**УДК 004.432**
**ББК 32.972.1**
**К21**

**Джон Карнелл, Иллари Уайлупо Санчес**
**К21** Микросервисы Spring в действии / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2022. – 490 с.: ил.

**ISBN 978-5-97060-971-2**

В книге рассказывается о том, как создавать приложения на основе микросервисов с использованием Java и Spring. Описываются особенности управления конфигурацией микросервисов и передовые практики их разработки. Уделено внимание защите потребителей, когда один или несколько экземпляров микросервисов выходят из строя.

Начав с создания простых служб, читатель постепенно перейдет к знакомству с приемами эффективного журналирования и мониторинга, научится реструктурировать приложения на Java с помощью инструментов Spring, освоит управление API с помощью Spring Cloud Gateway.

Издание предназначено для разработчиков на Java, имеющим опыт создания распределенных приложений и использования Spring, а также всем, кому интересно узнать, что необходимо для развертывания приложения на основе микросервисов в облаке.

УДК 004.432
ББК 32.972.1



ISBN (анг.) 978-1-617-29695-6
ISBN (рус.) 978-5-97060-971-2

*Я посвящаю эту книгу всем женщинам,*
*которые делают карьеру в естественно-научных,*
*инженерно-технических и математических дисциплинах.*

*Нет ничего невозможного, если упорно трудиться.*

# *Краткое оглавление*

# Содержание

# *Предисловие*

Эта книга – часть моей мечты внести свой вклад в развитие той области, которой я больше всего увлекаюсь, – информатики и, в частности, разработки программного обеспечения. Эти дисциплины демонстрируют свою исключительную важность во взаимосвязанном и глобальном настоящем. Мы ежедневно наблюдаем невероятные преобразования, которые они вызывают во всех сферах человеческой деятельности. Но зачем писать об архитектуре микросервисов, если есть много других достойных тем?

Слово «микросервисы» имеет множество толкований. В этой книге я подразумеваю под микросервисами распределенные, слабо связанные программные службы, которые выполняют ограниченное количество четко определенных задач. Микросервисы стали появляться как альтернатива монолитным приложениям, помогающая бороться с традиционными проблемами сложности в большой кодовой базе, разбивая ее на мелкие, четко ограниченные части.

В течение своей 13-летней карьеры я занималась разработкой программного обеспечения на разных языках и в разных программных архитектурах. Архитектуры, с которых я начинала, в настоящее время практически устарели. Современный мир заставляет нас постоянно двигаться вперед, и инновации в области разработки программного обеспечения развиваются ускоренными темпами. По этой причине, находясь в состоянии постоянного поиска новейших знаний и практик, несколько лет тому назад я решила окунуться в мир микросервисов. С тех пор я использовала эту архитектуру чаще других из-за ее преимуществ (таких как масштабируемость, скорость и удобство сопровождения). Успешный опыт работы в области микросервисов побудил меня взять на себя задачу написать эту книгу, чтобы систематизировать свои знания и поделиться ими с вами.

Как разработчик программного обеспечения, я понимаю, насколько важно постоянно приобретать и применять новые знания. Прежде чем взяться за эту книгу, я решила поделиться своими выводами и начала публиковать статьи о микросервисах в блоге коста-риканской компании (на моей родине), занимающейся разработкой программного обеспечения. Когда я писала эти статьи, я поняла, что нашла новую страсть и цель в своей профессиональной карьере. Через несколько месяцев после публикации одной

из моих статей я получила электронное письмо от издательства Manning Publications с предложением написать второе издание этой, которым и делюсь с вами сегодня.

Первое издание этой книги было написано Джоном Карнеллом (John Carnell), непревзойденным профессионалом с многолетним опытом разработки программного обеспечения. Я написала это второе издание на основе той книги, добавив мои собственные видение и понимание. Второе издание «*Микросервисов Spring*» покажет вам, как реализовать разнообразные шаблоны проектирования, чтобы получить успешную архитектуру микросервисов с помощью Spring – фреймворка, предлагающего готовые решения для многих распространенных задач разработки, с которыми вы неизбежно столкнетесь как разработчик микросервисов. А теперь давайте начнем захватывающее путешествие в мир микросервисов со Spring.

# *Об этой книге*

Книга «*Микросервисы Spring*», *второе издание* написана для разработчиков на Java/Spring, которым нужны практические советы и примеры создания и ввода в эксплуатацию приложений на основе микросервисов. Работая над этой книгой, мы хотели сохранить центральную идею, заложенную в первое издание, – применение основных шаблонов микросервисов, соответствующих новейшим практикам и примерам Spring Boot и Spring Cloud. Практически в каждой главе вы найдете конкретные шаблоны проектирования микросервисов, а также примеры реализации Spring Cloud.

## Кому адресована эта книга

- Разработчикам на Java, имеющим некоторый опыт (1–3 года) создания распределенных приложений.
- Имеющим опыт (более 1 года) использования Spring.
- Желающим научиться создавать приложения на основе микросервисов.
- Интересующимся особенностями использования микросервисов для создания облачных приложений.
- Стремящимся узнать, являются ли Java и Spring подходящими технологиями для создания приложений на основе микросервисов.
- Всем, кому интересно узнать, что необходимо для развертывания приложения на основе микросервисов в облаке.

## Структура книги

Эта книга состоит из 12 глав и 3 приложений.

- Глава 1 знакомит с архитектурой микросервисов как важным и актуальным подходом к созданию приложений, особенно облачных.
- Глава 2 рассказывает о технологиях Spring Cloud, которые мы будем использовать, и описывает порядок создания облачных микросервисов с учетом передовых практик известного руководства «Приложение двенадцати факторов». В этой главе также приводится пример создания простого микросервиса на основе REST с помощью Spring Boot.
- Глава 3 показывает микросервисы с точки зрения архитектора, прикладного программиста и инженера DevOps, а также демонстрирует реализацию некоторых передовых методов в нашем первом микросервисе на основе REST.
- Глава 4 знакомит с миром контейнеров, подчеркивая основные различия между контейнерами и виртуальными машинами (ВМ). Она также показывает, как запускать микросервисы в контейнерах, используя плагины Maven и команды Docker.
- Глава 5 описывает особенности управления конфигурацией микросервисов с помощью Spring Cloud Config. Spring Cloud Config помогает организовать хранение конфигураций микросервисов в едином репозитории с учетом их версий и воспроизводить их во всех экземплярах микросервисов.
- Глава 6 знакомит с шаблоном маршрутизации обнаружения служб. Здесь вы узнаете, как использовать Spring Cloud и службу Netflix Eureka, чтобы обеспечить независимость местоположения ваших служб от клиентов, которые их используют. Вы также узнаете, как реализовать балансировку нагрузки на стороне клиента с помощью Spring Cloud LoadBalancer и клиента Netflix Feign.
- Глава 7 посвящена защите потребителей ваших микросервисов, когда один или несколько экземпляров микросервисов выходят из строя или находятся в нерабочем состоянии. В этой главе показано, как с помощью Spring Cloud и Resilience4j реализовать шаблоны размыкателя цепи (circuit breaker), отката к резервной реализации (fallback) и герметичных отсеков (bulkhead).
- Глава 8 охватывает шаблон шлюза маршрутизации (gateway routing). Здесь, используя Spring Cloud Gateway, мы создадим единую точку входа для всех наших микросервисов и увидим, как использовать фильтры Spring Cloud Gateway для создания политик, которые могут применяться ко всем службам, доступным через шлюз.

- Глава 9 рассказывает, как реализовать аутентификацию и авторизацию с помощью Keycloak. Здесь рассматриваются некоторые основные принципы OAuth2 и способы использования Spring и Keycloak для защиты архитектуры микросервисов.
- Глава 10 рассматривает реализацию асинхронного обмена сообщениями между микросервисами с помощью Spring Cloud Stream и Apache Kafka. Она также показывает, как использовать Redis для кеширования запросов.
- Глава 11 показывает, как реализовать шаблоны журналирования, такие как корреляция журналов (log correlation), агрегирование журналов (log aggregation) и трассировка, с помощью Spring Cloud Sleuth, Zipkin и ELK Stack.
- Глава 12 является краеугольным камнем этой книги. Здесь мы соберем воедино все службы, создаваемые на протяжении всей книги, и развернем их в Amazon Elastic Kubernetes Service (Amazon EKS). Мы также обсудим автоматизацию процессов сборки и развертывания микросервисов с помощью таких инструментов, как Jenkins.
- Приложение A перечисляет дополнительные передовые практики строительства микросервисных архитектур и объясняет «Модель зрелости Ричардсона».
- Приложение B содержит дополнительные материалы по OAuth2. OAuth2 – чрезвычайно гибкая модель аутентификации, и в этом приложении дается краткий обзор различных способов использования OAuth2 для защиты приложений и микросервисов.
- Приложение C рассказывает, как контролировать микросервисы Spring Boot с помощью таких технологий, как Spring Boot Actuator, Micrometer, Prometheus и Grafana.

В целом мы советуем всем разработчикам прочитать главы 1, 2 и 3, где приводится важная информация о передовых методах реализации микросервисов с использованием Spring Boot и Java 11. Если вы плохо знакомы с Docker, то мы настоятельно рекомендуем внимательно прочитать главу 4, которая кратко представляет все ключевые концепции Docker, используемые в книге.

В остальной части книги обсуждаются некоторые шаблоны проектирования микросервисов, такие как обнаружение служб, распределенная трассировка, API-шлюз и др. Было бы желательно читать главы по порядку и опробовать примеры кода у себя. Если вам жалко тратить время на ввод кода примеров вручную, то можете загрузить его из репозитория GitHub по адресу https://github.com/ihuaylupo/manning-smia.

## О примерах кода

Эта книга содержит много примеров исходного кода в виде листингов и фрагментов в обычном тексте. В обоих случаях исходный код оформляется `моноширинным шрифтом`, чтобы его можно было отличить от обычного текста. Код всех примеров доступен в репозитории GitHub https://github.com/ihuaylupo/manning-smia.

Примеры хранятся в отдельных папках, по одной для каждой главы. Также обратите внимание, что весь код в этой книге создан для Java 11 и предполагает использование Maven в качестве основного инструмента сборки и Docker в качестве программной платформы контейнеризации. В каждой папке для каждой главы имеется файл README.md, где вы найдете следующую информацию:

- краткое введение в главу;
- перечень необходимых инструментов;
- раздел «How to Use», описывающий порядок использования;
- команду для сборки примеров;
- команду для запуска примеров;
- контакты и дополнительную информацию.

Одна из основных концепций, которых мы старались придерживаться на протяжении всей книги, заключается в том, что примеры кода в каждой главе должны работать полностью независимо от других глав. Что это значит? Вы можете, например, взять код из главы 10 и запустить его, минуя примеры из предыдущих глав. Вы увидите, что для каждой службы, созданной в каждой главе, имеется соответствующий образ Docker. В каждой главе мы используем Docker Compose для запуска образов Docker, чтобы гарантировать воспроизводимость среды выполнения.

Во многих случаях оригинальный исходный код был переформатирован; мы добавили переносы строк и изменили ширину отступов, чтобы уместить строки кода по ширине книжной страницы. В редких случаях даже этого оказалось недостаточно, поэтому в листингах вы увидите маркеры продолжения строки (➥). Кроме того, во многих листингах в книге, которые объясняются в тексте, мы убрали комментарии. Также многие листинги сопровождаются дополнительными аннотациями, подчеркивающими наиболее важные идеи.

## Живое обсуждение книги

Приобретая книгу «*Микросервисы Spring*», *второе издание*, вы получаете бесплатный доступ к частному веб-форуму, организованному издательством Manning Publications, где можно оставлять коммен-

тарии о книге, задавать технические вопросы, а также получать помощь от автора и других пользователей. Чтобы получить доступ к форуму и зарегистрироваться на нем, откройте в веб-браузере страницу https://livebook.manning.com/book/spring-microservices-in-action-second-edition/discussion. Узнать больше о форумах Manning и познакомиться с правилами поведения можно по адресу https://livebook.manning.com/#!/discussion.

Издательство Manning обязуется предоставить своим читателям место встречи, где может состояться содержательный диалог между отдельными читателями и между читателями и автором. Но со стороны авторов отсутствуют какие-либо обязательства уделять форуму какое-то определенное внимание – их присутствие на форуме остается добровольным (и неоплачиваемым). Мы предлагаем задавать авторам стимулирующие вопросы, чтобы их интерес не угасал!

Форум и архив с предыдущими обсуждениями остается доступным на сайте издательства, пока книга продолжает издаваться.

## Об авторах

**Джон Карнелл (John Carnell)** – архитектор программного обеспечения, возглавляет группу по взаимодействию с разработчиками в Genesys Cloud. Большую часть своего рабочего времени Джон учит клиентов Genesys Cloud и внутренних разработчиков, как развертывать облачные решения для контакт-центров и телефонии, а также рассказывает о передовых методах разработки на основе облачных технологий. Он занимается созданием микросервисов на базе телефонии с использованием платформы AWS. Его повседневная работа заключается в разработке и создании микросервисов для таких технологических платформ, как Java, Clojure и Go. Джон – одаренный оратор и писатель. Он регулярно выступает в местных группах пользователей и на симпозиуме The No Fluff Just Stuff Software Symposium. За последние 20 лет Джон был автором, соавтором и техническим рецензентом ряда книг по технологиям на основе Java и отраслевых публикаций. Имеет степень бакалавра, полученную в университете Маркетт и степень магистра делового администрирования, полученную в Висконсинском университете в городе Ошкош. Джон – увлеченный технолог, постоянно исследует новые технологии и языки программирования. Он живет в Кэри (Северная Каролина) со своей женой Джанет, тремя детьми (Кристофером, Агатой и Джеком) и – да, с собакой Вейдером.

**Иллари Уайлупо Санчес (Illary Huaylupo Sánchez)** – инженер-программист, выпускница университета Сенфотек. Имеет также степень магистра делового администрирования в области управления ИТ, полученную в Латиноамериканском университете науки и технологий (ULACIT) в Коста-Рике. Имеет обширные познания

в области разработки программного обеспечения, опыт работы с Java и другими языками программирования, такими как Python, C#, Node.js, а также с другими технологиями, такими как различные базы данных, фреймворки, облачные службы и многое другое. В настоящее время Иллари работает старшим инженером-программистом в Microsoft, Сан-Хосе, Коста-Рика, где проводит большую часть своего времени, исследуя и разрабатывая множество современных проектов. В ее профессиональном портфолио имеется также 12-летний опыт работы в качестве сертифицированного разработчика Oracle и старшего инженера-программиста в крупных компаниях, таких как IBM, Gorilla Logic, Cargill и BAC Credomatic (престижный латиноамериканский банк). Иллари любит сложные задачи и всегда готова изучать новые языки программирования и новые технологии. В свободное время она любит играть на басгитаре и проводить время с семьей и друзьями. Связаться с Иллари можно по адресу illaryhs@gmail.com.

## Об иллюстрации на обложке

На обложке «*Микросервисы Spring*» изображена иллюстрация, подписанная как «Человек из Хорватии». Она взята из недавнего переиздания книги Бальтазара Хакке (Balthasar Hacquet) «Images and Descriptions of Southwestern and Eastern Wenda, Illyrians, and Slavs», опубликованной этнографическим музеем в городе Сплит (Хорватия) в 2008 году. Хакке (1739–1815) – австрийский врач и ученый, много лет изучавший ботанику, геологию и этнографию Австрийской империи, а также Венето, Юлийских Альп и западных Балкан, населенных в прошлом народами иллирийских племен. Рисованные иллюстрации сопровождают многие научные статьи и книги Хаке. Богатое разнообразие рисунков в публикациях Хаке ярко свидетельствует об уникальности и индивидуальности восточных альпийских и северо-западных балканских регионов всего 200 лет назад.

Это было время, когда по одежде легко можно было отличить жителей соседних деревень, разделенных всего несколькими милями, или принадлежность к разным социальным классам или профессиям. С тех пор стиль одежды сильно изменился, и исчезло разнообразие, характеризующее различные области и страны. В настоящее время трудно отличить по одежде даже жителей разных континентов, и жители живописных городков и деревень в словенских Альпах или прибрежных балканских городах почти не отличаются от жителей других частей Европы.

Мы в издательстве Manning славим изобретательность, предприимчивость и радость компьютерного бизнеса обложками книг, изображающими богатство региональных различий двухвековой давности, оживших благодаря таким иллюстрациям, как эта.

# 1 Добро пожаловать в Spring Cloud

**Эта глава:**

- знакомит с архитектурой микросервисов:
- рассказывает, почему компании используют микросервисы;
- демонстрирует приемы использования Spring, Spring Boot и Spring Cloud для создания микросервисов;
- описывает модели облачных вычислений.

Внедрение любой новой архитектуры – непростая задача, сопряженная с множеством проблем, таких как масштабируемость приложений, обнаружение служб, мониторинг, распределенная трассировка, безопасность, управление и многих других. Эта книга познакомит вас с миром микросервисов Spring, научит решать все эти проблемы и покажет, какие компромиссы следует учитывать при реализации бизнес-приложений с использованием архитектуры микросервисов. В этой книге вы узнаете, как создавать приложения на основе микросервисов с использованием таких технологий, как Spring Cloud, Spring Boot, Swagger, Docker, Kubernetes, ELK (Elasticsearch, Logstash и Kibana), Stack, Grafana, Prometheus и др.

Если вы разработчик на Java, то эта книга поможет вам плавно перейти от создания традиционных приложений на основе Spring к созданию приложений на основе микросервисов, которые можно развернуть в облаке. Здесь вы найдете практические примеры, диаграммы и подробное описание с дополнительными сведениями о реализации архитектуры микросервисов.

Прочитав эту книгу, вы научитесь применять на практике такие технологии и методы, как балансировка нагрузки на стороне клиентов, динамическое масштабирование, распределенная трассировка и многие другие, и создавать гибкие, современные и автономные бизнес-приложения на основе микросервисов с помощью Spring Boot и Spring Cloud. Вы также научитесь создавать свои собственные конвейеры сборки/развертывания, чтобы обеспечить непрерывную доставку и интеграцию с вашим бизнесом, применяя такие технологии, как Kubernetes, Jenkins и Docker.

## 1.1. Эволюция архитектуры микросервисов

Программная архитектура охватывает все фундаментальные аспекты, определяющие структуру, работу и взаимодействие программных компонентов. Эта книга рассказывает, как создать архитектуру микросервисов, состоящую из слабо связанных программных служб, выполняющих узкий круг четко определенных задач и взаимодействующих посредством передачи сообщений по сети. Для начала рассмотрим различия между микросервисами и некоторыми другими распространенными архитектурами.

### 1.1.1. N-уровневая архитектура

Одним из распространенных типов архитектуры корпоративного программного обеспечения является многоуровневая или n-уровневая архитектура. Приложения с этой архитектурой делятся на несколько уровней, каждый со своими обязанностями и функциями, такими как пользовательский интерфейс, службы, данные, тестирование и т. д. Например, при создании приложения создается отдельный проект или решение для пользовательского интерфейса, затем еще один для служб, еще один для уровня данных и т. д. В итоге объединение нескольких проектов дает целое приложение. В больших корпоративных системах n-уровневые приложения имеют множество преимуществ, в том числе:

- n-уровневые приложения позволяют четко разделить задачи и рассматривать такие элементы, как пользовательский интерфейс, данные и бизнес-логику по отдельности;
- команды разработчиков могут работать над различными компонентами независимо друг от друга;
- корпоративная архитектура хорошо изучена, поэтому относительно легко найти квалифицированных разработчиков для многоуровневых проектов.

Но n-уровневые приложения имеют и недостатки:

- после внесения изменений в код приходится останавливать и повторно запускать все приложение;

- сообщения, как правило, курсируют вверх и вниз через уровни, что может быть неэффективным;
- рефакторинг большого многоуровневого приложения после развертывания может оказаться сложной задачей.

Некоторые из тем, обсуждаемых в этой книге, относятся непосредственно к многоуровневым приложениям, однако мы в большей степени сосредоточимся на различениях между микросервисами и еще одной распространенной архитектурой, часто называемой монолитом.

### 1.1.2. Что такое монолитная архитектура?

Многие веб-приложения небольшого и среднего размера создаются с использованием монолитной архитектуры . Монолитное приложение доставляется как единственный развертываемый программный артефакт. Все его компоненты – пользовательский интерфейс, бизнес-логика и логика доступа к базе данных – объединены в единое приложение и развертываются на сервере приложений. На рис. 1.1 показана базовая архитектура такого приложения.

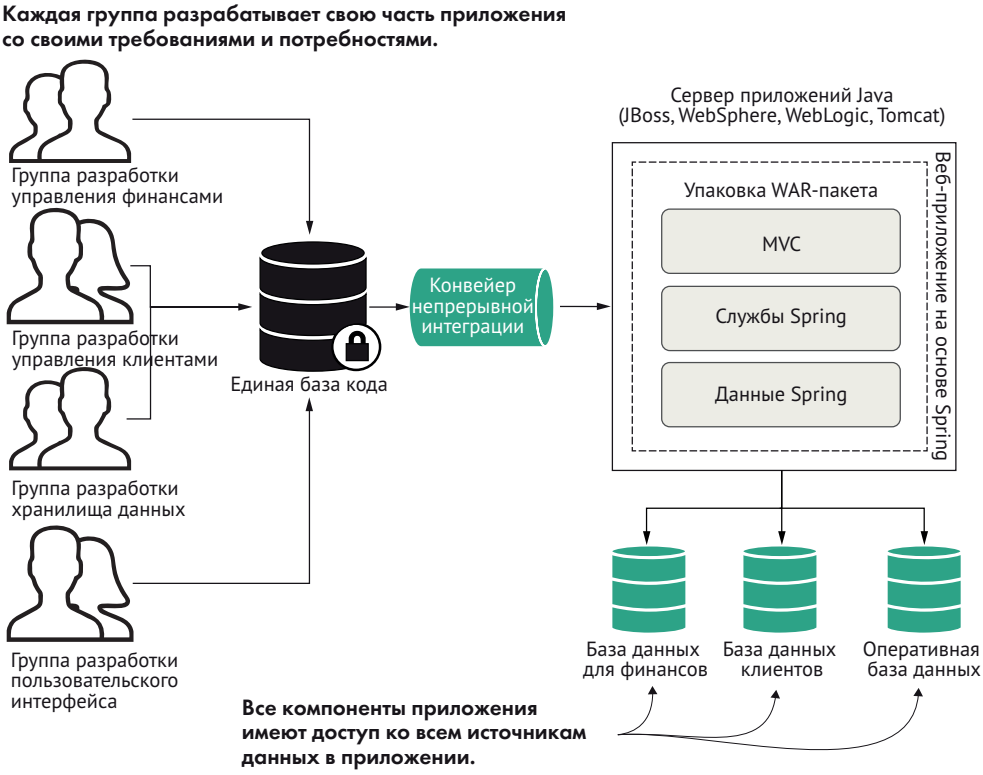


**Рис. 1.1.** Монолитные приложения вынуждают несколько групп разработчиков синхронизировать дату развертывания, потому что их код необходимо собирать, тестировать и развертывать как единое целое

Конечно, приложение может быть развернуто как единое целое, но часто над одним таким приложением работает несколько групп разработчиков. Каждая группа отвечает за свою часть приложения, обычно ориентированную на конкретных клиентов. Например, представьте такой сценарий: у нас есть внутреннее приложение для управления взаимоотношениями с клиентами (Customer Relations Management, CRM), которое предполагает координацию действий нескольких групп разработки пользовательского интерфейса, логики управления клиентами, хранилища данных, логики управления финансами и, возможно, многих других.

Сторонники микросервисной архитектуры часто отзываются о монолитных приложениях с негативным оттенком, но иногда монолитная архитектура является отличным выбором. Монолиты проще создавать и развертывать, чем более сложные многоуровневые или микросервисные архитектуры. Если сценарий использования четко определен и вряд ли изменится в будущем, то часто решение начать с монолита может оказаться не таким плохим.

Однако с увеличением размеров и сложности приложения управлять монолитом становится все труднее. Любое изменение в монолитном приложении может иметь каскадный эффект, оказывая влияние на другие части приложения, что может существенно осложнить интеграцию и развертывание в промышленной системе. Наш третий вариант, микросервисная архитектура, предлагает большую гибкости и удобство сопровождения.

### 1.1.3. Что такое микросервис?

Идея микросервисов изначально возникла в сообществе разработчиков программного обеспечения как прямой ответ на многие проблемы (как технические, так и организационные), связанные с попытками масштабирования больших монолитных приложений. *Микросервис* – это небольшая, слабо связанная распределенная служба. Микросервисы позволяют взять приложение с обширным набором функций и разложить его на простые в управлении компоненты с четко определенными обязанностями. Микросервисы помогают преодолевать традиционные проблемы сложности большой базы кода, разбивая ее на небольшие четко определенные части.

Ключевые понятия, о которых следует помнить, рассуждая о микросервисах, – это *декомпозиция* и *развязка* (unbunding). Функции приложений должны быть полностью независимы друг от друга. Если взять упомянутое выше приложение CRM и разложить его на микросервисы, то получившийся результат мог бы выглядеть примерно так, как показано на рис. 1.2.

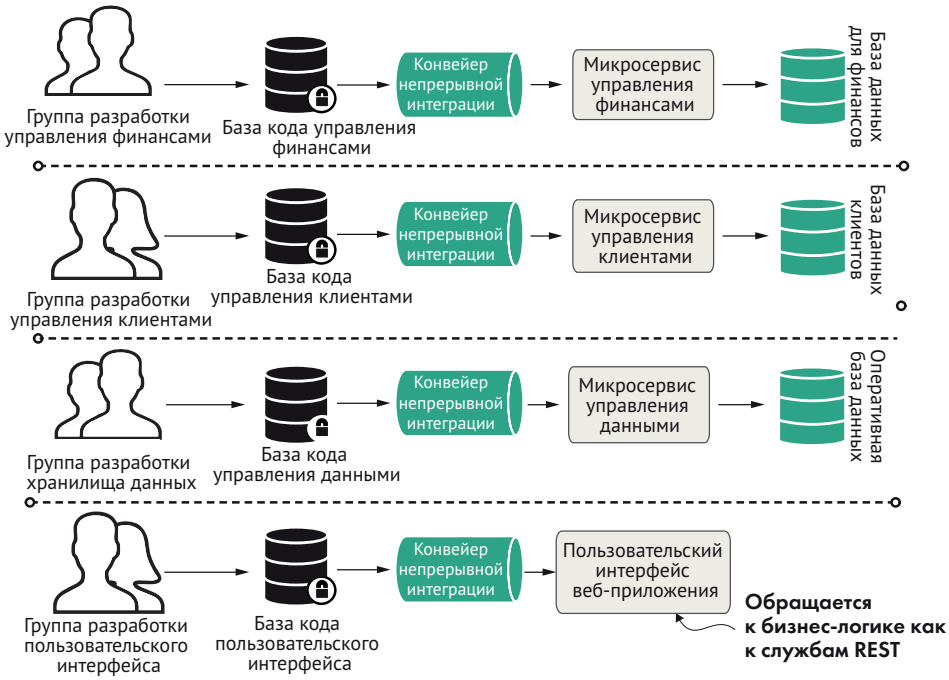


**Рис. 1.2.** При использовании микросервисной архитектуры приложение CRM разбивается на набор независимых микросервисов, что позволяет каждой группе разработчиков двигаться вперед в своем собственном темпе

Как показано на рис. 1.2, каждая группа разработчиков единолично владеет кодом и инфраструктурой своей службы. Они могут собирать, развертывать и тестировать свой код независимо друг от друга, потому что репозиторий системы управления версиями и инфраструктура (сервер приложений и база данных) теперь полностью независимы от других частей приложения. Напомним характеристики микросервисной архитектуры:

- логика приложения разбита на мелкие компоненты с четко определенными согласованными границами ответственности;
- каждый компонент отвечает за узкий круг задач и развертывается независимо от других; один микросервис отвечает за одну часть предметной области;
- для обмена данными между собой микросервисы используют облегченные протоколы, такие как HTTP и JSON (JavaScript Object Notation – форма записи объектов JavaScript);
- приложения на основе микросервисов всегда обмениваются данными с использованием технологически нейтрального формата (чаще всего используется JSON), поэтому техническая реализация службы не имеет значения; это означает, что прило-

жение, состоящее из микросервисов, может быть написано на нескольких языках и с использованием нескольких технологий;

- микросервисы – благодаря небольшому размеру, независимому и распределенному характеру – позволяют организациям иметь небольшие группы разработчиков с четко определенными сферами ответственности. Эти группы могут работать над достижения единой цели, например над созданием приложения, но каждая несет ответственность только за те службы, над которыми они работают.

На рис. 1.3 сравниваются монолитная и микросервисная архитектуры на примере типичного небольшого приложения электронной коммерции.

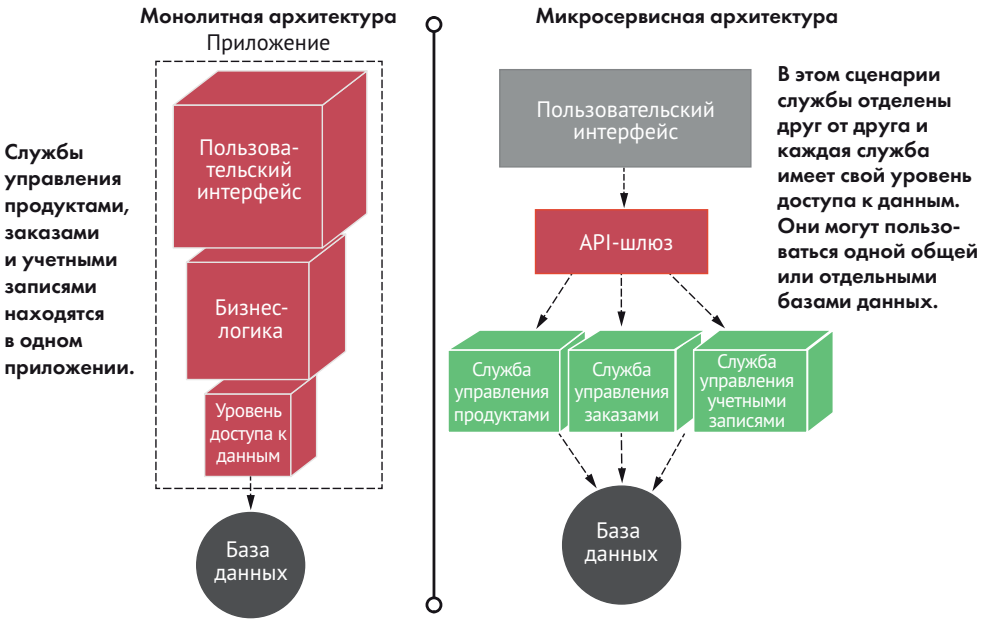


**Рис. 1.3.** Сравнение монолитной и микросервисной архитектур

### 1.1.4. Зачем менять способ создания приложений?

Компании, раньше обслуживавшие местные рынки, внезапно обнаружили, что могут использовать глобальную базу клиентов. Однако более широкой глобальной клиентской базе сопутствует мировая конкуренция. Усиление конкуренции влияет на подходы к разработке приложений. Например:

- *возросшая сложность.* Клиенты ожидают, что все подразделения организации будут знать, кто они есть. Но «изолированные» приложения, взаимодействующие с одной базой данных и не интегрирующиеся с другими приложениями, больше не яв-

ляются нормой. Современные приложения должны взаимодействовать с множеством служб и баз данных, находящихся не только внутри компании, но и в других компаниях, оказывающих интернет-услуги;

- *клиенты хотят быстрее получать обновления.* Клиенты больше не хотят ждать выхода следующей ежегодной версии программного пакета. Они ожидают, что функции в программном продукте будут разделены и новые версии будут выпускаться быстро, в течение нескольких недель (или даже дней);
- *клиентам также нужны высокая производительность и масштабируемость.* Глобальные приложения чрезвычайно затрудняют прогнозирование количества транзакций, с которым сможет справиться приложение, и когда это количество будет достигнуто. Приложения должны быстро масштабироваться в ту или иную сторону, в зависимости от объема трафика;
- *клиенты ожидают, что их приложения всегда будут доступны.* Клиенты находятся на расстоянии одного щелчка от конкурента, поэтому приложения компании должны быть очень устойчивыми. Сбои или проблемы в одной части приложения не должны приводить к прекращению работы всего приложения.

Чтобы оправдать эти ожидания, мы, как разработчики приложений, должны раскрыть тайну создания легко масштабируемых приложений с высокой степенью избыточности, которая заключается в том, чтобы разбить приложения на небольшие службы, которые можно создавать и развертывать независимо друг от друга. «Разбив» приложения на мелкие службы и переместив их из единого монолитного артефакта в распределенную среду, можно строить системы:

- *гибкие* – отдельные службы можно конструировать и реорганизовывать независимо друг от друга и быстро предоставлять новые возможности. Чем меньше единица служба, тем проще ее изменить и тем меньше времени уходит на ее тестирование и развертывание;
- *устойчивые* – приложение, разделенное на отдельные службы, больше не представляет собой единый «комок грязи», в котором выход из строя одной части приводит к сбою всего приложения. Сбои можно локализовать в небольшой части приложения и устранить до того, как приложение остановится. В случае неисправимой ошибки приложение может продолжать работать, оказывая более узкий круг услуг;
- *масштабируемые* – приложение, разделенное на отдельные службы, легко распределить по горизонтали между несколькими серверами и тем самым масштабировать функции/службы. В монолитном приложении, где вся логика взаимосвязана,

приложение должно масштабироваться целиком, даже если узким местом является лишь небольшая его часть. Масштабирование небольших служб проще и намного рентабельнее.

Учитывая вышесказанное, начнем обсуждение микросервисов. И имейте в виду следующее:

*небольшие, простые и разделенные службы = масштабируемые, устойчивые и гибкие приложения.*

Важно понимать, что системы и организации могут извлечь выгоду из использования микросервисов. Чтобы организация могла получить преимущества, можем применить *закон Конвея* в обратном порядке. Этот закон указывает несколько моментов, которые могут улучшить коммуникацию и структуру компании.

В законе Конвея (впервые был описан в апреле 1968 года Мелвином Р. Конвеем (Melvin R. Conway) в статье «How Do Committees Invent») говорится, что «организации, проектирующие системы... ограничены дизайном, который копирует структуру коммуникации в этой организации». По сути, это означает, что характер коммуникации внутри команды и между командами напрямую отражается в коде, который они создают.

Если применить закон Конвея в обратном порядке (также известный как *обратный маневр Конвея*) и спроектировать структуру компании, опираясь на микросервисную архитектуру, то коммуникация, стабильность и организационная структура приложений улучшатся, если создать слабосвязанные и автономные группы для реализации микросервисов.

## 1.2. Микросервисы со Spring

Spring стал самым популярным фреймворком разработки приложений на Java. Он основан на идее внедрения зависимостей. *Инфраструктура внедрения зависимостей* позволяет эффективнее управлять большими Java-проектами за счет оформления отношений между объектами в приложении через соглашения (и аннотации) вместо создания жестких связей. Spring располагается посередине между различными Java-классами в приложении и управляет их зависимостями. По сути, Spring позволяет собирать код подобно набору деталей конструктора лего.

Что особенно впечатляет в фреймворке Spring и является свидетельством одаренности сообщества его разработчиков, так это ее способность оставаться актуальным и изобретать себя заново. Разработчики Spring быстро заметили нарастающую тенденцию ухода от монолитных приложений, в которых логика представления приложения, бизнес-логика и логика доступа к данным упакованы вместе и развертываются как единый артефакт, и перехода к рас-

пределенным моделям, в которых небольшие службы можно быстро развернуть в облаке. В ответ на эту тенденцию разработчики Spring запустили два проекта: Spring Boot и Spring Cloud.

Spring Boot – это переосмысление фреймворка Spring. Поддерживая основные возможности Spring, Spring Boot лишился многих «корпоративных» функций, имеющихся в Spring, и вместо этого предоставляет возможность создания на Java микросервисов в архитектурном стиле REST (Representational State Transfer – передача репрезентативного состояния). С помощью нескольких простых аннотаций разработчик Java может быстро создать службу REST, упаковать и развернуть ее без использования внешнего приложения в роли контейнера.

> **ПРИМЕЧАНИЕ.** Мы более подробно рассмотрим архитектурный стиль REST в главе 3, однако уже сейчас можно отметить, что основная его идея заключается в том, что службы должны использовать HTTP-глаголы (GET, POST, PUT и DELETE) для представления своих основных действий и легковесный веб-ориентированный протокол сериализации данных, такой как JSON обмена данными.

Вот некоторые ключевые особенности Spring Boot:

- встроенный веб-сервер, помогающий избежать сложностей при развертывании приложения: Tomcat (по умолчанию), Jetty или Undertow.

  Это один из важнейших компонентов Spring Boot; выбранный веб-сервер включается в развертываемый архив JAR. Единственное, что необходимо развертываемым приложениям Spring Boot, – это установить Java на сервере;
- предопределенная конфигурация для быстрого начала работы над проектом (для начинающих);
- автоматическая настройка возможностей Spring – когда это возможно;
- широкий спектр возможностей, готовых к использованию в промышленном окружении (таких как метрики, безопасность, проверка статуса, хранение конфигурации вовне и т. д.).

Использование Spring Boot дает нашим микросервисам следующие преимущества:

- сокращает время разработки и увеличивает эффективность и производительность;
- предлагает встроенный HTTP-сервер для запуска веб-приложений;
- позволяет избавиться от большого количества шаблонного кода;

- упрощает интеграцию с экосистемой Spring (включая Spring Data, Spring Security, Spring Cloud и др.);
- предоставляет коллекцию различных плагинов для разработки.

Поскольку микросервисы стали одним из наиболее распространенных архитектурных шаблонов создания облачных приложений, сообщество разработчиков Spring создали для нас Spring Cloud – фреймворк, упрощающий развертывание микросервисов в частном или общедоступном облаке. Spring Cloud объединяет несколько популярных фреймворков микросервисов для управления облаком, что позволяет использовать и развертывать эти технологии простым аннотированием кода. Мы рассмотрим различные компоненты Spring Cloud в следующей главе.

## 1.3. Что мы будем создавать?

Эта книга предлагает пошаговое руководство по созданию полноценного приложения с микросервисной архитектурой на основе Spring Boot, Spring Cloud и других полезных и современных технологий. На рис. 1.4 показана общая структура интеграции некоторых служб и технологий, которые мы будем использовать на протяжении всей книги.

На рис. 1.4 показан клиентский запрос на обновление и получение информации об организации в микросервисной архитектуре, которую мы создадим. Чтобы послать запрос, клиент сначала должен пройти аутентификацию с помощью Keycloak и получить токен доступа. Затем клиент посылает запрос в API-шлюз Spring Cloud. Служба API-шлюза – это точка входа в нашу архитектуру; она обращается к службе обнаружения Eureka, чтобы получить местоположение служб организации и лицензий, а затем вызывает конкретный микросервис.

Получив запрос, служба организации проверяет с помощью Keycloak токен доступа на наличие соответствующих привилегий у пользователя. После проверки обновляет и извлекает информацию из базы данных организации и возвращает ее клиенту в форме HTTP-ответа. Дополнительно после обновления информации об организации служба добавляет сообщение в тему Kafka, чтобы уведомить службу лицензий об изменении.

Получив сообщение, служба лицензий Redis сохраняет конкретную информацию в своей базе данных, размещенной в оперативной памяти. На протяжении всего этого процесса архитектура использует распределенную трассировку из Zipkin, Elasticsearch и Logstash для управления журналами и экспортирует и отображает метрики приложения с помощью Spring Boot Actuator, Prometheus и Grafana.

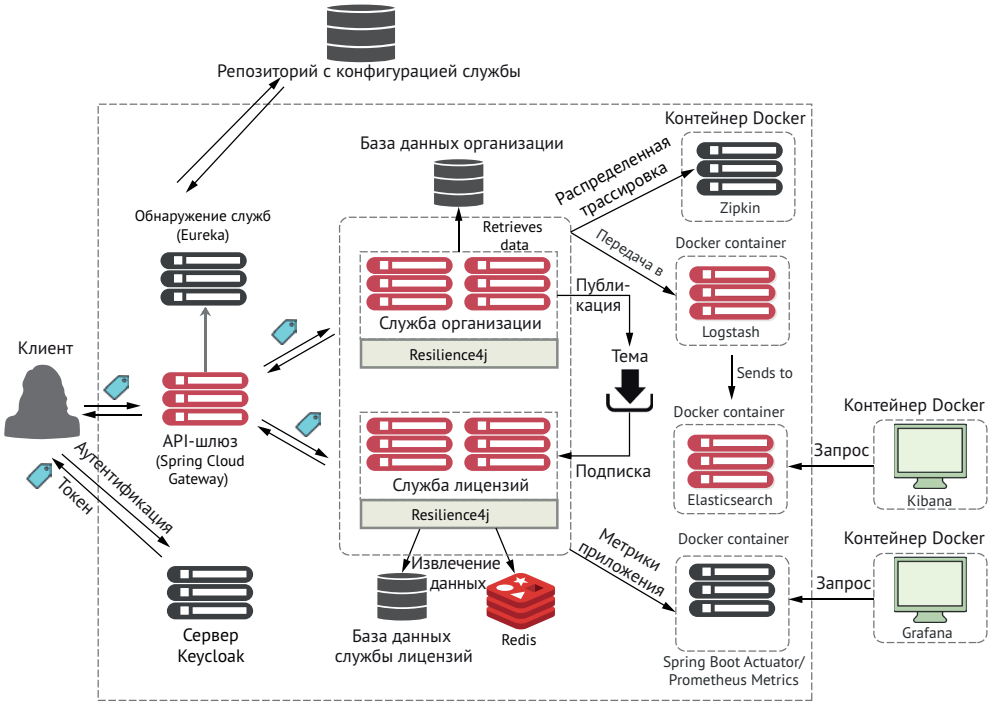


**Рис. 1.4.** Общая структура сервисов и технологий, которые мы используем в этой книге в качестве примера

Двигаясь вперед, мы познакомимся со всеми этими технологиями: Spring Boot, Spring Cloud, Elasticsearch, Logstash, Kibana, Prometheus, Grafana и Kafka и др. Они могут показаться сложными, но я буду постепенно показывать и рассказывать, как создавать и интегрировать различные компоненты, составляющие диаграмму на рис. 1.4.

## 1.4. О чем эта книга?

Эта книга рассматривает широкий круг тем. Она охватывает все, от базовых определений до сложных реализаций микросервисной архитектуры.

### 1.4.1. Что вы узнаете в этой книге

Эта книга посвящена созданию приложений на основе микросервисов с использованием проектов Spring, таких как Spring Boot и Spring Cloud, которые можно развернуть в частном облаке вашей компании или в общедоступном облаке, таком как Amazon, Google или Azure. В этой книге рассматриваются следующие темы:

- что такое микросервис, приемы и особенности проектирования, которые необходимо применять и учитывать при создании приложений на основе микросервисов;

- когда не следует создавать приложение на основе микросервисов;
- как создавать микросервисы с помощью фреймворка Spring Boot;
- основные практики поддержки приложений на основе микросервисов и особенно облачных приложений;
- что такое Docker и как его интегрировать с приложением на основе микросервисов;
- как можно использовать Spring Cloud для реализации методов эксплуатации, описанных далее в этой главе;
- как создавать метрики приложения и отображать их в инструментах мониторинга;
- как организовать распределенную трассировку с помощью Zipkin и Sleuth;
- как управлять журналами приложений с помощью стека ELK;
- как, используя полученные знания, построить конвейер развертывания, который можно использовать для развертывания служб в частном или в общедоступном облаке.

Прочитав эту книгу, вы обретете знания, необходимые для создания и развертывания микросервиса Spring Boot. Вы также познакомитесь с ключевыми архитектурными решениями, которые пригодятся вам для ввода в действие ваших микросервисов. Вы узнаете, как можно комбинировать управление конфигурацией служб, обнаружение служб, обмен сообщениями, журналирование и трассировку, а также безопасность для создания надежной среды микросервисов. Наконец, вы увидите, как можно развертывать микросервисы с использованием различных технологий.

### 1.4.2. Почему эта книга актуальна для вас?

Я полагаю, что коль скоро вы дошли до этого момента, то это означает, что вы:

- разработчик на Java или хорошо разбираетесь в Java;
- имеете опыт использования Spring;
- желаете научиться создавать приложения на основе микросервисов;
- интересуетесь возможностью использовать микросервисы для создания облачных приложений;
- хотите узнать, подходят ли Java и Spring для создания приложений на основе микросервисов;
- хотите познакомиться с передовыми технологиями создания микросервисной архитектуры;
- хотите увидеть, как реализуется развертывание приложения на основе микросервисов в облаке.

Эта книга подробно рассказывает о реализации архитектуры микросервисов на Java. Она предоставляет описательную и визуальную информацию и множество практических примеров кода, чтобы вы могли понять, как реализовать эту архитектуру с использованием последних версий различных проектов Spring, таких как Spring Boot и Spring Cloud.

Кроме того, эта книга дает краткое введение в шаблоны проектирования микросервисов, передовой опыт и инфраструктурные технологии, которые идут рука об руку с архитектурой этого типа, моделируя реальную среду разработки приложений. Давайте на мгновение переключим наше внимание и рассмотрим создание простого микросервиса с использованием Spring Boot.

## 1.5. Облачные приложения и приложения на основе микросервисов

В этом разделе вы увидите, как создать микросервис с помощью Spring Boot, и узнаете, почему облачные окружения являются наиболее подходящими для приложений на основе микросервисов.

### 1.5.1. Создание микросервиса с помощью Spring Boot

В этом разделе я не буду подробно описывать код микросервиса, а только познакомлю с порядком создания службы, чтобы вы могли убедиться, насколько просто пользоваться Spring Boot. Для этого мы создадим простую службу REST «Hello World» с одной главной конечной точкой, реализующей глагол HTTP GET. Эта конечная точка будет получать параметры запроса и параметры URL (также известные как *переменные пути*). На рис. 1.5 показано, что будет делать эта служба и общий поток обработки запроса пользователя в микросервисе Spring Boot.

Этот пример ни в коем случае не является исчерпывающим, и он не иллюстрирует порядок создания микросервисов промышленного уровня. Его цель состоит лишь в том, чтобы показать, как мало кода требуется для его написания. Мы не будем подробно обсуждать настройку файлов сборки проекта или детали кода – всему этому мы уделим должное внимание в главе 2. Желающие увидеть файл pom.xml для Maven и фактический код смогут найти все это в репозитории книги.

**ПРИМЕЧАНИЕ.** Исходный код всех примеров из главы 1 доступен в репозитории GitHub книги по адресу https://github.com/ihuaylupo/manning-smia/tree/master/chapter1.


**Рис. 1.5.** Spring Boot абстрагирует общие задачи микросервиса REST (маршрутизация к бизнес-логике, синтаксический анализ параметров в URL, преобразование JSON в объекты Java и обратно) и позволяет разработчику сосредоточиться на бизнес-логике службы. На этом рисунке показаны три разных способа передачи параметров нашему контроллеру

В этом примере имеется только один класс Java с именем `Application`, который находится в файле com/huaylupo/spmia/ch01/SimpleApplication/Application.java. Мы будем использовать этот класс для предоставления конечной точки REST с именем `/hello`. Определение класса `Application` показано в листинге 1.1.

**Листинг 1.1.** Служба Hello World с использованием Spring Boot: (очень) простой микросервис Spring

**Сообщает Spring Boot, что этот класс является точкой входа для службы Spring Boot.**

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
@RequestMapping(value="hello")
public class Application {
```

**Сообщает Spring Boot, что код в этом классе должен экспортироваться как Spring RestController.**

**Все URL-адреса, поддерживаемые этим приложением, начинаются с префикса /hello.**

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

    @GetMapping(value="/{firstName}")
    public String helloGET(
        @PathVariable("firstName") String firstName,
        @RequestParam("lastName") String lastName) {
        return String.format(
               "{\"message\":\"Hello %s %s\"}",
                 firstName, lastName);
    }
}

class HelloRequest{

    private String firstName;
    private String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

**Экспортирует как конечную точку REST GET, которая принимает два параметра: firstName (через @PathVariable) и lastName (через @RequestParam).**

**Параметры firstName и lastName отображаются в две переменные, которые передаются в функцию hello.**

**Возвращает простую строку JSON, которую мы создали вручную (в главе 2 мы не будем создавать данных в формате JSON).**

**Содержит поля структуры JSON, отправленной пользователем.**

В листинге 1.1 экспортируется единственная конечная точка, обрабатывающая HTTP-запросы GET, которая принимает два параметра (`firstName` и `lastName`) в URL: один из переменной пути (`@PathVariable`), а другой из параметра запроса (`@RequestParam`). Конечная точка возвращает простую строку в формате JSON, содержащем сообщение `"Hello firstName lastName"`. Запрос GET к конечной точке `/hello/illary?LastName=huaylupo` к этой службе вернет:

```
{“message”:”Hello illary huaylupo”}
```

Давайте запустим приложение Spring Boot. Для этого выполним следующую команду в командной строке. Это команда Maven, она использует плагин Spring Boot, как указано в файле pom.xml, для запуска приложения с использованием встроенного сервера Tomcat. Когда вы выполните команду `mvn spring-boot:run` и приложе-

ние запустится, в окне терминала должны появиться строки, как показано на рис. 1.6.

```
mvn spring-boot:run
```

**ПРИМЕЧАНИЕ.** Перед запуском команды в терминале перейдите в корневой каталог проекта. Корневой каталог проекта – это каталог, в котором находится файл pom.xml. В противном случае вы столкнетесь с ошибкой: «No plugin found for prefix 'spring-boot' in the current project and in the plugin groups» (Не найден плагин для префикса 'spring-boot' в текущем проекте и в группах плагинов).

### Java против Groovy и Maven против Gradle

Платформа Spring Boot поддерживает языки программирования Java и Groovy. Также Spring Boot поддерживает инструменты сборки Maven и Gradle. Gradle – это предметно-ориентированный язык (Domain Specific Language, DSL) на основе Groovy, и он используется для описания конфигурации проекта вместо XML, как в Maven. Язык Gradle – мощный, гибкий и пользуется большой популярностью, но сообщество разработчиков на Java по-прежнему использует Maven. Поэтому, чтобы не терять нити рассуждений и охватить максимально широкую аудиторию, все примеры в этой книге приводятся только для Maven.

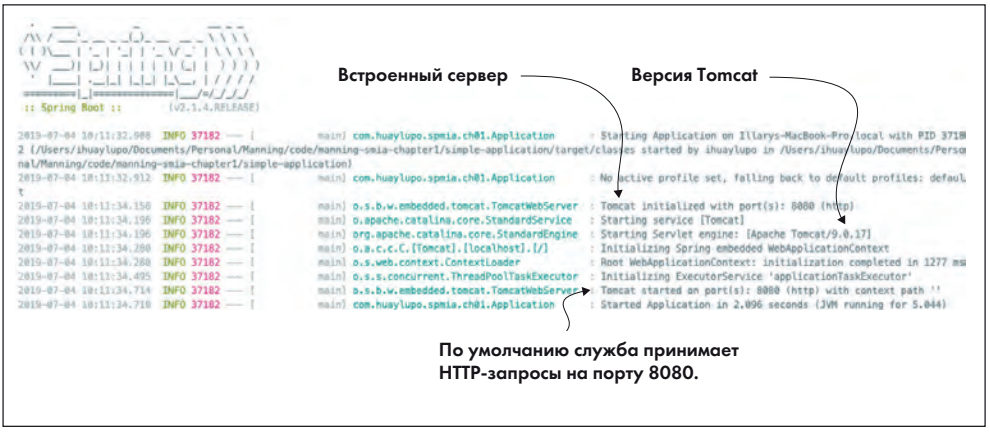


**Рис. 1.6.** Служба Spring Boot сообщает номер поддерживаемого порта в консоли

Чтобы вызвать службу, нужен инструмент REST. Для обращения к службам REST существует множество инструментов как с графическим интерфейсом, так и с интерфейсом командной строки. В этой книге мы будем использовать Postman (https://www.getpostman.com/). На рис. 1.7 и 1.8 показаны два примера отправки разных запросов из Postman к конечным точкам и полученные в ответ результаты.

На рис. 1.8 показано, как отправить HTTP-запрос POST. В данном случае этот запрос был отправлен только в демонстрационных целях. В следующих главах вы увидите, что метод POST предпочтительнее, когда служба должна создавать новые записи.

Этот простой пример кода не демонстрирует ни всех возможностей Spring Boot, ни передовых методов создания служб. Зато он показывает, что для создания на Java полноценной службы HTTP JSON REST с маршрутизацией по URL и параметрам достаточно написать всего несколько строк кода. Java – мощный язык, но он приобрел репутацию чересчур многословного, по сравнению с другими языками. Однако благодаря Spring желаемого результата можно достичь всего с несколькими строками кода. А теперь давайте рассмотрим, в каких случаях предпочтительнее использовать микросервисы для создания приложений.

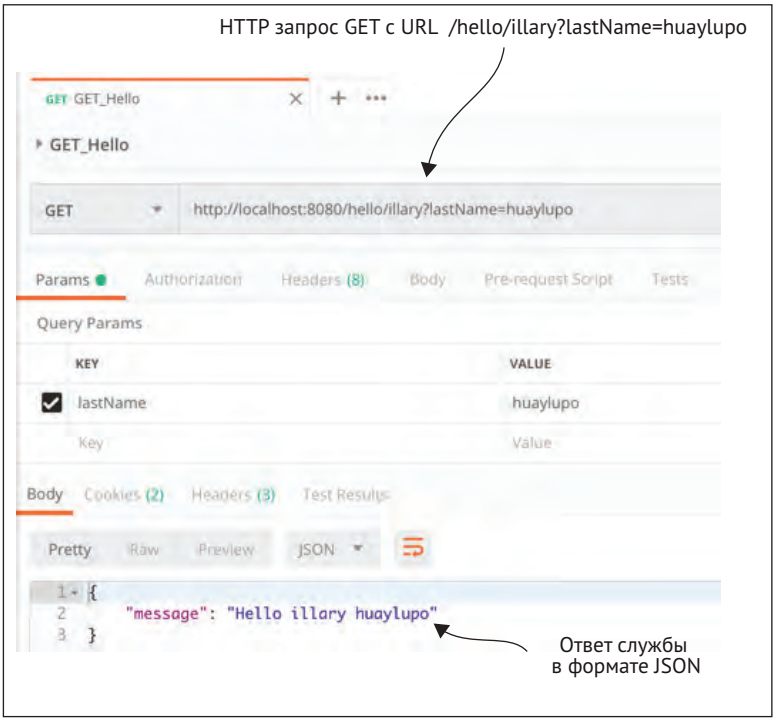


**Рис. 1.7.** HTTP-запрос GET к конечной точке /hello и полученный ответ в формате JSON

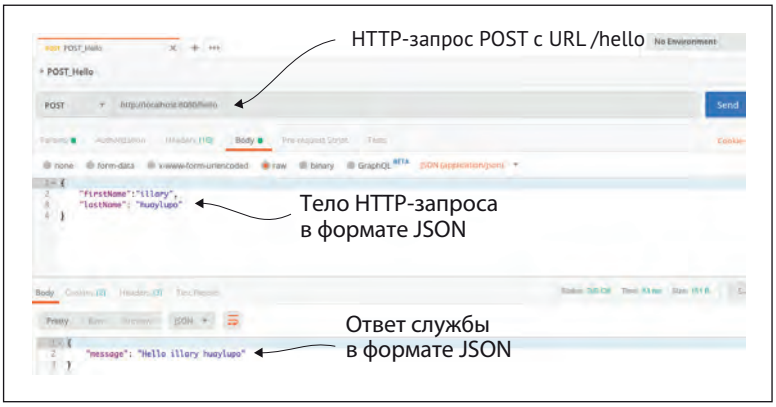


**Рис. 1.8** HTTP-запрос POST к конечной точке /hello с телом запроса и полученным ответом в формате JSON

### 1.5.2. Что такое облачные вычисления?

Облачные вычисления – это оказание вычислительных и виртуализированных ИТ-услуг – баз данных, сетей, программных продуктов, серверов, аналитики и много другого – через интернет для предоставления гибкого, безопасного и простого в использовании окружения. Облачные вычисления дают значительные выгоды компаниям, такие как низкие начальные инвестиции, простота использования и обслуживания, а также масштабируемость.

Модели облачных вычислений позволяют пользователю выбирать уровень контроля над информацией и услугами, которые они предоставляют. Эти модели известны своими аббревиатурами обычно в формате *XaaS*, обозначающими *нечто как услуга (anything as a service)*. Ниже перечислены наиболее распространенные модели облачных вычислений, а на рис. 1.9 показаны различия между этими моделями.

- *Инфраструктура как услуга (Infrastructure as a Service, IaaS)*. Поставщик предоставляет инфраструктуру, которая позволяет получать доступ к вычислительным ресурсам, таким как серверы, хранилища и сети. В этой модели пользователь сам отвечает за все, что связано с обслуживанием инфраструктуры и масштабированием приложений.

  Примерами платформ IaaS могут служить AWS (EC2), Azure Virtual Machines, Google Compute Engine и Kubernetes.

- *Контейнер как услуга (Container as a Service, CaaS)*. Промежуточная модель между IaaS и PaaS. Относится к форме виртуализации на основе контейнеров. В отличие от модели IaaS, где разработчик имеет полный контроль над виртуальной машиной, где развертывается служба, в модели CaaS вам предоставляется легковесный переносимый виртуальный контейнер (например, Docker). Поставщик облачных услуг запускает вирту-

альный сервер, на котором выполняется контейнер, а также имеются инструменты для создания, развертывания, мониторинга и масштабирования контейнеров.

Примерами платформ CaaS могут служить Google Container Engine (GKE) и Amazon Elastic Container Service (ECS). В главе 11 вы увидите, как разворачивать свои микросервисы в Amazon ECS.

- *Платформа как услуга (Platform as a Service, PaaS).* Эта модель предоставляет платформу и окружение, которые позволяют пользователям сосредоточиться на разработке, выполнении и обслуживании приложения. Приложения могут создаваться с помощью инструментов, предоставляемых поставщиком услуги (например, операционная система, системы управления базами данных, техническая поддержка, хранилище, хостинг, сеть и т. д.). Пользователям не нужно вкладывать средства в физическую инфраструктуру или тратить время на управление ею, что позволяет им сосредоточиться исключительно на разработке приложений.

  Примерами платформ PaaS могут служить Google App Engine, Cloud Foundry, Heroku и AWS Elastic Beanstalk.

- *Функция как услуга (Function as a Service, FaaS).* Также известна как бессерверная архитектура – несмотря на такое интересное название, эта архитектура не означает, что код выполняется без сервера. Эта модель предлагает возможность выполнения функций в облаке, которым поставщик услуги предоставляет все необходимые серверы. Бессерверная архитектура позволяет сосредоточиться исключительно на разработке служб, не беспокоясь о масштабировании, инициализации и администрировании серверов. То есть мы можем сосредоточиться только на наших функциях, не заботясь о какой-либо административной инфраструктуре.

  Примерами платформ FaaS могут служить AWS (Lambda), Google Cloud Function и Azure Functions.

- *Программное обеспечение как услуга (Software as a Service, SaaS).* Эта модель, также известная как программное обеспечение по запросу, позволяет пользователям использовать конкретное приложение без необходимости развертывать или обслуживать его. В большинстве случаев доступ к таким приложениям осуществляется через веб-браузер. Управление приложением, данными, операционной системой, виртуализацией, серверами, хранилищем и сетью осуществляет поставщик услуг. Пользователь просто приобретает услугу и использует программное обеспечение.

  Примерами платформ SaaS могут служить Salesforce, SAP и Google Business.

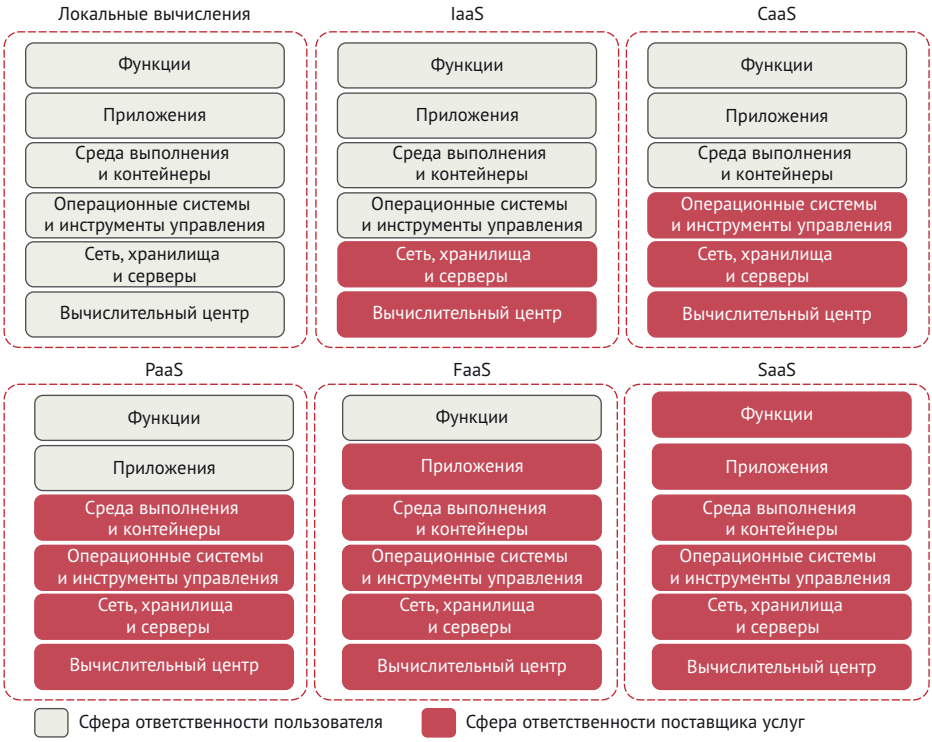


**Рис. 1.9.** Различия между моделями облачных вычислений сводятся к разграничению сфер ответственности между пользователем и поставщиком облачных услуг

> **ПРИМЕЧАНИЕ.** Без должной осмотрительности при использовании платформы на основе FaaS можно оказаться привязанным к конкретному поставщику услуг облачных вычислений, потому что в таких случаях код развертывается в среде выполнения, зависящей от поставщика. Модель FaaS позволяет писать службы на распространенных языках программирования (Java, Python, JavaScript и т. д.), но вынуждает использовать конкретный базовый API поставщика услуг и движок времени выполнения.

### 1.5.3. В чем преимущества облачных вычислений и микросервисов?

Одна из основных идей микросервисной архитектуры заключается в том, что каждая служба упаковывается и развертывается как отдельный и независимый артефакт. Экземпляры служб должны быстро запускаться, и каждый должен быть неотличим от другого. При разработке микросервиса вам рано или поздно придется решить, в каком из следующих источников будет развернута служба.

- *Физический сервер.* Можно создавать и развертывать микросервисы на физических машинах, но немногие организации делают это из-за ограничений, свойственных физическим серверам. Быстро увеличить емкость физического сервера невозможно, а горизонтальное масштабирование микросервиса на нескольких физических серверах может стать чрезвычайно дорогостоящим делом.
- *Образы виртуальных машин.* Одно из главных преимуществ микросервисов – возможность быстро запускать и останавливать экземпляры в ответ на события масштабирования и сбои. Виртуальные машины (ВМ) – это сердце и душа ведущих поставщиков облачных услуг.
- *Виртуальный контейнер.* Виртуальные контейнеры являются естественным продолжением образов виртуальных машин. В этом сценарии службы развертываются не в полноценной виртуальной машине, а в контейнерах Docker (или аналогичной контейнерной технологии) в облаке. Виртуальные контейнеры работают внутри виртуальной машины. Контейнерные технологии позволяют разделить одну виртуальную машину на серию автономных процессов, совместно использующих один и тот же образ. Микросервис можно упаковать в образ контейнера, а затем быстро развернуть и запустить несколько его экземпляров в частном или общедоступном облаке IaaS.

Преимущество облачных микросервисов заключается в таком важном их свойстве, как *эластичность*. Поставщики облачных услуг способны быстро запускать новые виртуальные машины и контейнеры за считанные минуты. Если потребности в вычислительной мощности для ваших служб упадут, то вы можете уменьшить количество действующих контейнеров, чтобы избежать дополнительных затрат. Использование услуг облачных вычислений для развертывания микросервисов дает вашим приложениям значительно большую горизонтальную масштабируемость (добавление дополнительных серверов и экземпляров служб).

Эластичность сервера также означает увеличение устойчивости ваших приложений. Если в одном из ваших микросервисов возникнут проблемы и произойдет сбой, то запуск новых экземпляров службы может обеспечить поддержание приложения в работоспособном состоянии достаточно долго, чтобы ваша группа разработчиков могла решить проблему.

В этой книге все микросервисы и соответствующая инфраструктура будут развертываться в облаке на основе CaaS с использованием контейнеров Docker. Это обычная топология развертывания микросервисов. Вот наиболее типичные характеристики облачных услуг CaaS.

- *Упрощенное управление инфраструктурой.* Поставщики облачных услуг CaaS дают вам возможность контролировать работу своих служб и запускать и останавливать службы с помощью простых вызовов API.
- *Значительная горизонтальная масштабируемость.* Поставщики облачных услуг CaaS позволяют быстро запустить один или несколько экземпляров службы. Эта означает, что вы можете быстро масштабировать службы и направлять трафик в обход неисправных серверов.
- *Высокая избыточность за счет географического распределения.* Провайдеры CaaS имеют несколько центров обработки данных. Развертывая свои микросервисы на платформе CaaS, вы можете получить более высокий уровень избыточности, чем может дать использование кластеров в одном центре обработки данных.

### Почему использование PaaS для микросервисов – плохая идея?

Выше в этой главе было представлено пять типов облачных платформ: инфраструктура как услуга (IaaS), контейнер как услуга (CaaS), платформа как услуга (PaaS), функция как услуга (FaaS) и программное обеспечение как услуга (SaaS). В этой книге особое внимание уделяется созданию микросервисов с использованием модели CaaS. Некоторые провайдеры облачных услуг позволяют абстрагироваться от инфраструктуры развертывания микросервиса, однако в этой книге мы будем стараться сохранить независимость от выбора поставщика услуг и развертывать все части приложения (включая серверы).

Например, Cloud Foundry, AWS Elastic Beanstalk, Google App Engine и Heroku дают возможность развертывать службы, не заботясь о базовом контейнере. Они предоставляют веб-интерфейс и интерфейс командной строки (Command-Line Interface, CLI), позволяющие развернуть приложение из файла WAR или JAR. Об установке и настройке сервера приложений и соответствующего Java-контейнера позаботится сам поставщик услуг. Это удобно, но платформа каждого поставщика облачных услуг имеет свои особенности, связанные с его индивидуальным решением PaaS.

Службы, создаваемые в этой книге, будут упаковываться в контейнеры Docker; основная причина такого решения состоит в том, что Docker можно развернуть у всех ведущих поставщиков услуг облачных вычислений. В следующих главах вы узнаете, что такое Docker и как использовать эту технологию для запуска всех служб и инфраструктуры, представленных в этой книге.

## 1.6. Микросервисы – это больше чем код

Идея создания отдельных микросервисов проста и понятна, но запуск и поддержка отказоустойчивого приложения на основе микросервисов (особенно в облаке) предполагают нечто большее, чем просто написание программного кода службы. На рис. 1.10 показаны некоторые рекомендации, которые следует учитывать при создании микросервиса.


**Рис. 1.10.** Микросервисы – это не только бизнес-логика. Вам нужно подумать об окружении, в котором будут выполняться службы, и как эти службы будут масштабироваться

Чтобы получить надежную службу, необходимо учесть несколько аспектов. Давайте рассмотрим вопросы на рис. 1.10 более подробно.

- *Оптимальный размер.* Выбирайте оптимальный размер для своих микросервисов, чтобы не возложить на них слишком широкие обязанности. Помните, что выбор правильного размера служб позволяет быстро вносить изменения и снижает общий риск сбоя всего приложения.
- *Прозрачность местоположения.* Контролируйте физические детали вызова службы. В приложении на основе микросервисов несколько экземпляров службы могут быстро запускаться и завершаться.

- *Устойчивость.* Защищайте пользователей ваших микросервисов и приложение в целом, организуя маршрутизацию в обход отказавших служб и используя решения «отказоустойчивости».
- *Повторяемость.* Гарантируйте использование одной и той же версии кода и конфигурации при запуске каждого нового экземпляра вашей службы.
- *Масштабируемость.* Организуйте взаимодействия так, чтобы свести к минимуму прямые зависимости между вашими службами и гарантировать возможность плавного масштабирования своих микросервисов.

Для учета всех перечисленных рекомендаций в этой книге применяется подход, основанный на шаблонах. Используя этот подход, мы рассмотрим общие решения с применением разных технологий. Несмотря на то что для реализации шаблонов в этой книге мы решили использовать Spring Boot и Spring Cloud, ничто не помешает вам взять идеи, представленные здесь, и воплотить их на других технологических платформах. В частности, мы рассмотрим следующие шаблоны проектирования микросервисов:

- базовый шаблон разработки;
- шаблоны устойчивости клиентов;
- шаблоны журналирования и трассировки;
- шаблон сборки и развертывания;
- шаблоны маршрутизации;
- шаблоны безопасности;
- шаблоны сбора метрик приложений.

Важно понимать, что нет формального свода правил создания микросервисов. В следующем разделе мы рассмотрим список общих аспектов, которые следует учитывать при создании микросервиса.

## 1.7. Базовый шаблон разработки микросервисов

Базовый шаблон разработки микросервисов определяет самые основы. На рис. 1.11 перечислены темы, касающиеся проектирования служб, которые мы рассмотрим далее.


**Рис. 1.11.** При проектировании микросервиса необходимо подумать о том, как эта служба будет использоваться

Следующие шаблоны (перечисленные на рис. 1.11) определяют основы создания микросервисов.

- *Степень детализации службы.* Как правильно выполнить декомпозицию предметной области на микросервисы, чтобы каждый имел соответствующий круг обязанностей? Слишком широкий круг обязанностей, включающий разные задачи, затрудняет обслуживание и изменение службы с течением времени. Слишком мелкая детализация увеличивает общую сложность приложения и превращает службу в простой уровень абстракции данных без какой-либо логики, кроме той, которая необходима для доступа к хранилищу. О детализации служб мы поговорим в главе 3.
- *Протоколы связи.* Протоколы задают порядок взаимодействий с вашей службой. Первый шаг – выбор характера протокола, синхронный или асинхронный. Для синхронных взаимодействий чаще всего используется REST на основе HTTP с использованием XML (Extensible Markup Language – рас-

ширяемый язык разметки), JSON (JavaScript Object Notation – форма записи объектов JavaScript) или двоичных форматов, таких как Thrift, для обмена данными. Для асинхронных взаимодействий наиболее популярным выбором является AMQP (Advanced Message Queuing Protocol – расширенный протокол организации очередей сообщений), использующий очереди (для взаимодействий типа точка–точка) или темы (для взаимодействий типа публикация–подписка) с брокерами сообщений, такими как RabbitMQ, Apache Kafka и Amazon Simple Queue Service (SQS). С протоколами связи мы будем знакомиться в последующих главах.

- *Архитектура интерфейса.* Как лучше всего организовать интерфейс, который другие разработчики будут использовать для вызова вашей службы? Как структурировать оказываемые услуги? Какие приемы существуют? Подробнее методы и практики организации интерфейса рассматриваются в последующих главах.
- *Управление конфигурацией службы.* Как организовать управление конфигурацией микросервиса так, чтобы ее можно было перемещать между разными окружениями в облаке? Управление конфигурацией с использованием внешних служб и профилей мы рассмотрим в главе 5.
- *Обработка событий между службами.* Как с помощью событий минимизировать количество жестких зависимостей между службами и повысить отказоустойчивость приложения? Мы реализуем управляемую событиями архитектуру с использованием Spring Cloud Stream в главе 10.

## 1.8. Шаблоны маршрутизации

Шаблоны маршрутизации микросервисов определяют, как клиентское приложение обнаруживает местонахождение службы и отправляет ей запросы. В облачном приложении могут быть запущены сотни экземпляров микросервиса. Чтобы обеспечить соблюдение политик безопасности и выбора контента, необходимо абстрагировать физический IP-адрес служб и организовать единую точку входа. Решить эту задачу помогают следующие шаблоны.

- *Обнаружение служб.* С помощью функции обнаружения служб и ее ключевого компонента – реестра служб – можно сделать микросервис доступным для обнаружения клиентскими приложениями без жесткого определения местоположения службы в их коде. Как? Мы выясним это в главе 6. Помните, что механизм обнаружения служб – это внутренняя служба, а не служба, ориентированная на клиентов.

Обратите внимание, что в этой книге мы используем Netflix Eureka Service Discovery, но есть и другие реестры служб, такие как etcd, Consul и Apache Zookeeper. Кроме того, в некоторых системах вообще нет явного реестра служб. Вместо него они используют коммуникационную инфраструктуру, известную как *сервисная сетка (service mesh)*.

- *Маршрутизация служб.* С помощью API-шлюза в приложениях на основе микросервисов можно организовать единую точку входа для всех служб и обеспечить единообразное применение политик безопасности и правил маршрутизации к нескольким службам и их экземплярам. Как? С помощью Spring Cloud API Gateway, как будет рассказываться в главе 8.

На рис. 1.12 показано, что обнаружение и маршрутизация служб выполняются в четко определенной последовательности (сначала выполняется маршрутизация, а затем обнаружение службы). Однако эти две модели не зависят друг от друга. Например, можно реализовать обнаружение служб без маршрутизации и точно так же можно реализовать маршрутизацию служб без обнаружения (хотя реализация при этом усложнится).

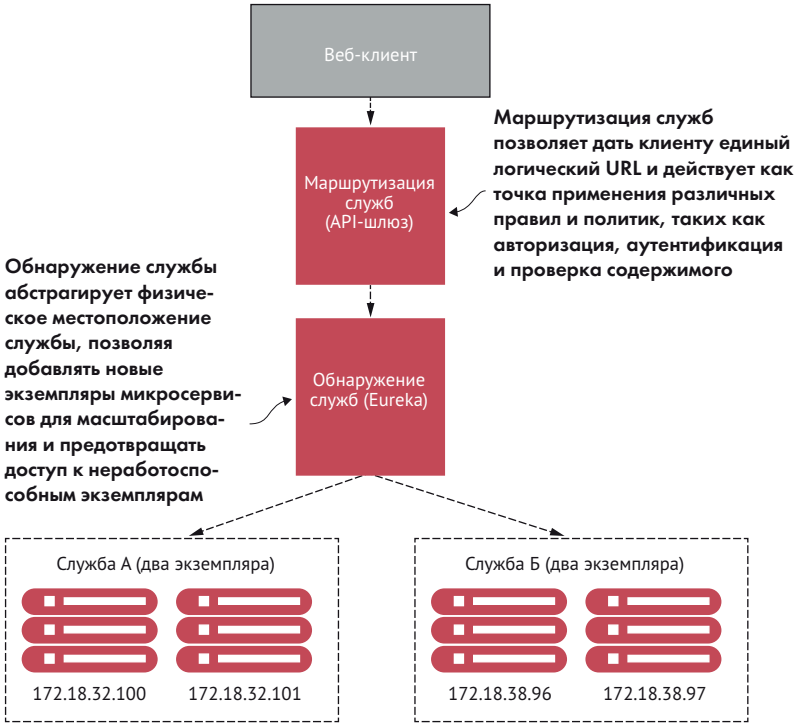


**Рис. 1.12.** Обнаружение и маршрутизация служб – ключевые элементы любого крупномасштабного приложения на основе микросервисов

## 1.9. Устойчивость клиентов

Микросервисные архитектуры по определению имеют распределенный характер, поэтому следует очень внимательно относиться к предотвращению каскадного распространения проблем из одной службы (или экземпляра службы) к ее потребителям. С этой целью мы рассмотрим четыре модели организации устойчивости клиентов.

- *Балансировка нагрузки на стороне клиента.* Реализуется кешированием местоположений экземпляров службы и равномерным распределением трафика между всеми работоспособными экземплярами.
- *Шаблон размыкателя цепи (Circuit Breaker).* Не позволяет клиенту продолжать обращаться к службе, потерпевшей сбой или испытывающей проблемы с производительностью. Медленно работающая служба напрасно потребляет ресурсы вызывающего клиента, поэтому вызовы к такой службе должны быстро завершаться с признаком ошибки, чтобы клиент мог быстро среагировать и предпринять соответствующие действия.
- *Шаблон отката к резервной реализации (fallback).* На случай сбоя службы должен иметься «сменный» механизм, который даст возможность клиенту попытаться выполнить свою работу с помощью альтернативных средств, отличных от вызываемого микросервиса.
- *Шаблон герметичных отсеков (Bulkhead).* Приложения на основе микросервисов используют несколько распределенных ресурсов для выполнения своей работы. Этот шаблон упорядочивает вызовы к этим ресурсам так, чтобы сбой в одной службе не мог отрицательно сказаться на остальной части приложения.

На рис. 1.13 показано, как эти шаблоны защищают клиента от некорректного поведения службы. Эти шаблоны рассматриваются в главе 7.

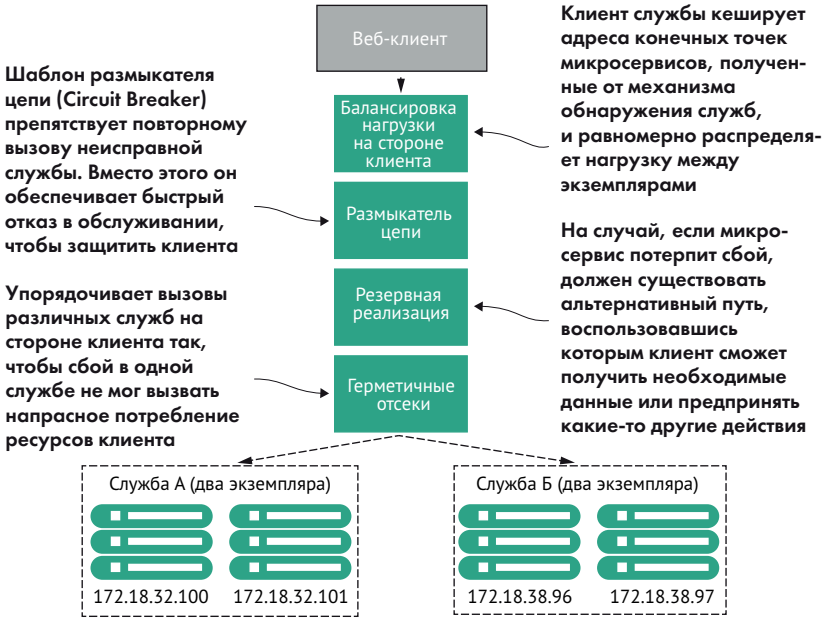


**Рис. 1.13.** В приложениях на основе микросервисов вы должны защитить клиента от некорректного поведения службы. Помните, что медленная или неработающая служба может вызвать сбои, выходящие за рамки самой службы

## 1.10. Шаблоны безопасности

Чтобы гарантировать защиту микросервисов от несанкционированного доступа, можно использовать следующие шаблоны безопасности, разрешающие выполнять запросы только обладателям надлежащих учетных данных. На рис. 1.14 показано, как можно реализовать эти три шаблона для организации службы аутентификации, которая защитит ваши микросервисы.

- *Аутентификация.* Помогает определить, что клиент, вызывающий службу, является тем, кем за кого себя выдает.
- *Авторизация.* Определяет, какие действия разрешено выполнять клиенту службы.
- *Управление учетными данными и их распространение.* Предотвращает необходимость постоянного предоставления клиентом своих учетных данных при обращении к службе. Добиться этого можно с использованием стандартов безопасности на основе токенов, таких как OAuth2 и JSON Web Tokens (JWT), суть которых заключается в получении токена, который можно включать в запросы к службе для аутентификации и авторизации пользователя.

### Что такое OAuth 2.0?

OAuth2 – это инфраструктура безопасности на основе токенов, которая позволяет пользователю аутентифицировать себя с помощью сторонней службы аутентификации. После успешной аутентификации пользователь получает токен, который *должен* пересылаться службе с каждым запросом.

Основная цель OAuth2 заключается в том, чтобы при вызове нескольких служб для выполнения запроса пользователя каждая из этих служб могла аутентифицировать пользователя без предоставления его учетных данных. Мы будем рассматривать OAuth в главе 9, однако я советую дополнительно ознакомиться с документацией по OAuth 2.0 (https://www.oauth.com/), написанной Аароном Пареки (Aaron Parecki).


**Рис. 1.14.** Используя схему безопасности на основе токенов, можно реализовать аутентификацию и авторизацию клиента без передачи его учетных данных

## 1.11. Шаблоны журналирования и трассировки

Обратной стороной архитектуры микросервисов является сложность отладки, трассировки и выявления проблем, потому что одно простое действие может породить многочисленные вызовы к микросервисам в приложении. В следующих главах мы расскажем, как реализовать распределенную трассировку с помощью Spring Cloud Sleuth, Zipkin и ELK Stack. При этом мы познакомимся с тремя основными шаблонами журналирования и трассировки.

- *Корреляция журналов*. Этот шаблон определяет порядок связывания воедино всех журналов, создаваемых службами для трассировки одной пользовательской транзакции. На примере этого шаблона мы рассмотрим реализацию идентификатора корреляции, который передается во все вызовы служб, участвующие в транзакции, и может использоваться для связывания записей в журналах, созданных службами.
- *Агрегирование журналов*. На примере этого шаблона мы посмотрим, как собрать воедино все журналы, созданные микросервисами (и их отдельными экземплярами), в единую базу данных, чтобы потом оценить характеристики производительности служб, вовлеченных в транзакцию.
- *Трассировка микросервисов*. На примере этого шаблона мы посмотрим, как визуализировать поток операций в клиентской транзакции, выполняемых всеми задействованными службами, чтобы оценить характеристики их производительности.

На рис. 1.15 показано, как эти шаблоны сочетаются друг с другом. Мы рассмотрим шаблоны журналирования и трассировки в главе 11.


**Рис. 1.15.** Хорошо продуманная стратегия журналирования и трассировки позволяет управлять отладкой транзакций в нескольких службах

## 1.12. Шаблон сбора метрик приложения

Шаблон сбора метрик описывает порядок извлечения метрик для мониторинга приложения и формирования предупреждений о возможных сбоях. Он определяет, как служба метрик осуществляет получение (извлечение), хранение и запрос бизнес-данных с целью предотвращения потенциальных проблем с производительностью в наших службах. Этот шаблон включает три основных компонента.

- *Метрики*. Фрагменты важной информации, описывающие состояние приложения и как эту информацию получить.
- *Служба метрик*. Запрашивает метрики у приложения и хранит их.
- *Пакет визуализации метрик*. Визуализирует данные, характеризующие состояние приложения и инфраструктуры.

Как показано на рис. 1.16, метрики, генерируемые микросервисами, в значительной степени зависят от службы метрик и пакета визуализации. Бессмысленно генерировать метрики, отражающие бесконечный поток информации, если нет возможности отобразить и проанализировать эту информацию. Метрики могут пересылаться микросервисом автоматически или извлекаться по запросу службы метрик:

- в первом случае экземпляр микросервиса вызывает API службы метрик и передает ему данные о приложении;
- во втором случае служба метрик сама посылает запросы микросервисам для выборки данных о приложении.

Важно понимать, что метрики являются важным элементом микросервисных архитектур, и требования к мониторингу в этих архитектурах обычно выше, чем в монолитных структурах, из-за их распределенного характера.

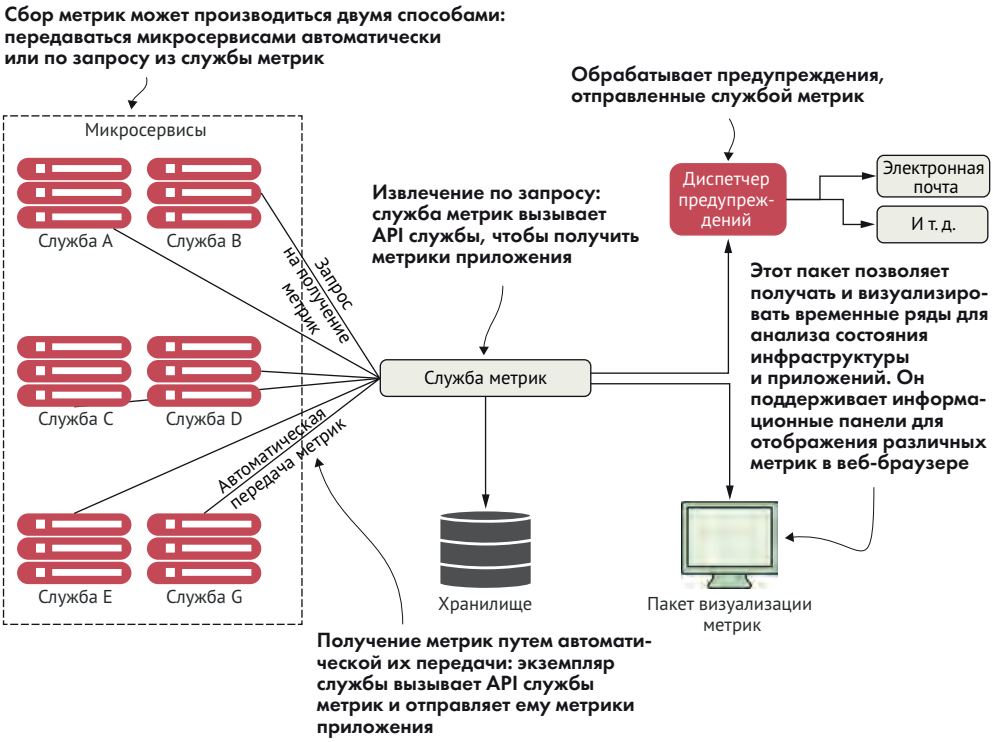


**Рис. 1.16.** Метрики извлекаются из микросервисов или передаются ими автоматически, собираются и хранятся в службе метрик, откуда потом они могут быть получены пакетом визуализации и инструментами управления предупреждениями

## 1.13. Шаблоны сборки/развертывания микросервисов

Одно из основных требований архитектуры микросервисов заключается в том, что каждый экземпляр микросервиса должен быть идентичен всем другим его экземплярам. Нельзя допускать дрейфа конфигурации (когда что-то меняется в ней после развертывания экземпляра), потому что это может привести к нестабильному поведению приложения.

Цель этого шаблона – интегрировать конфигурацию инфраструктуры прямо в процесс сборки/развертывания, чтобы вам больше не приходилось развертывать программные артефакты, такие как файлы WAR или EAR, в уже работающую часть инфраструктуры. Вместо этого процесс сборки должен создавать и компилировать микросервис и образ виртуального сервера для его выполнения. А процесс развертывания микросервиса должен развертывать образ всей машины с запущенным на ней сервером.

Рисунок 1.17 иллюстрирует эту идею. В конце книги мы покажем, как создать конвейер сборки/развертывания. В главе 12 мы рассмотрим следующие шаблоны и темы.

- *Конвейеры сборки и развертывания.* Помогает определить повторяемый процесс сборки и развертывания, в котором упор делается на сборку одной кнопкой и развертывание в любом окружении в вашей организации.
- *Инфраструктура как код.* Определяет отношение к службам как к коду, который может выполняться и управляться с помощью системы управления версиями.
- *Неизменяемые серверы.* Создание образа микросервиса позволяет гарантировать, что он никогда не изменится после развертывания.
- *Серверы Phoenix.* Гарантирует регулярное отключение и воссоздание серверов, на которых запущены отдельные контейнеры, из неизменяемого образа. Чем дольше работает сервер, тем вероятней появление отклонений от стандартной конфигурации. Дрейф конфигурации может возникнуть, если специальные изменения в конфигурации не фиксируются в системе управления версиями.

Цель этих шаблонов – выявление и безжалостное искоренение дрейфа конфигурации еще до того, как он поразит промышленное окружение.

> **ПРИМЕЧАНИЕ.** В примерах кода в этой книге (кроме главы 12) все действия выполняются локально на настольном компьютере. Примеры из первых глав можно запускать непосредственно из командной строки. Но, начиная с главы 3, весь код должен компилироваться и запускаться в контейнерах Docker.

Теперь, после краткого знакомства с шаблонами, которые мы будем использовать на протяжении всей книги, продолжим их обсуждение во второй главе. В следующей главе мы расскажем о технологиях Spring Cloud и о некоторых передовых методах разработки приложений, ориентированных на облачные микросервисы, а также сделаем первые шаги на пути к созданию нашего первого микросервиса с использованием Spring Boot и Java.

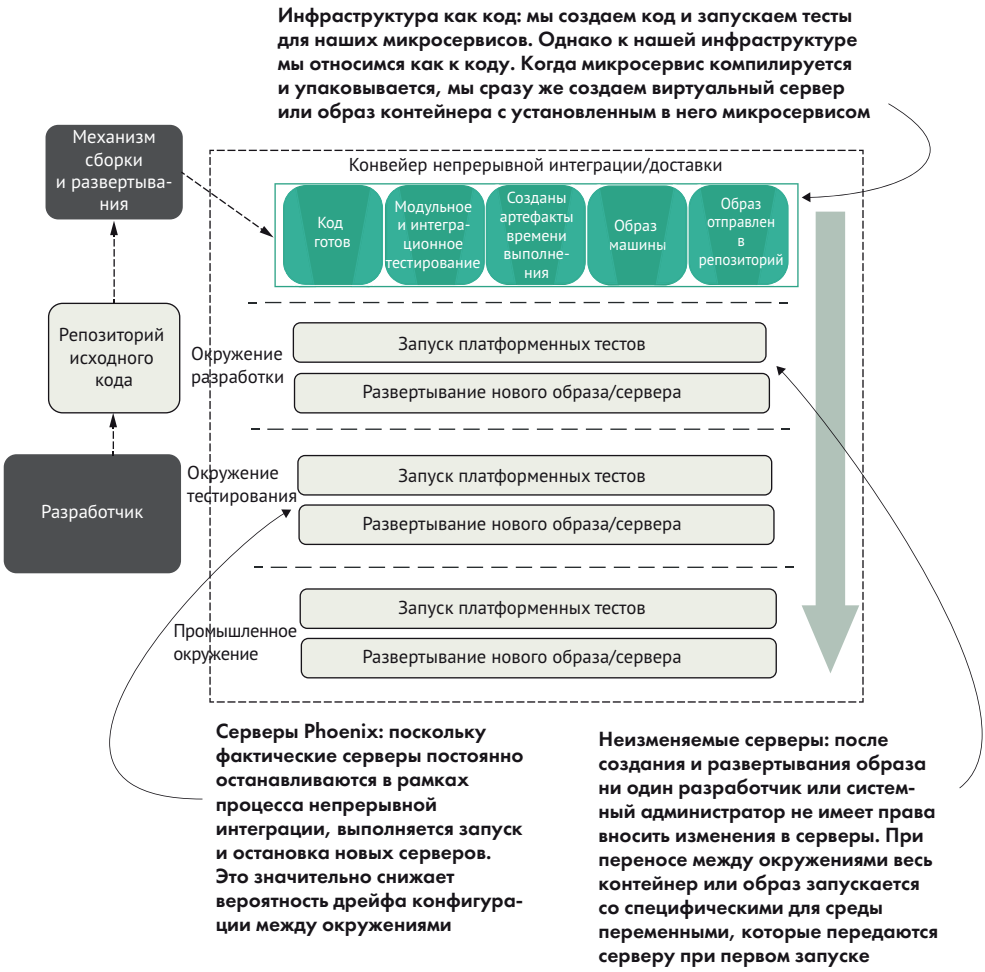


**Рис. 1.17.** Микросервис и сервер, на котором он выполняется, должны развертываться атомарно, как единый артефакт во всех окружениях

## Итоги

- В монолитных архитектурах все процессы тесно связаны и работают как единая служба.
- Микросервисы – это очень маленькие программные компоненты, отвечающие за узкий круг задач.
- Spring Boot позволяет создавать архитектуры обоих типов.
- Монолитные архитектуры обычно идеально подходят для создания простых и легковесных приложений, а микросервисные архитектуры – для сложных и постоянно развивающихся приложений. В конечном итоге выбор архитектуры, кроме прочих факторов, зависит от размера проекта, времени и требований.

- Фреймворк Spring Boot упрощает создание микросервисов на основе REST/JSON. Его цель – дать возможность быстро создавать микросервисы, написав лишь нескольких аннотаций.
- Писать микросервисы легко, но создание полноценных реализаций, пригодных для использования в промышленном окружении, требует дополнительных усилий. Существует несколько категорий шаблонов разработки микросервисов, включая базовые шаблоны разработки, шаблоны маршрутизации, устойчивости клиентов, безопасности, сбора метрик приложений и сборки/развертывания.
- Шаблоны маршрутизации определяют, как клиентское приложение обнаружит службу и обратится к ней.
- Чтобы предотвратить каскадное распространение проблемы в экземпляре службы и за ее пределы, используйте шаблоны устойчивости клиентов. К ним относятся шаблон размыкателя цепи (Circuit Breaker), предотвращающий передачу запросов отказавшей службе; шаблон отката к резервной реализации (Fallback), добавляющий альтернативные пути для получения данных или выполнения определенного действия при сбое основной службы; шаблон балансировки нагрузки на стороне клиента для масштабирования и устранения всех возможных узких мест; а также шаблон герметичных отсеков (Bulkhead) для ограничения количества одновременных вызовов службы, чтобы исключить отрицательное влияние низкопроизводительных запросов на другие службы.
- OAuth 2.0 – наиболее распространенный протокол авторизации пользователей и отличный выбор для защиты микросервисной архитектуры.
- Шаблон сборки/развертывания позволяет интегрировать конфигурацию вашей инфраструктуры прямо в процесс сборки/развертывания, чтобы вам больше не приходилось развертывать программные артефакты, такие как файлы WAR или EAR, в уже работающую часть инфраструктуры.