. Однако только для того, чтобы понять суть, предположим, что разработчик, следуя парадигме микросервисов, сконструировал приложение калькулятора, реализовав каждую операцию в виде отдельной, автономной службы, как показано на рис. 1.2. В этом случае микросервисы будут вызывать друг друга по HTTP или другому протоколу. Если в какой-либо из функций в монолитном приложении возникнет ошибка (например, переполнение), это может привести к отказу всего приложения. Однако в случае с микросервисами проблема коснется только службы, где возникла ошибка, а остальные по-прежнему будут доступны для пользователей.

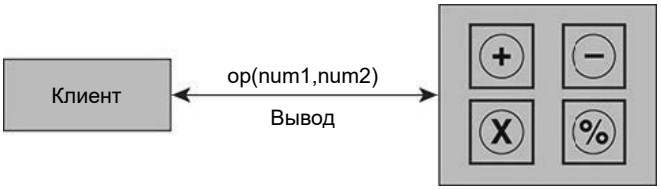

Рис. 1.2 ❖ Простая программа-калькулятор с архитектурой на основе микросервисов

Цель этого простого примера – подчеркнуть самое большое преимущество парадигмы микросервисов: она позволяет упростить реализацию сложного приложения, разделив его на более простые и автономные компоненты. Благодаря этой простоте можно, например, добавлять новые возможности, не влияя на другие службы.

Кроме того, каждый микросервис может развиваться и масштабироваться независимо. Например, предположим, что потребовалось добавить в приложение-калькулятор новую операцию, основанную на уже доступной функциональности: вычисление квадрата числа. Это очень простая операция и не требует изменения существующего кода. Мы создадим новый микросервис, который через стандартный документированный API вызовет микросервис «умножения» (см. рис. 1.3). Следовательно, нам потребуется написать, скомпилировать и развернуть только один микросервис, в отличие от монолитного приложения, требующего полной перекомпиляции и повторного развертывания с возможной приостановкой обслуживания клиентов.

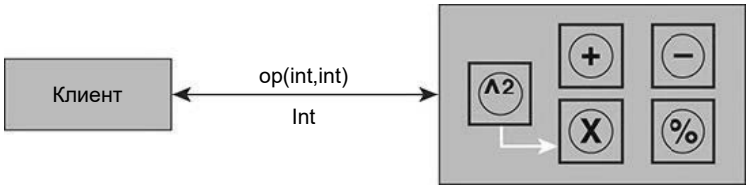

Рис. 1.3 ❖ В архитектуре на основе микросервисов легко добавить новую функцию «квадрат числа»

Также можно создавать микросервисы, которые вызываются только другими микросервисами и не используются непосредственно клиентским приложением. Например, как показано на рис. 1.4, клиент может вызвать лишь три микросервиса на уровне 1, тогда как первый микросервис на уровне 1 может вызвать два микросервиса на уровнях 2 и 3, как показано стрелками. Микросервисы, подобные этим двум, часто называют *вспомогательными*.

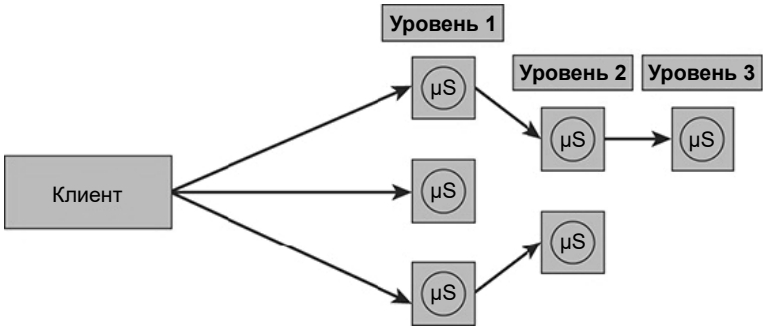

Рис. 1.4 ❖ Микросервиса, вызывающая другие микросервисы[1]

Идея микросервисов не нова, но в последнее время она набирает популярность, так как избавляет от проблем, характерных для монолитных приложений.

Давайте рассмотрим другой пример и обсудим эти проблемы. Представьте систему электронной коммерции и ее компоненты на верхнем уровне, как показано на рис. 1.5.

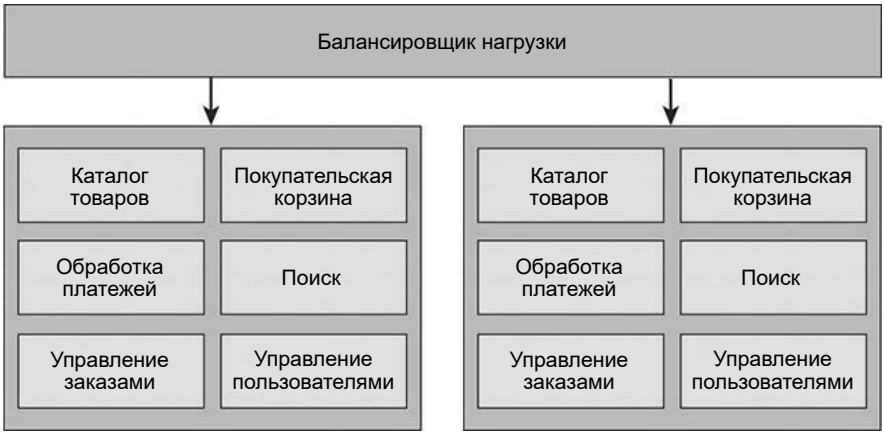

Рис. 1.5 ❖ Основные компоненты монолитной системы электронной коммерции

Эта система может прекрасно подходить для малых и средних компаний. Отдел технического обеспечения создает и развертывает в промышленном окружении единственный пакет, а горизонтальная масштабируемость системы легко обеспечивается развертыванием нескольких копий приложения и размещением балансировщика нагрузки перед ними. С ростом бизнеса растут и требования к возможностям, что влечет расширение кода, увеличение численности специалистов и, в свою очередь, сложность выпуска новых версий, развертывания и сопровождения приложения. Со временем приложение становится все более сложным, что затрудняет разграничение ответственности за код и функциональные

[1] Здесь и далее μS – микросервис. – *Прим. перев.*

возможности между разработчиками. В этот момент, как правило, все начинает рушиться, и организация сталкивается со следующими проблемами:

- низкой производительностью;
- плохой масштабируемостью;
- долгими циклами регрессионного тестирования;
- долгими циклами обновления и повторного развертывания, влекущими невозможность быстро внедрить мелкие исправления и улучшения;
- незапланированными простоями;
- возможными простоями на время, пока производится обновление;
- невозможностью внедрения новых технологий и языков программирования;
- невозможностью выборочного масштабирования необходимых компонентов или функций.

Одними из множества последствий, вытекающих из этих проблем, которые обычно остаются незамеченными, являются разочарование, испытываемое инженерами, и нарастание конфликтных ситуаций в коллективе.

В таких ситуациях очень может пригодиться парадигма микросервисов. Применение этой парадигмы оправдано только в отношении больших монолитных приложений, поскольку влечет за собой затраты, которые могут оказаться нецелесообразными, если приложение невелико или поддерживает малый бизнес. Чтобы на этом этапе зрелости разложить монолитное приложение на микросервисы, может потребоваться инвестировать кругленькую сумму. Поэтому организации обычно начинают добавлять новые возможности, реализуя их как микросервисы, а затем, уже получая отдачу от инвестиций, постепенно преобразовывают старое приложение.

Представьте, что нам нужно обновить компонент «покупательская корзина» из предыдущего примера. В зависимости от архитектуры старого программного обеспечения для этого может потребоваться не только добавление или обновление кода, но и выполнение регрессионного тестирования всего кода или только той его части, которая так или иначе связана с покупательской корзиной. Также потребуется перекомпилировать, протестировать и развернуть приложение целиком, что может привести к простою или замедлить работу приложения. Кроме того, разработчик может посчитать, что конкретную функциональность проще и эффективнее было бы реализовать на каком-то новом языке, таком как Scala. Это его желание, вероятно, останется невыполнимым, если не выделить средства, чтобы переписать все приложение на этом новом языке. Фактически разработчик окажется привязанным к выбору своих предшественников, который, возможно, был правильным в то время, но в данный момент уже не является оптимальным.

Давайте посмотрим, как микросервисы могут помочь в такой ситуации. Как уже обсуждалось, выделим компоненты монолитного приложения в отдельные микросервисы, как показано на рис. 1.6.

Эти микросервисы развертываются по отдельности, и каждая реализует только одну функцию. Если понадобится изменить микросервис, реализующую покупательскую корзину, мы должны будем изменить только код этого микросервиса. Сделать это будет намного проще, так же как проще будет провести тестирование и развертывание. Микросервисы не только решают проблемы, характерные для монолитных приложений, но и предлагают ряд преимуществ, которые способствуют внедрению технологии непрерывного развертывания.

Рис. 1.6 ❖ Компоненты системы электронной коммерции, выделенные в отдельные микросервисы

Модульная архитектура

Согласно отчету The Standish Group за 2016 г. («CHAOS Report 2016»), только 29 % программных проектов во всей отрасли уложились в отведенное время и смету. Это означает, что 71 % проектов потерпел неудачу или их успех оказался весьма спорным. Неудачи в основном были обусловлены проблемами качества, незавершенностью, перерасходом бюджета и т. д. Вследствие этого было введено много новых практик и стандартов управления проектами, которым должны были следовать организации, занимающиеся разработкой программного обеспечения (например, стандартов разработки программного обеспечения IEEE, стандартов тестирования программного обеспечения). Основная цель этих стандартов – управление сложностью с применением передовых практик. В результате организации, принявшие на вооружение эти стандарты, улучшили шансы на завершение проектов и увеличили срок службы приложений.

Средний срок службы программных приложений или платформ составляет от 4 до 6 лет, после чего они устаревают по различным причинам. В числе причин могут быть изменение требований с течением времени, невозможность масштабирования из-за устаревшей архитектуры, устаревание технологий и т. д. Индустрия постоянно развивается и все время требует создания приложений нового поколения, что означает необходимость переписывать программное обеспечение с использованием новейших технологий, архитектур и передовых практик. Но в какой-то момент необходимо задать вопрос: так ли необходимо изменять каждый компонент, т. е. весь пакет? Оказывается, это требуется не всегда. Некоторые компоненты или части действительно можно усовершенствовать, задействовав новые технологии, но обычно этот вариант не подходит, потому что архитектура не обеспечивает модульности и не позволяет заменять отдельные компоненты или части программного обеспечения новым кодом.

Мы разрабатываем монолитные приложения, а значит, должны следовать стандартам, помогающим преодолеть сложности. Но если устранить саму сложность с помощью парадигмы микросервисов, мы получим модульную архитектуру, значительно увеличивающую срок службы программного обеспечения, и сможем

уменьшить нашу зависимость от большого количества стандартов, избавиться от громоздкого процесса разработки и сэкономить время, а значит, ускорить цикл создания программного обеспечения.

Помимо эффективности процесса, модульная архитектура также позволит добиться существенной экономии в будущем, когда потребуется обновить платформу. Вместо того чтобы все начать сначала, мы сможем хирургическим путем удалить устаревшие микросервисы и заменить их новыми, реализованными с использованием более совершенных технологий и архитектур. Это одно из ключевых долгосрочных преимуществ парадигмы микросервисов, отличающих ее от других парадигм. Но даже простое увеличение модульности делает внедрение подхода на основе микросервисов достойным инвестиций.

Другие преимущества микросервисов

Помимо достоинств, перечисленных выше, микросервисовы могут предложить следующие выгоды:

- **простота.** Каждый микросервис выполняет только одну четко определенную функцию, поэтому требуется меньше кода, меньше зависимостей от другого кода и уменьшается вероятность ошибок;
- **масштабируемость.** Для масштабирования монолитного приложения его необходимо развернуть на нескольких серверах и настроить балансировщик нагрузки. Невозможно масштабировать только часть приложения. Здесь действует принцип «все или ничего». С микросервисами можно масштабировать только компоненты, подвергающиеся высокой нагрузке, как показано на рис. 1.7. Возможность дифференцированной масштабируемости является очень простой и важной особенностью микросервисов;

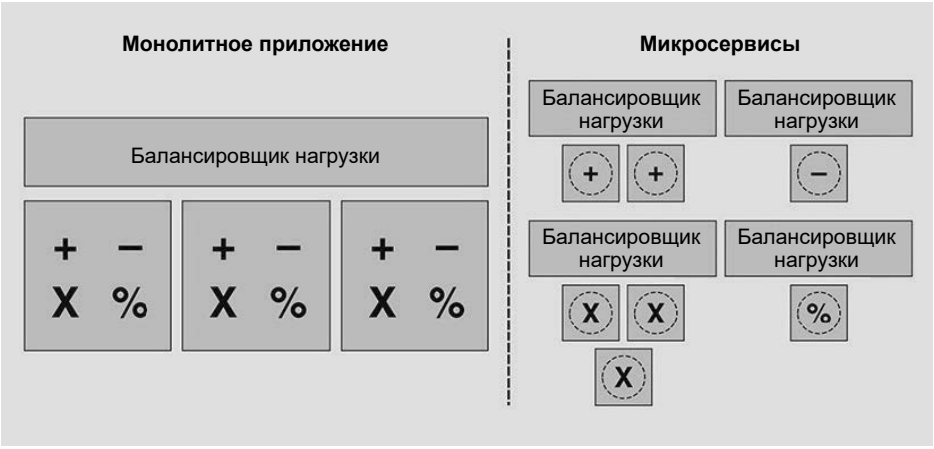

Рис. 1.7 ❖ Сравнение масштабируемости

- **непрерывное развертывание.** Благодаря меньшему количеству взаимозависимостей в коде и более быстрому циклу разработки парадигма микросервисов поддерживает культуру непрерывного развертывания и интегра-

ции разработки и эксплуатации (DevOps) и фактически подталкивает к ее использованию;

- **больше свободы и меньше зависимости.** Микросервисы по определению автономны и независимы. Команда разработчиков может сосредоточиться на своем микросервисе и свободно расширять ее возможности, не опасаясь нарушить работу другого микросервиса, пока они гарантируют неизменность интерфейса или реализуют новый интерфейс, обратно совместимый с прежним;
- **изоляция отказов.** Изоляция отказов – это явление, когда отказ в одной части системы не приводит к отказу всей системы, т. е. отказы оказываются изолированными от системы. В монолитном приложении сбой в любой его части приведет к сбою всего приложения, потому что оно представляет собой единый процесс. В случае с микросервисами дело обстоит иначе: сбой в одном микросервисе может привести к отказу этого микросервиса, но вовсе не обязательно приведет к отказу всей программы, потому что отказавший микросервис выполняется в отдельном процессе. Например, в системе электронной коммерции, основанной на архитектуре микросервисов, в случае сбоя микросервиса «отзывы о продукте» пользователи по-прежнему смогут просматривать список товаров, имеющихся в наличии, выбирать товары для покупки, просматривать содержимое покупательской корзины и размещать заказы. Единственное, чего они не смогут, – это увидеть отзывы, пока не будет исправлен микросервис просмотра отзывов. Если бы приложение было монолитным, ошибка в компоненте, отвечающем за просмотр отзывов, могла бы прервать работу всего приложения;
- **разделение и децентрализация данных.** В отличие от монолитных приложений, где все данные обычно хранятся вместе в центральной базе данных, микросервисовы дают возможность разделить данные. Каждый микросервис может владеть только своими данными и не делиться ими с другими микросервисами;
- **широта выбора.** В отличие от монолитного приложения, где все компоненты используют единую базу данных, платформу и должны быть написаны на одном языке программирования, микросервисы дают возможность использовать инструменты, лучше подходящие для каждого конкретного случая. Один микросервис может использовать Oracle и ОС Linux, а другая – NoSQL и Microsoft Windows. Больше нет необходимости связывать себя с определенными стеками технологий.

Недостатки микросервисов

Ничто не дается бесплатно, и преимущества микросервисов тоже имеют свою цену. Двигаясь в сторону микросервисов, мы должны знать, какие проблемы связаны с этой архитектурой. Не волнуйтесь! В следующей части данной книги вы узнаете, как преодолеть эти проблемы с использованием определенных систем и приложений. А теперь перечислим некоторые проблемы, характерные для микросервисов:

- **сложность поиска и устранения неисправностей.** Микросервисы предлагают свои возможности посредством механизма взаимодействий между

микросервисами, что увеличивает число потенциальных точек отказа. Это делает ответы на следующие вопросы более сложными:
  - Насколько нормально работает моя система в данный момент?
  - Если конечный пользователь сообщает о такой проблеме, как низкая производительность или длительные периоды ожидания, с чего начать устранение неполадок?
  - Отследить путь обработки запроса в монолитном приложении намного проще. Но в приложении, состоящем из микросервисов, каждый запрос может быть разбит на несколько запросов, обрабатываемых разными микросервисами. Поиск и устранение неполадок в этом случае могут стать немного сложнее;
- **увеличенные задержки.** Внутрипроцессные взаимодействия (как и в монолитных приложениях) выполняются намного быстрее межпроцессных (как в случае с микросервисами);
- **сложность сопровождения.** Когда приложение состоит из сотен или даже тысяч микросервисов, группам оперативного сопровождения приходится преодолевать сложности, связанные с настройкой инфраструктуры, развертыванием, мониторингом, резервным копированием и управлением. Можно даже сказать, что сложности монолитной архитектуры переносятся на системную сторону микросервисов. Тем не менее эти сложности можно преодолеть с помощью высокого уровня автоматизации;
- **управление версиями.** Из-за того что приложение может состоять из тысяч микросервисов, управление версиями становится немного сложнее. Для его осуществления требуется использовать более совершенные системы управления версиями.