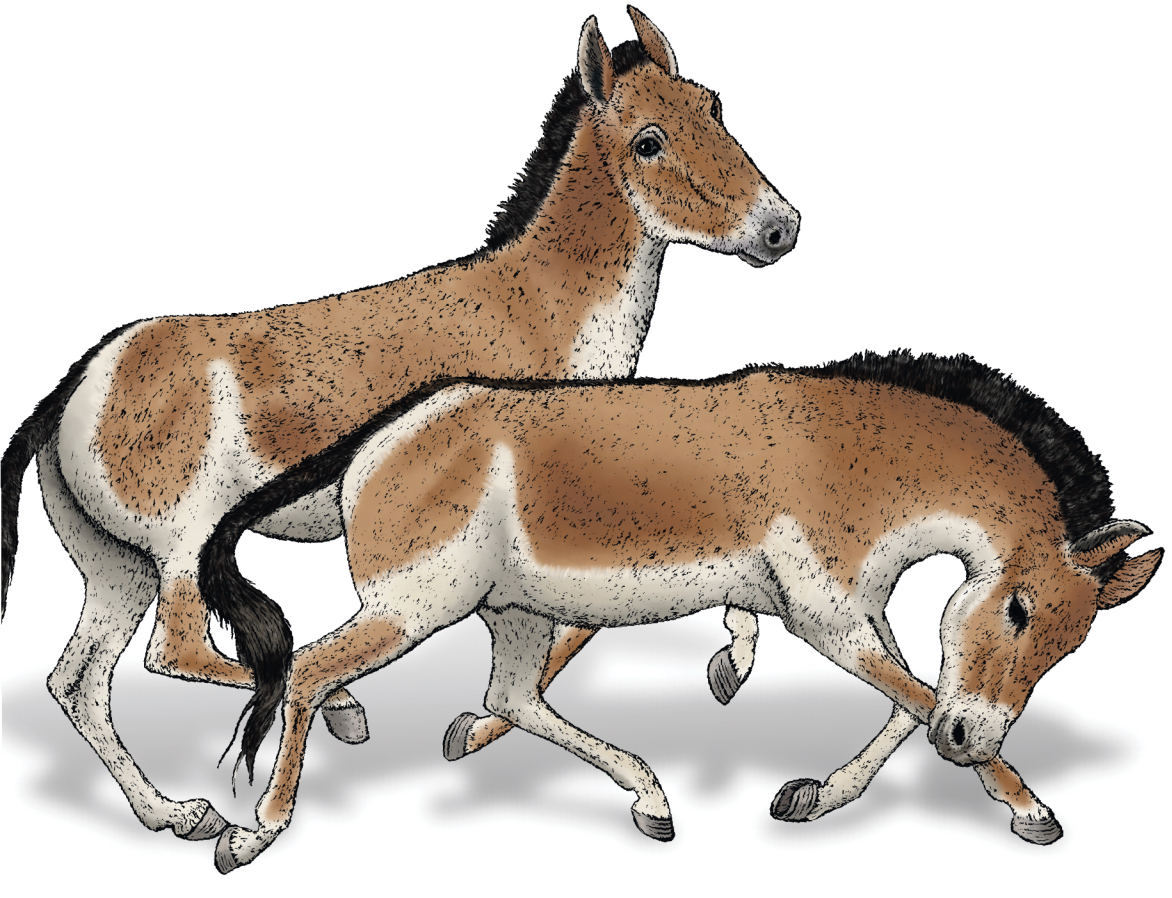
O'REILLY®

Модернизация Java Enterprise

облачные технологии для разработчиков



Маркус Эйзеле
Натале Винто

Modernizing Enterprise Java

A Concise Cloud Native Guide for Developers

Markus Eisele and Natale Vinto

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Модернизация Java Enterprise

облачные технологии для разработчиков

Маркус Эйзеле , Натале Винто

Эйзеле Маркус, Винто Натале

Э30 Модернизация Java Enterprise: облачные технологии для разработчиков. —

В разговорах о технологиях постоянно упоминаются контейнеры, микросервисы и распределенные системы, однако большинство приложений по-прежнему работают на базе монолитных архитектур, основанных на традиционных процессах разработки. Давайте поближе познакомимся с хорошо зарекомендовавшими себя моделями на основе Java и разберемся, как перенести эти монолитные приложения в будущее.

Опираясь на многолетний опыт модернизации приложений, Маркус Эйзеле и Натале Винто показывают, что необходимо сделать для обновления приложений Java, как разделить на части монолитные приложения и перейти на современный программный стек, работающий как в облаке, так и в локальной среде.

Краткое содержание

От платформы к экосистеме 12

Предисловие 14

От издательства 19

Глава 1. Обзор разработки корпоративных приложений 20

Глава 2. Путь к облачным приложениям на Java 33

Глава 3. Путешествуйте налегке 76

Глава 4. Платформа разработки программного обеспечения на основе Kubernetes 99

Глава 5. Больше чем простой перенос: работа с наследием 129

Глава 6. Создание приложений для Kubernetes 154

Глава 7. Решения завтрашнего дня: бессерверные вычисления 175

Об авторах 204

Иллюстрация на обложке 205

Оглавление

От платформы к экосистеме 12

Предисловие 14
- Условные обозначения 15
- Использование программного кода примеров 16
- Благодарности 17

От издательства 19

Глава 1. Обзор разработки корпоративных приложений 20
- От общего к частному. В чем привлекательность облаков 21
- Что означает «облачный» 23
- Разработка для Kubernetes 24
- Контейнеры и оркестрация для разработчиков 25
  - Среда выполнения контейнеров 26
  - Разновидности Kubernetes 26
- Управление сложностью разработки 27
- DevOps и гибкость 31
- Резюме 32

Глава 2. Путь к облачным приложениям на Java 33
Облачный инструментарий 34
Архитектура 34
Создание микросервиса описи товаров с помощью Quarkus 36
Создание проекта Quarkus с помощью Maven 38
Реализация предметной модели 42
Создание RESTful-сервиса 44
Запуск приложения в режиме разработки 45
Создание микросервиса Catalog с помощью Spring Boot 48
Создание проекта Spring Boot с помощью Maven 49
Реализация предметной модели 53
Создание репозитория данных 55
Создание RESTful-сервиса 56
Создание сервиса шлюза с помощью Vert.x 61
Создание проекта Vert.x Maven 63
Подготовка шлюза API 65
Создание пользовательского интерфейса с помощью Node.js и AngularJS 72
Запуск интерфейса 73
Резюме 75

Глава 3. Путешествуйте налегке 76
Трехуровневая или распределенная система 77
Технологические обновления, модернизация и трансформация 78
Концепция 6R 80
Разделяй и контейнеризируй 84
Kubernetes как новый сервер приложений 85
Определение целевой платформы 90
Обязательные шаги и инструменты миграции 94
Готовьтесь к большим делам 95
Резюме 98

Глава 4. Платформа разработки программного обеспечения на основе Kubernetes 99
Разработчики и Kubernetes 100
Что делает Kubernetes 101
Чего не делает Kubernetes 102
Инфраструктура как код 103
Образы контейнеров 104
Dockerfile 105
Сборка образов контейнеров 107
Запуск контейнеров 108
Реестр 109
Развертывание в Kubernetes 110
Под 111
Объект Service 113
Объект Deployment 115
Kubernetes и Java 119
Jib 119
JKube 122
Резюме 128

Глава 5. Больше чем простой перенос: работа с наследием 129
Управление наследием 131
Оценка приложений для миграции 131
Оценка функциональности для миграции 138
Подходы к миграции 140
Защита наследия (Replatform) 140
Создание чего-то нового (рефакторинг) 145
Проблемы 150
Избегайте двойной записи 150

Продолжительные транзакции ... 151
Слишком быстрое удаление старого кода ... 152
Особенности интеграции ... 152
Резюме ... 152

Глава 6. Создание приложений для Kubernetes ... 154
Поиск правильного баланса между масштабируемостью и сложностью ... 155
Функциональные требования к современным архитектурам ... 156
Управление через API ... 157
Обнаружение ... 157
Безопасность и авторизация ... 158
Мониторинг ... 159
Трассировка ... 160
Журналирование ... 161
CI/CD ... 162
Отладка микросервисов ... 166
Переадресация портов ... 168
Режим удаленной разработки Quarkus ... 169
Telepresence ... 170
Резюме ... 173

Глава 7. Решения завтрашнего дня: бессерверные вычисления ... 175
Что такое бессерверные вычисления ... 176
Архитектурная эволюция ... 177
Варианты использования: данные, ИИ и машинное обучение ... 179
Варианты использования: периферийные вычисления и Интернет вещей ... 180
Knative: бессерверное решение для Kubernetes ... 182

Бессерверные архитектуры, управляемые событиями 185
Функция как сервис для Java-приложений 190
Развертывание функций для приложений Java 191
Boson Function CLI (func) 192
Резюме 202

Об авторах 204

Иллюстрация на обложке 205

Моей семье. Без них ничего этого не было бы.

Маркус

Фабрицио Скарчелло, модернизатору, новатору и дорогому другу.

Натале

От платформы к экосистеме

Если вы не находились в глубоком уединении на протяжении последних нескольких лет, то не могли не заметить, что корпоративный мир переходит к облачным технологиям, к которым относятся, например, микросервисы, Kubernetes, контейнеры Linux и многое другое. Однако не сразу стало очевидно, что в этом новом облачном мире Java играет особенно важную роль, даже несмотря на то что корпоративные разработчики использовали его на протяжении более двух десятилетий.

Java, фреймворки и стеки, созданные на его основе, часто рассматриваются как монолитные, медленные, потребляющие много памяти и дискового пространства, а динамическая природа Java, по всей видимости, не согласуется с предположениями о неизменности, бытующими в Kubernetes. Для многих миллионов разработчиков на Java это может стать серьезной проблемой, особенно если потребуется воссоздать на другом языке богатство экосистемы Java, включающее интегрированные среды разработки (integrated development environment, IDE), сторонние библиотеки и т. д., которые помогали повышать продуктивность разработчиков на протяжении многих лет.

К счастью, сообщество разработчиков и производителей Java с готовностью приняло вызов облачного мира: они быстро внесли необходимые изменения в язык, фреймворки и т. д., чтобы разработчики на Java могли

использовать свои навыки на этом новом рубеже. К этим изменениям относятся такие технологии, как Quarkus, GraalVM, Eclipse Vert.x, Spring Boot и OpenJDK.

Однако эффективность их использования в облачных средах не всегда очевидна. Когда в игру вступает CI/CD? Какое место занимают образы контейнеров Linux и Kubernetes? Мониторинг, наблюдаемость, проверка работоспособности микросервисов и многое другое могут показаться чрезвычайно сложными задачами даже для самого опытного разработчика.

К счастью, авторы данной книги дают ответы на эти вопросы. Маркус и Натале рассказывают, что происходит в мире Java при переходе к облачному миру, а также об облачных технологиях, которые могут быть незнакомыми, но важными для надежной работы распределенных микросервисов. Эта книга станет отличной отправной точкой в путешествии по облачным технологиям и за их пределами как для опытных разработчиков на Java, так и для новичков!

Марк Литтл (Mark Little),
вице-президент по разработке
промежуточного программного
обеспечения, Red Hat

Предисловие

Мы написали эту книгу для разработчиков, которые хотят использовать свои монолитные модели на основе Java в будущем. Структура книги выглядит следующим образом:

- в главе 1 представлены основные технологии и идеи, которые будут обсуждаться на протяжении всей книги;
- в главе 2 рассказывается о возможности реализации микросервисной архитектуры с помощью различных фреймворков Java и о том, как разделить типичный монолит на более разнообразную и гетерогенную среду;
- в главе 3 представлен ряд основных стратегий миграции и показан порядок оценки целевой платформы разработки;
- в главе 4 обсуждается использование Java-разработчиками возможностей Kubernetes в целях модернизации и улучшения своих приложений;
- в главе 5 рассматриваются проверенные паттерны, стандартизированные инструменты и ресурсы с открытым исходным кодом, способные помочь в создании долговечных систем, которые могут развиваться и изменяться в соответствии с вашими потребностями;

- в главе 6 демонстрируются основные задачи в Kubernetes, такие как журналирование, мониторинг и отладка приложений;
- в главе 7 анализируется создание современных приложений с помощью модели бессерверных вычислений. Здесь описываются некоторые из наиболее распространенных архитектур и вариантов использования, с которыми разработчики на Java наверняка столкнутся в ближайшей перспективе.

Условные обозначения

В книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины и важные понятия.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена файлов и их расширений.

`Моноширинный жирный шрифт`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

`Моноширинный курсив`

Показывает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, элементов интерфейса, названий каталогов.

Этот рисунок указывает на совет или предложение.

Этот рисунок указывает на общее примечание.

Этот рисунок указывает на предупреждение.

Использование программного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) можно скачать по адресу https://oreil.ly/modernentjava.

Если у вас вопрос технического характера или возникла проблема при использовании примеров кода, то напишите нам: bookquestions@oreilly.com.

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Благодарности

Мы хотим поблагодарить Джейсона «Джея» Добиса (Jason “Jay” Dobies) за его страсть к изучению нашего немецкого/итальянского английского. Он не только помог нам улучшить наши писательские навыки, но и дал ценную информацию как первый читатель. Спасибо за участие в этом путешествии.

Все говорят, что писать книги — тяжелая работа. Мы знали это и раньше, но недооценили дополнительные сложности, обусловленные тем, что мы начали писать, когда нагрянула пандемия. Мы оба пережили больше взлетов и падений, чем ожидали, и только благодаря нашим семьям и друзьям у нас появилась возможность закончить эту книгу.

Мы также не можем не поблагодарить команду издательства O’Reilly за их терпение, гибкость и открытость. Спасибо вам, Сюзанна Маккуэйд (Suzanne McQuade), Николь Таше (Nicole Taché) и Амелия Блевинс (Amelia Blevins)!

Спасибо Red Hat за то, что дали замечательную работу и возможность обучаться и расти. Мы любим открытость и тоже стремимся делиться знаниями, как и все остальные работники этой компании. Спасибо коллегам за поддержку! Технические рецензенты, всякий раз задавая правильные вопросы, помогли выявить слабые места в книге и исправить их. Спасибо вам за самоотверженность и вдохновение: Себастьен Блан

(Sébastien Blanc), Алекс Сото (Alex Soto) и Марк Хильденбранд (Marc Hildenbrand)!

В основе этой книги лежит не только суммарный опыт, накопленный нами, но прежде всего пример. Изначально он был создан Маду Кулибали (Madou Coulibaly) и Алексом Грумом (Alex Groom). С его помощью они обучили многих разработчиков эффективным приемам создания облачных приложений и позволили нам использовать его в качестве основы.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

Обзор разработки корпоративных приложений

Разработка корпоративных приложений всегда была одной из самых интересных областей разработки программного обеспечения, а последнее десятилетие оказалось особенно захватывающим периодом. В 2010-х годах высокораспределенные микросервисы постепенно вытеснили классические трехуровневые архитектуры, а почти безграничные ресурсы облачной инфраструктуры обусловили устаревание тяжеловесных серверов приложений. Столкнувшись со сложностями объединения частей распределенных архитектур, многие сомневаются в том, что этот сложный мир микросервисов необходим. Реальность такова, что большинство приложений все еще остаются хорошо продуманными монолитами, которые развиваются в соответствии с традиционными принципами разработки программного обеспечения.

Однако способы развертывания и эксплуатации программного обеспечения изменились очень быстро. Мы наблюдаем, как DevOps перерастает в GitOps, расширяя обязанности разработчиков за пределы кода приложения и включая необходимую инфраструктуру. Эта книга, основанная на книге *Modern Java EE Design Patterns* Маркуса Эйзеле (Markus Eisele) (O'Reilly; https://oreil.ly/1cROz), рассматривает модернизацию в более широ-

ком смысле, чем просто модульная организация программ. Мы хотим помочь вам разобраться в различных компонентах современной платформы разработки на основе Kubernetes и понять, как создавать и поддерживать приложения на ее основе.

Цель книги — дать возможность оценить факторы успеха и движущие силы модернизации приложений и облачных архитектур. Все свое внимание мы уделим модернизации корпоративных приложений на Java, включая процесс выбора приложений, пригодных для модернизации, и обзор инструментов и методологий, помогающих управлять модернизацией. Вместо того чтобы обсуждать паттерны, мы приведем примеры, которые помогут применить ваши знания.

Мы не будем противопоставлять монолитные и распределенные приложения, а постараемся помочь вам понять, как перенести ваши приложения в облако, обойдясь минимумом затрат.

Вы можете использовать книгу как справочник и читать главы в любом порядке. Однако мы организовали материал так, что повествование начинается с описания концепций высокого уровня и заканчивается реализацией. Кроме того, мы считаем важным сначала познакомиться с различными понятиями облачных сред, а затем начинать создавать для них приложения.

От общего к частному. В чем привлекательность облаков

Различия между общедоступными, частными и гибридными облаками и многооблачными средами (мультиоблаками) когда-то легко определялись местоположением и владением. Сегодня два этих признака больше не являются единственными важными факторами классификации облаков. Начнем с более полного определения различных сред и выясним, чем они хороши.

Общедоступная облачная среда обычно создается из ресурсов, не принадлежащих конечному пользователю, и они могут перераспределяться между арендаторами. Ресурсы частной облачной среды принадлежат исключительно конечному пользователю и обычно находятся за пользовательским брандмауэром в центре обработки данных или (иногда) на локальной машине. Совокупность нескольких облачных сред с определенной степенью переносимости, оркестрации и управления рабочими нагрузками называется гибридным облаком. А совокупность несвязанных и независимых облаков обычно называют мультиоблаком. Гибридный и мультиоблачный подходы являются взаимоисключающими — нельзя иметь и то и другое одновременно, поскольку облака будут либо взаимосвязаны (гибридное облако), либо нет (мультиоблако).

Корпоративные приложения все чаще развертываются в облаках разных типов, так как предприятия стремятся повысить безопасность и производительность за счет расширенного портфеля сред. Но безопасность и производительность — лишь две из многих причин переноса рабочих нагрузок в гибридные или мультиоблачные среды. Основной мотивацией для многих людей является модель «плати только за то, что используешь». Вместо того чтобы вкладывать средства в дорогостоящее локальное оборудование, которое сложно и дорого масштабировать, можно использовать ресурсы облака и только в необходимом объеме. При этом вам не нужно вкладывать деньги в оборудование, коммунальные услуги или строительство собственного центра обработки данных. Вам даже не нужны специальные IT-команды для управления вашим облачным центром обработки данных, поскольку вы можете воспользоваться опытом сотрудников облачного провайдера.

Для разработчиков облако подразумевает самообслуживание и гибкость, избавляя от необходимости ждать развертывания среды и позволяя выбирать необходимые инфраструктурные компоненты (например, базы данных, брокеры сообщений и т. д.), что в конечном счете ускоряет цикл разработки. Помимо этих основных преимуществ, можно воспользоваться специальными функциями, доступными разработчикам в некоторых облачных средах. Так, OpenShift имеет встроенную консоль разработки,

позволяющую напрямую редактировать детали топологии приложений. Облачные IDE (например, Eclipse Che (https://www.eclipse.org/che)) предоставляют доступ к рабочим областям разработки через браузер и устраняют необходимость настраивать локальную среду для команд.

Кроме того, облачные инфраструктуры поддерживают автоматизацию процессов развертывания, позволяющую развертывать программное обеспечение в тестовых и промышленных средах одним нажатием кнопки, — обязательное требование, предъявляемое группами, использующими методики гибкой разработки и DevOps. Те, кто знаком с разработкой архитектур микросервисов, не понаслышке знают о необходимости 100%-ной автоматизации. Но она выходит далеко за рамки прикладных частей: распространяется на инфраструктуру и нижестоящие системы. В этом вам помогут Ansible (https://www.ansible.com), Helm (https://helm.sh) и операторы Kubernetes (https://oreil.ly/lhaPm). Подробнее об автоматизации мы поговорим в главе 4, а тему операторов Kubernetes затронем в главе 7.

Что означает «облачный»

Вы, наверное, слышали об *облачном* подходе к разработке приложений и сервисов, особенно после того, как в 2015 году была основана организация Cloud Native Computing Foundation (CNCF), выпустившая Kubernetes v1. Билл Уайлдер (Bill Wilder) впервые использовал термин «облачный» в своей книге *Cloud Architecture Patterns* (O'Reilly; https://oreil.ly/hmeAC). По словам Уайлдера, облачное приложение спроектировано так, чтобы в полной мере использовать преимущества облачных платформ за счет применения сервисов этих платформ и автоматического масштабирования. Уайлдер написал свою книгу в период растущего интереса к разработке и развертыванию облачных приложений. Разработчики могли выбирать из широкого круга различных общедоступных и частных платформ, включая Amazon AWS, Google Cloud, Microsoft Azure, и множества других платформ менее известных провайдеров. Но в то же время все более распространенным становилось развертывание гибридных облаков, что создавало проблемы.

Вот как CNCF (https://oreil.ly/Sadph) определяет термин «облачный»:

> Облачные технологии позволяют организациям создавать и запускать масштабируемые приложения в современных динамичных средах — в общедоступных, частных и гибридных облаках. Примерами таких технологий могут служить контейнеры, сервисные сетки, микросервисы, неизменяемая инфраструктура и декларативные API.
>
> Эти технологии обеспечивают устойчивость, управляемость и наблюдаемость слабосвязанных систем. В сочетании с надежной автоматизацией они позволяют инженерам часто вносить важные изменения, дающие предсказуемый результат, с минимальными трудозатратами.
>
> *CNCF Cloud Native Definition v1.0*

Подобными облачными технологиями являются двенадцатифакторные приложения (https://12factor.net/ru/). Соответствующий манифест определяет паттерны для создания приложений, доставляемых через облако. Эти паттерны пересекаются с паттернами облачной архитектуры Уайлдера, но двенадцатифакторная методология может применяться к приложениям, которые написаны на любом языке программирования и используют любую комбинацию вспомогательных сервисов (баз данных, очередей, кэш-памяти и т. д.).

Разработка для Kubernetes

Для разработчиков, развертывающих приложения в гибридном облаке, имеет смысл сместить акцент с понятия «облачный» на понятие «для Kubernetes». Одно из первых упоминаний «для Kubernetes» имело место еще в 2017 году. В статье в блоге Medium описываются различия между понятиями «для Kubernetes» (https://oreil.ly/2quU8) и «облачный» как обусловленные набором технологий, оптимизированных для Kubernetes. Основной вывод заключается в том, что к приложениям «для Kubernetes» относятся специализированные приложения, которые одновременно яв-

ляются облачными. Если просто облачные приложения предназначены для выполнения в облаке, то приложения для Kubernetes разрабатываются для выполнения под управлением Kubernetes.

На заре облачной разработки различия в оркестрации не позволяли приложениям быть по-настоящему облачными. Kubernetes решает проблему оркестрации, но не распространяется на сервисы облачных провайдеров (например, Roles и Permissions) и не предоставляет шину событий (например, Kafka). Идея о том, что приложения для Kubernetes являются специализированными облачными, означает, что между ними есть много общего. Основное различие заключается в независимости от облачного провайдера. Использование всех преимуществ гибридного облака и совместимость с разными облачными провайдерами требует, чтобы приложения можно было развернуть в облаке любого провайдера. Без этого вы будете привязаны к одному облачному провайдеру и вам придется полагаться на его безупречную работу. Чтобы в полной мере использовать преимущества гибридного облака, приложения должны создаваться на основе Kubernetes. Разработка для Kubernetes решает проблему переносимости. Подробнее об особенностях создания приложений для Kubernetes мы поговорим в главе 2.

Контейнеры и оркестрация для разработчиков

Один из ключевых элементов переносимости — *контейнер*. Он служит представлением доли ресурсов хост-системы вместе с приложением. Контейнеры появились еще на заре Linux по мере введения изолированных сред chroot и стали основным направлением развития контейнеров процессов Google, которые в конечном итоге превратились в контрольные группы cgroups. Их популярность резко возросла в 2013 году, в первую очередь благодаря появлению Docker, сделавшему их доступными для многих разработчиков. Важно различать Docker как компанию, контейнеры Docker, образы Docker и инструменты разработчика Docker, к которым мы все привыкли. Все началось с контейнеров Docker, однако Kubernetes позволяет запускать контейнеры в любой среде выполнения

контейнеров (например, containerd (https://containerd.io) или CRI-O (https://cri-o.io)), поддерживающей интерфейс среды выполнения контейнера (Container Runtime Interface, CRI). То, что многие называют образами Docker, на самом деле является образами, упакованными в формате Open Container Initiative (OCI) (https://opencontainers.org/).

Среда выполнения контейнеров

Контейнеры предлагают облегченную версию пространства пользователя в операционной системе Linux, урезанную до самого необходимого. Тем не менее контейнер — это все еще операционная система, и качество контейнера имеет такое же значение, как и основная операционная система. Поддержка образов контейнеров требует многих затрат на проектирование, анализа безопасности и оценки ресурсов, необходимых для поддержки образов контейнеров. Как следствие, нужно тестировать не только сами образы, но и их поведение на заданном хосте. Использование сертифицированных и совместимых с OCI базовых образов устраняет препятствия, которые могут возникать при перемещении приложений между платформами. В идеале базовые образы уже поставляются с необходимыми средами выполнения для разных языков. Для приложений на основе Java хорошей основой является универсальный базовый образ Red Hat (https://oreil.ly/KH9od). Больше о контейнерах и о том, как их используют разработчики, мы поговорим в главе 4.

Разновидности Kubernetes

До сих пор мы обсуждали Kubernetes как общую концепцию. И мы продолжим использовать слово *Kubernetes*, говоря о технологии, лежащей в основе оркестрации контейнеров. Название Kubernetes (иногда просто K8s) относится к проекту с открытым исходным кодом (https://kubernetes.io), широко известному как свод стандартов для основных функций оркестрации контейнеров. В книге мы будем использовать «простой» термин Kubernetes, рассуждая о стандартной функциональности внутри Kubernetes. Сообщество Kubernetes создало множество

разных дистрибутивов и даже разновидностей Kubernetes. Организация CNCF запустила сертифицированную программу соответствия Kubernetes (Certified Kubernetes Conformance Program (https://oreil.ly/n4XH9)), в которой на момент написания этих слов было перечислено более 138 продуктов от 108 производителей. Список включает полные дистрибутивы (например, MicroK8s, OpenShift, Rancher), предложения хостинга (например, Google Kubernetes Engine, Amazon Elastic Kubernetes Service, Azure AKS Engine) и инсталляторы (например, minikube, VanillaStack). Все они имеют общее ядро, но добавляют дополнительные функции или средства, которые, по мнению производителей, будут востребованы. В этой книге мы не делаем никаких предложений о том, какой вариант Kubernetes использовать. Вам придется самостоятельно решить, с помощью чего вы будете управлять своими рабочими нагрузками. Но, чтобы помочь вам локально опробовать примеры из книги, мы предлагаем использовать minikube (https://oreil.ly/sCQUo) и не просим выполнять полноценную установку где-то в облаке.

Управление сложностью разработки

Один из наиболее важных аспектов разработки для Kubernetes — управление средой разработки. Количество задач, которые необходимо решить для успешного развертывания в нескольких средах, увеличивается в геометрической прогрессии. Одна из причин — растущее количество отдельных частей приложений или микросервисов. Еще одна причина — конфигурация инфраструктуры для конкретного приложения. На рис. 1.1 представлен краткий обзор типичной среды с инструментами, необходимыми для полностью автоматизированной разработки. Мы поговорим о некоторых из них в книге, чтобы помочь вам начать работать в новой среде. Основные задачи разработки не изменились. Вы по-прежнему будете писать приложения или сервисы, используя подходящие фреймворки, такие как Quarkus (https://quarkus.io), как это делаем мы в книге. Данная часть рабочего процесса разработки обычно называется внутренним циклом разработки.

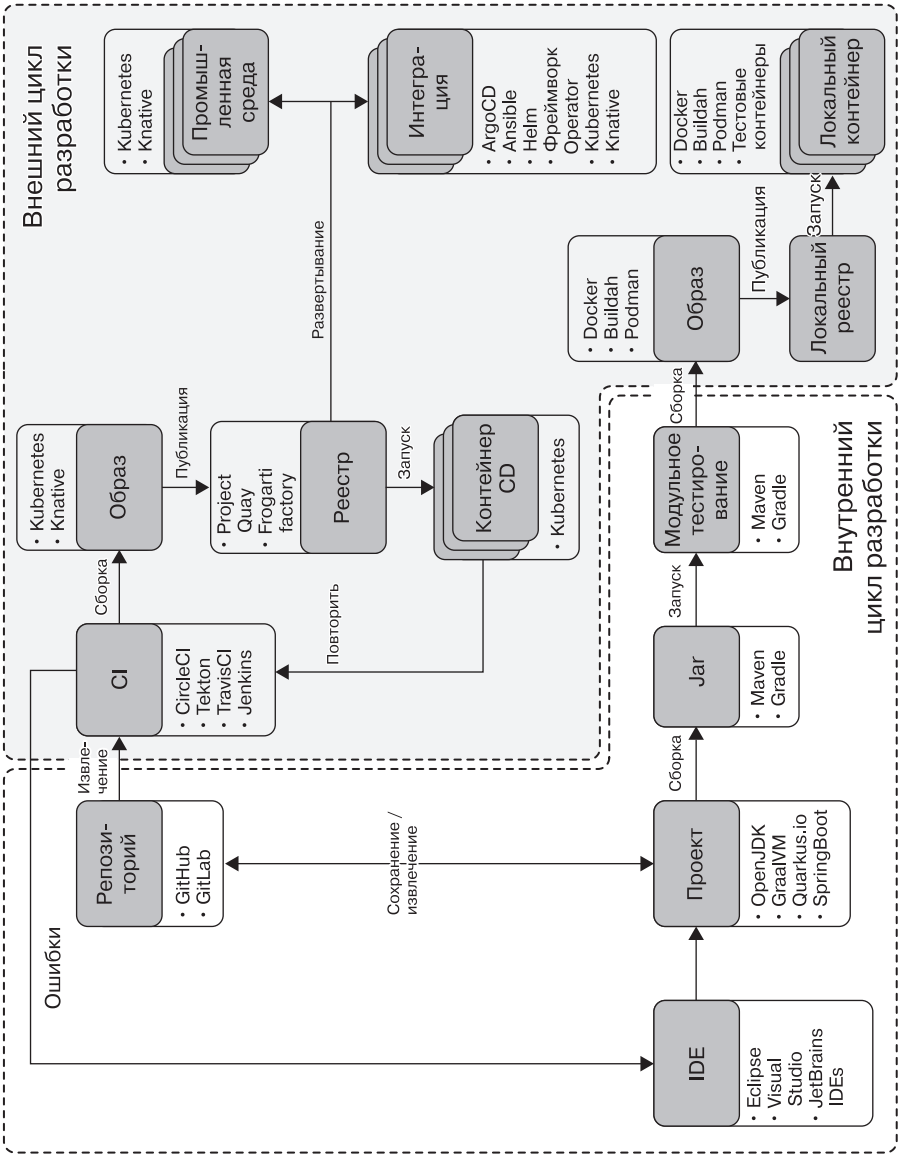


Рис. 1.1. Внутренний и внешний циклы разработки и инструменты, рекомендованные авторами

Бóльшую часть времени мы потратим на изучение изменений и возможностей во внешнем цикле. Внешний цикл принимает созданное и протестированное приложение и запускает его в работу с помощью различных механизмов. Важно понимать, что в книге мы выражаем довольно категоричные мнения. Они отражают наши знания и представления о том, как сделать разработчиков на Java максимально продуктивными, быстрыми и, возможно, даже счастливыми. Исходя из этого, мы будем рекомендовать конкретные инструменты и методы. Как показано выше на рис. 1.1, иногда у вас на выбор будет несколько вариантов. В книге мы выбрали наиболее традиционный для Java-разработчиков путь. Для сборки приложений мы используем Maven вместо Gradle, а для создания образов контейнеров — podman. Мы также используем OpenJDK, а не GraalVM и придерживаемся JUnit вместо Testcontainers (https://oreil.ly/kbudT).

Но облачная экосистема, представленная в ландшафте CNCF (https://oreil.ly/kqsG9), предлагает еще больше инструментов на выбор. Воспринимайте книгу как путеводитель для разработчика Enterprise Java.

Помимо выбора технологий, вам также придется решить, как использовать эту новую экосистему. Благодаря разнообразию доступных инструментов появляется еще одно измерение, которое позволяет выбрать уровень взаимодействия с Kubernetes. Мы различаем консервативный и гибкий подходы (рис. 1.2). Как разработчик, одержимый желанием вникнуть во все тонкости, вы можете изучить все примеры, какие вам встретятся, и использовать только возможности Kubernetes при создании файлов YAML.

Первоначально аббревиатура YAML расшифровывалась как Yet Another Markup Language (еще один язык разметки). Это название было задумано как остроумная ссылка на его роль как языка разметки. Но позднее эта аббревиатура стала расшифровываться как YAML Ain't Markup Language (YAML — это не язык разметки), обозначая предназначение языка для описания данных.

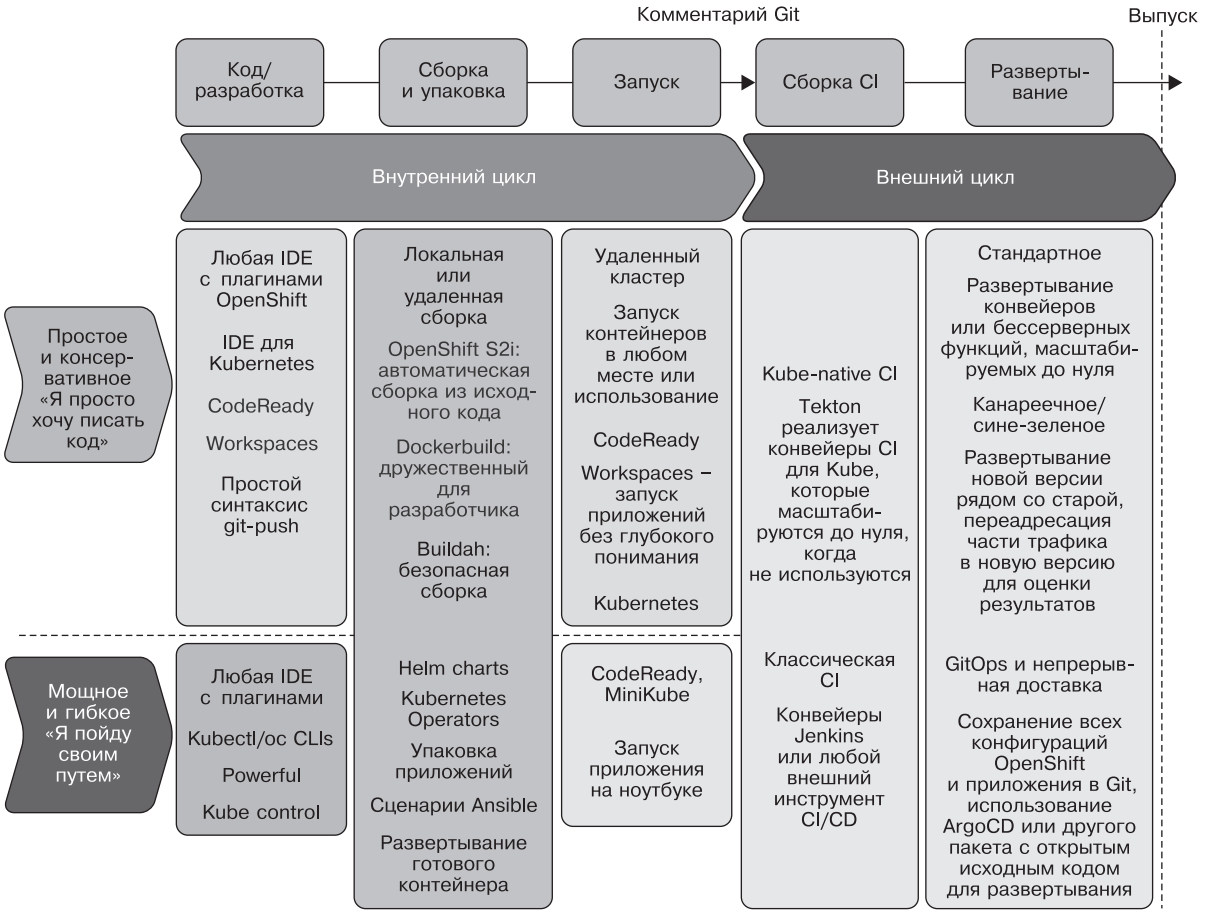


Рис. 1.2. Консерватизм или гибкость — выбор технологий во внутреннем и внешнем циклах разработки

Вы можете решить сосредоточиться исключительно на исходном коде и не отвлекаться от реализации бизнес-логики. Этого можно добиться с помощью инструментов разработчика, предоставляемых некоторыми дистрибутивами. В зависимости от того, что для вас наиболее важно в процессе разработки, на выбор есть несколько вариантов. Вы можете использовать базовый интерфейс командной строки (command-line interface, CLI) Kubernetes `kubctl` вместо, например, `oc`, специфичного для OpenShift CLI. Тем, кто хочет быть ближе к законченному продукту, мы предлагаем попробовать CodeReady Containers (https://oreil.ly/vhyZ7). Это кластер OpenShift, который можно развернуть на ноутбуке и с которым легко начать работу. Но в любом случае выбор за вами.

Еще один отличный инструмент, который мы можем порекомендовать, — это `odo` (https://oreil.ly/IyjTm), универсальный интерфейс командной строки разработчика для проектов на основе Kubernetes. Существующие инструменты, такие как `kubectl` и `oc`, больше ориентированы на операции и требуют глубокого понимания базовых концепций. Инструмент `odo` абстрагирует для разработчика сложные понятия Kubernetes. Два примера выбора из внешнего цикла разработки — это решения непрерывной интеграции (Continous Integration, CI). В книге мы используем Tekton (https://tekton.dev), и у вас будет возможность изучить его в главе 6. Кроме того, в Kubernetes можно использовать Jenkins в виде оператора Jenkins (https://oreil.ly/0Z1Cv) или даже Jenkins X. Какой бы выбор вы ни сделали, в конце концов вы освоите Kubernetes.

DevOps и гибкость

Следующее важное нововведение при модернизации корпоративного Java-приложения — создание объединенных групп, отвечающих за все: от идеи до эксплуатации. Некоторые считают, что методология DevOps сосредоточена исключительно на операционных аспектах и сочетается с самообслуживанием для разработчиков. Но мы твердо верим, что DevOps — это командная культура, ориентированная на долгосрочное взаимодействие. Термин DevOps получен из начальных букв слов

development (разработка) и operations (эксплуатация), но фактически представляет набор идей и практик гораздо более широкий, чем простая сумма двух терминов. DevOps включает заботу о безопасности, совместную работу, анализ данных и многое другое. DevOps определяет подходы к ускорению процессов, с помощью которых новое бизнес-требование проходит путь от разработки кода до развертывания в промышленной среде. Эти подходы требуют, чтобы команды разработчиков и специалистов по эксплуатации общались и относились с пониманием к своим товарищам по команде. Кроме того, необходимы масштабируемость и гибкая настройка. Разработчики, имеющие богатый опыт программирования, должны тесно сотрудничать со специалистами по эксплуатации и помогать ускорять сборку, тестирование и выпуск программного обеспечения, не нанося ущерб надежности. Все вместе это позволяет чаще вводить в эксплуатацию новые изменения и более динамично использовать инфраструктуру. Смена традиционного подхода к разработке на методологию DevOps часто описывается как *трансформация*, и ее обсуждение выходит далеко за рамки этой книги. Тем не менее это очень важный аспект, и данная трансформация прекрасно описана во многих книгах, статьях и презентациях как «обучение слонов танцам».

Резюме

В этой главе вы познакомились с основными определениями, а также наиболее важными технологиями и концепциями, которые мы будем использовать в книге. В следующей главе мы представим исходный код нашей первой модернизации приложения.

ГЛАВА 2

Путь к облачным приложениям на Java

Πάντα ῥεῖ (панта рей — «все течет») — известный афоризм философа Гераклита, описывающий изменчивое состояние нашего существования и призывающий реагировать и приспосабливаться. Этот афоризм прекрасно описывает правильный подход к эволюции, наблюдаемой в информационных технологиях в целом и в языках программирования и средах выполнения в частности, где гетерогенные, распределенные, мультиоблачные рабочие нагрузки более типичны для бизнес-задач.

Технологии Java и Jakarta EE (ранее известная как Java EE) тоже развиваются в этом направлении, стремясь найти баланс между накопленным опытом разработки корпоративных решений и потребностью в быстро меняющихся облачных сценариях, когда наши приложения могут беспрепятственно выполняться во многих облаках. В этой главе мы расскажем о компонентах, необходимых для перехода Java-приложений в облако, и познакомим с Java-реализацией интернет-магазина под названием Coolstore.

Облачный инструментарий

В настоящее время микросервисы стали общепринятой и общепризнанной практикой. Для разработчиков JavaEE это означает резкое изменение парадигмы, когда бизнес-логика находится не на единственном сервере приложений, а распределена по различным микросервисам. Они работают на отдельных серверах приложений, таких как Tomcat или Undertow, оптимизированных с точки зрения потребляемых ресурсов, чтобы сделать такое их сосуществование в облачном мире функциональным и производительным.

Сегодня монолитный подход можно преобразовать в гетерогенную и даже независимую от языка программирования модель, в которой каждый модуль управляется определенным компонентом, работающим в другом приложении. Помимо передовых практик, таких как модели на основе API, сложность заключается в сохранении этого разнообразия. Однако в настоящее время Java предоставляет набор инструментов и фреймворков, помогающих сосредоточиться на конкретных нуждах и упростить совместную работу. В этой главе вы узнаете, как создать и развернуть приложение на основе микросервисов, использующих разные фреймворки Java.

Архитектура

Приложение интернет-магазина Coolstore — это типичное веб-приложение, содержащее три компонента:

- *уровень представления* — пользовательский интерфейс для отображения списка товаров, доступных для приобретения;
- *уровень модели* — серверная часть с бизнес-логикой каталогизации и индексации всех продаваемых товаров;
- *уровень данных* — база данных, хранящая записи о сделках и товарах.

Все вместе эти компоненты образуют интернет-магазин с каталогом товаров и описью складских запасов, которые можно организовать в виде архитектуры, изображенной на рис. 2.1.

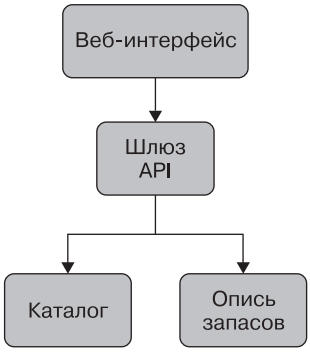


Рис. 2.1. Архитектура интернет-магазина Coolstore

Три компонента, упомянутых выше, отображаются в микросервисы, каждый из которых отвечает за свой уровень:

- *сервис каталога (Catalog Service)* использует REST API для доступа к содержимому каталога, хранящегося в реляционной базе данных;
- *сервис описи запасов (Inventory Service)* использует REST API для доступа к описи запасов товаров, хранящейся в реляционной базе данных;
- *сервис шлюза (Gateway Service)* принимает запросы и передает их сервису каталогов или сервису описи запасов;
- *сервис веб-интерфейса (WebUI Service)* вызывает сервис шлюза, чтобы получить необходимую информацию.

Уровни представления и модели представлены этими микросервисами, причем последний имеет интерфейс с уровнем данных, роль которого может играть какая-либо СУБД. Наша реализация интернет-магазина называется Coolstore, и ее веб-интерфейс выглядит следующим образом (рис. 2.2).

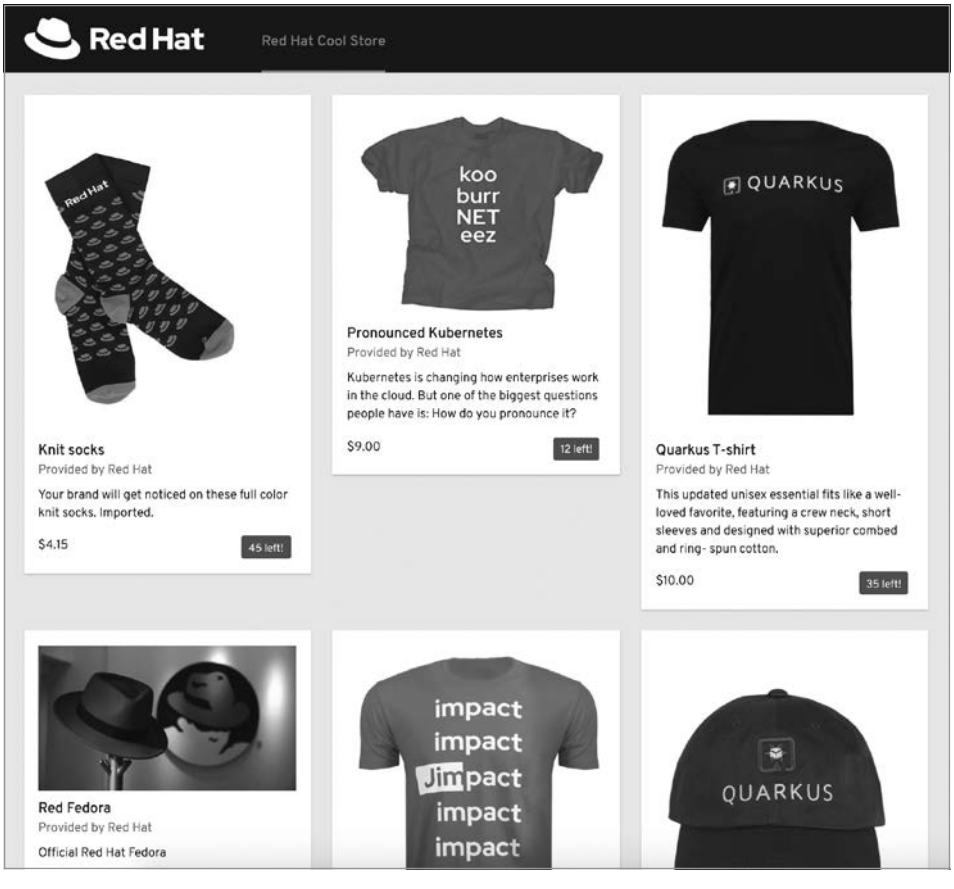


Рис. 2.2. Главная веб-страница Coolstore

Создание микросервиса описи товаров с помощью Quarkus

Quarkus (https://quarkus.io/) — это Java-фреймворк для Kubernetes, включающий все необходимое и предназначенный специально для выполнения на виртуальных машинах Java (Java virtual machines, JVM), оптимизированных для работы в контейнерах. Все это превращает данный

фреймворк в эффективную платформу для бессерверных, облачных и Kubernetes-сред.

Он предназначен для работы с популярными стандартами, платформами и библиотеками Java, такими как Eclipse MicroProfile и Spring, а также Apache Kafka, RESTEasy (JAX-RS), Hibernate ORM (JPA), Infinispan, Camel и многие другие. Кроме того, он предоставляет верную информацию в GraalVM (универсальная виртуальная машина для запуска приложений, написанных на нескольких языках, включая Java и JavaScript), чтобы приложения можно было скомпилировать в низкоуровневый код.

Quarkus позволяет реализовывать микросервисные архитектуры и предоставляет набор инструментов, помогающих разработчикам отлаживать и тестировать код. В нашем проекте интернет-магазина мы используем Quarkus для реализации микросервиса Inventory (рис. 2.3).

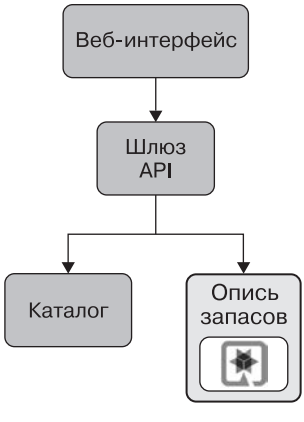


Рис. 2.3. Микросервис Inventory, реализованный на основе Quarkus

Весь исходный код этого примера можно найти в репозитории книги на GitHub (https://oreil.ly/zqbWB).

Создание проекта Quarkus с помощью Maven

Создать новый проект, использующий Quarkus, можно с помощью Maven или Gradle.

Maven и Gradle — два популярных средства настройки Java-проектов и управления зависимостями. Они различаются стратегиями управления зависимостями и имеют разные форматы представления конфигурации (XML и Kotlin DSL), но с точки зрения основных возможностей их можно считать эквивалентными. В книге мы будем использовать фреймворк Maven, так как он имеет более широкую поддержку в IDE и инструментах разработки.

Настройка нового проекта Maven производится с помощью плагина `quarkus-maven-plugin` и начинается с запуска следующей команды:

```
mvn io.quarkus:quarkus-maven-plugin:2.1.4.Final:create \
  -DprojectGroupId=com.redhat.cloudnative \
  -DprojectArtifactId=inventory-quarkus \
  -DprojectVersion=1.0.0-SNAPSHOT \
  -DclassName="com.redhat.cloudnative.InventoryResource" \
  -Dextensions="quarkus-resteasy,quarkus-resteasy-jsonb,
    quarkus-hibernate-orm-panache,quarkus-jdbc-h2"
```

Вы можете также создать проект Quarkus с помощью онлайн-конфигуратора, доступного по адресу https://code.quarkus.io.

Эта команда создаст каркас проекта с классом `InventoryResource`, который мы будем использовать для реализации микросервиса Inventory.

Посмотрим на сгенерированный файл `pom.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.redhat.cloudnative</groupId> ❶
```

```
<artifactId>inventory-quarkus</artifactId>
<version>1.0.0-SNAPSHOT</version>
<properties>
    <quarkus-plugin.version>2.1.4.Final</quarkus-plugin.version>
    <quarkus.platform.artifact-id>quarkus-bom</quarkus.platform.artifact-id>
    <quarkus.platform.group-id>io.quarkus</quarkus.platform.group-id>
    <quarkus.platform.version>2.1.4.Final</quarkus.platform.version>
    <compiler-plugin.version>3.8.1</compiler-plugin.version>
    <surefire-plugin.version>3.0.0-M5</surefire-plugin.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <maven.compiler.parameters>true</maven.compiler.parameters>
</properties>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-bom</artifactId> ❷
            <version>${quarkus.platform.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies> ❸
    <dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-resteasy</artifactId>
    </dependency>
    <dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-junit5</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>io.rest-assured</groupId>
      <artifactId>rest-assured</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-resteasy-jsonb</artifactId>
    </dependency>
    <dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-hibernate-orm-panache</artifactId>
    </dependency>
```

```
        <dependency>
          <groupId>io.quarkus</groupId>
          <artifactId>quarkus-jdbc-h2</artifactId>
        </dependency>
    </dependencies>
...
            <build>
                <plugins>
                    <plugin>
                        <groupId>io.quarkus</groupId>
                        <artifactId>quarkus-maven-plugin</artifactId> ❹
                        <version>${quarkus-plugin.version}</version>
                        <executions>
                            <execution>
                                <goals>
                                    <goal>native-image</goal>
                                </goals>
                                <configuration>
                                    <enableHttpUrlHandler>true
                                    </enableHttpUrlHandler>
                                </configuration>
                            </execution>
                        </executions>
                    </plugin>
...
    </profiles>
</project>
```

❶ Здесь определяются значения параметров `groupId`, `artifactId` и `version`. Полный список доступных параметров приводится ниже в табл. 2.1.

❷ Спецификация Quarkus импортируется, чтобы мы получили возможность не указывать версию для различных зависимостей Quarkus.

❸ Здесь перечислены все зависимости проекта, выраженные в виде расширений для добавления. Мы включили:

- поддержку сервисов JSON REST, позволяющую разрабатывать сервисы REST (https://oreil.ly/hsHvV), которые могут принимать и возвращать данные в формате JSON;
- Hibernate ORM Panache (https://oreil.ly/zqJDh): реализацию JPA, которая предлагает весь спектр возможностей Object Relational Mapper и дает возможность упростить слой хранения данных на основе Hibernate, помогая создавать и обслуживать ваши сущности;

- библиотеку поддержки баз данных Datasources (H2) (https://oreil.ly/Q5nGV): она дает возможность подключиться к базе данных; в этом примере мы будем применять H2, базу данных в памяти, готовую к использованию в приложениях Java.

❹ Плагин `quarkus-maven-plugin`, отвечающий за упаковку приложения, а также за поддержку режима разработки.

Таблица 2.1. Параметры проекта Quarkus Maven

Атрибут	Значение по умолчанию	Описание
`projectGroupId`	`com.redhat.cloudnative`	Идентификатор группы проекта
`projectArtifactId`	*Обязательный*	Идентификатор артефакта созданного проекта. В его отсутствие активируется интерактивный режим
`projectVersion`	`1.0-SNAPSHOT`	Версия созданного проекта
`platformGroupId`	`io.quarkus`	Идентификатор группы целевой платформы. Учитывая, что все существующие платформы берут начало в io.quarkus, это значение не используется явно, но все же дает такую возможность
`platformArtifactId`	`quarkus-universe-bom`	Идентификатор артефакта спецификации BOM целевой платформы. Чтобы использовать Quarkus локально, этот параметр должен иметь значение `quarkus-bom`
`platformVersion`	Если значение не указано явно, то по умолчанию интерпретируется как последняя версия	Версия платформы для использования в проекте. Можно указать диапазон версий, и в этом случае будет использоваться самая последняя версия из указанного диапазона
`className`	*Не создается, если опущено*	Полное имя созданного ресурса
`path`	`/hello`	Путь к ресурсу; имеет смысл, только если установлен параметр `className`
`extensions`	`[]`	Список расширений (через запятую) для добавления в проект

Получить список всех доступных расширений можно, запустив в каталоге проекта команду:

```
./mvnw quarkus:list-extensions
```

Реализация предметной модели

Пришло время написать код и создать *предметную модель* и конечную точку RESTful сервиса Inventory. Предметная модель — популярный паттерн разработки программного обеспечения, и он прекрасно подходит для использования в облаке. Уровень абстракции, заданный паттерном, делает его пригодным для использования в качестве объектно-ориентированного способа моделирования бизнес-логики микросервисов.

Найти определение предметной модели можно в классе `Inventory` в репозитории этой книги на GitHub (https://oreil.ly/JE6CD).

Наша реализация предметной модели состоит из экземпляра сущности `Entity`, представляющего перечень товаров на уровне хранения данных:

```
package com.redhat.cloudnative;

import javax.persistence.Entity;
import javax.persistence.Table;

import io.quarkus.hibernate.orm.panache.PanacheEntity;

import javax.persistence.Column;

@Entity ❶
@Table(name = "INVENTORY") ❷
public class Inventory extends PanacheEntity{ ❸

    @Column
    public int quantity; ❸

    @Override
    public String toString() {
        return "Inventory [Id='" + id + '\'' + ", quantity=" + quantity + ']';
    }
}
```

❶ `@Entity` отмечает класс как сущность JPA.

❷ `@Table` настраивает процесс создания таблицы, определяя имя таблицы и ограничения базы данных, в данном случае таблица получает имя `INVENTORY`.

❸ Quarkus автоматически сгенерирует методы чтения/записи для общедоступных атрибутов при наследовании класса `PanacheEntity`. Кроме того, автоматически будет добавлен атрибут `id`.

Когда модель будет определена, можно обновить свойства в файле `application.properties`, чтобы получить инструкции по заполнению данных для нашего микросервиса:

```
quarkus.datasource.jdbc.url=jdbc:h2:mem:inventory;
DB_CLOSE_ON_EXIT=FALSE;DB_CLOSE_DELAY=-1 ❶
quarkus.datasource.db-kind=h2
quarkus.hibernate-orm.database.generation=drop-and-create
quarkus.hibernate-orm.log.sql=true
quarkus.hibernate-orm.sql-load-script=import.sql ❷
%prod.quarkus.package.uber-jar=true ❸
```

❶ Путь JDBC для базы данных в памяти; его можно изменить, чтобы организовать подключение к БД другого типа, такой как СУБД.

❷ SQL-сценарий для заполнения Coolstore некоторыми данными:

```
INSERT INTO INVENTORY(id, quantity) VALUES (100000, 0);
INSERT INTO INVENTORY(id, quantity) VALUES (329299, 35);
INSERT INTO INVENTORY(id, quantity) VALUES (329199, 12);
INSERT INTO INVENTORY(id, quantity) VALUES (165613, 45);
INSERT INTO INVENTORY(id, quantity) VALUES (165614, 87);
INSERT INTO INVENTORY(id, quantity) VALUES (165954, 43);
INSERT INTO INVENTORY(id, quantity) VALUES (444434, 32);
INSERT INTO INVENTORY(id, quantity) VALUES (444435, 53);
```

❸ `uber-jar` содержит все зависимости, необходимые для запуска приложения с помощью `java -jar`. По умолчанию в Quarkus генерация `uber-jar` выключена. Параметр с префиксом `%prod` активируется только при создании jar-файла, предназначенного для развертывания.

Создание RESTful-сервиса

Создавая REST-сервисы, Quarkus следует стандарту JAX-RS. При формировании нового проекта, как было показано выше, пример сервиса `hello` был создан в пути `className`, который мы определили. Теперь нам нужно экспортировать сервис REST, с помощью которого можно будет получить количество доступных единиц товара из описи, используя:

- путь `/api/inventory/{itemId}`;
- HTTP-метод `GET`.

Сервис должен вернуть количество товара с указанным идентификатором, имеющееся на складе.

Изменим определение класса `InventoryResource` следующим образом:

```
package com.redhat.cloudnative;

import javax.enterprise.context.ApplicationScoped;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/api/inventory")
@ApplicationScoped
public class InventoryResource {
    @GET
    @Path("/{itemId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Inventory getAvailability(@PathParam("itemId") long itemId) {
        Inventory inventory = Inventory.findById(itemId); ❶
        return inventory;
    }
}
```

❶ Унаследовав `PanacheEntity`, мы можем использовать паттерн сохранения активной записи вместо объекта доступа к данным (Data Access

Obect, DAO). Это означает, что наша сущность автоматически получает все методы, выполняющие операции с хранилищем.

Мы только что реализовали параметризуемую конечную точку REST нашего микросервиса, обслуживающую в JSON-представлении товары, которые есть в нашем магазине Coolstore. Таким образом, мы создали слой для обработки HTTP-запросов `GET` в нашей модели данных `Inventory`, реализованной на предыдущем шаге.

Quarkus избавляет от необходимости создавать класс Application. Он поддерживается, но не требуется. Кроме того, ресурсы создаются в единственном экземпляре для приложения, а не для каждого запроса. Изменить это поведение можно с помощью аннотаций *Scoped (ApplicationScoped, RequestScoped и т. д.).

Запуск приложения в режиме разработки

Режим разработки в Quarkus — одна из самых интересных особенностей облачной разработки на Java на сегодняшний день. Этот режим обеспечивает горячее развертывание с фоновой компиляцией, благодаря чему при изменении файла с исходным кодом на Java или файла ресурсов и последующем обновлении страницы в браузере изменения автоматически вступают в силу. То же относится к файлам ресурсов, таким как файл с конфигурационными свойствами. Кроме того, обновление страницы в браузере запускает сканирование рабочей области, и при обнаружении любых изменений файлы Java перекомпилируются, а приложение повторно развертывается; после этого ваш запрос обрабатывается повторно развернутым приложением. Если в процессе компиляции или развертывания возникнут какие-либо проблемы, то страница с ошибкой сообщит вам об этом.

Запустить приложение в режиме разработки можно с помощью встроенной цели Maven с именем `quarkus:dev`:

```
./mvnw compile quarkus:dev
```

После запуска приложения в режиме разработки должен появиться такой вывод:

```
...
Hibernate:

    drop table if exists INVENTORY CASCADE
Hibernate:

    create table INVENTORY (
       id bigint not null,
        quantity integer,
        primary key (id)
    )

Hibernate:
    INSERT INTO INVENTORY(id, quantity) VALUES (100000, 0)
Hibernate:
    INSERT INTO INVENTORY(id, quantity) VALUES (329299, 35)
Hibernate:
    INSERT INTO INVENTORY(id, quantity) VALUES (329199, 12)
Hibernate:
    INSERT INTO INVENTORY(id, quantity) VALUES (165613, 45)
Hibernate:
    INSERT INTO INVENTORY(id, quantity) VALUES (165614, 87)
Hibernate:
    INSERT INTO INVENTORY(id, quantity) VALUES (165954, 43)
Hibernate:
    INSERT INTO INVENTORY(id, quantity) VALUES (444434, 32)
Hibernate:
    INSERT INTO INVENTORY(id, quantity) VALUES (444435, 53)
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2020-12-02 13:11:16,565 INFO  [io.quarkus] (Quarkus Main Thread)
inventory-quarkus 1.0.0-SNAPSHOT on JVM (powered by Quarkus 1.7.2.Final)
started in 1.487s. Listening on: http://0.0.0.0:8080
2020-12-02 13:11:16,575 INFO  [io.quarkus] (Quarkus Main Thread)
Profile dev activated. Live Coding activated.
2020-12-02 13:11:16,575 INFO  [io.quarkus] (Quarkus Main Thread)
Installed features: [agroal, cdi, hibernate-orm, jdbc-h2, mutiny,
narayana-jta, resteasy, resteasy-jsonb, smallrye-context-propagation]
```

Как следует из вывода, `Hibernate` создал базу данных, носящую имя предметной модели, и заполнил ее некоторыми исходными данными, определенными в файле свойств.

Формируя проект в начале этой главы, мы включили ряд зависимостей, таких как Panache, и использовали их для представления нашей модели данных как сущности Entity в базе данных.

Кроме того, можно заметить, что после успешного запуска наше приложение прослушивает порт 8080. Если сейчас в браузере ввести адрес http://localhost:8080, то он откроет страницу приветствия Quarkus (рис. 2.4).

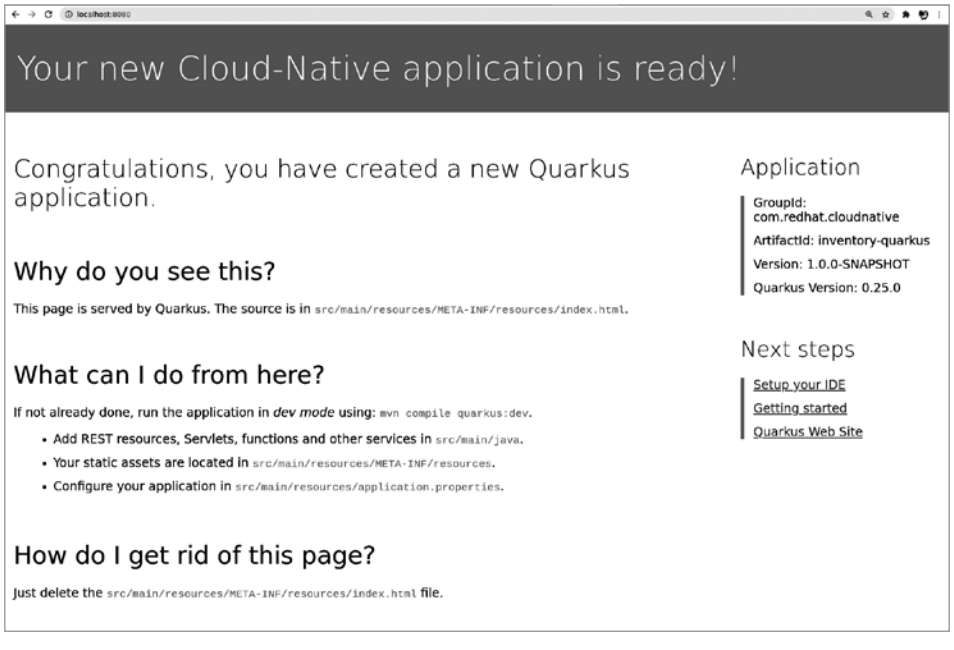


Рис. 2.4. Страница приветствия Quarkus

Остановить приложение, выполняющееся в режиме разработки, можно, нажав сочетание клавиш Ctrl+C в терминале, где было запущено приложение. Если Quarkus 2 запускается в режиме разработки, то по умолчанию включается функция непрерывного тестирования, когда после сохранения изменений в коде автоматически запускаются тесты.

Теперь можете попробовать запросить один из товаров, добавленных в файле `import.sql`, чтобы проверить, правильно ли работает наш микросервис.

Для этого просто введите в браузере адрес http://localhost:8080/api/inventory/329299.

В результате вы должны получить следующий вывод:

```
{
   "id":"329299",
   "quantity":35
}
```

REST API вернул объект JSON, представляющий количество единиц этого товара, имеющихся на складе. Поздравляем, с помощью Quarkus вы создали ваш первый облачный микросервис!

Теперь мы перейдем к разработке других микросервисов, которые будут использовать этот микросервис, поэтому оставьте его запущенным, чтобы иметь возможность продолжить разработку приложения Coolstore.

Создание микросервиса Catalog с помощью Spring Boot

Spring Boot (https://spring.io/projects/spring-boot) — проверенный временем фреймворк, позволяющий легко создавать автономные приложения на основе Spring (https://spring.io/) со встроенными веб-контейнерами, такими как Tomcat (или веб-сервер JBoss), Jetty и Undertow, которые можно запускать в JVM непосредственно с помощью команды `java -jar`. Spring Boot также позволяет создавать war-файлы для развертывания в автономных веб-контейнерах.

Консервативный подход означает, что выбор из множества вариантов, доступных на платформе Spring, и сторонних библиотек уже сделан в Spring Boot, благодаря чему вы можете начать работу, обойдясь минимумом усилий и настроек.

Библиотека Spring Boot очень популярна в облачной разработке на Java, поскольку, как отмечается на официальном сайте, позволяет легко создавать автономные приложения промышленного качества на основе Spring (https://oreil.ly/KYWe5), которые вы можете «просто запускать». Мы включим Spring Boot в нашу архитектуру микросервиса Catalog (рис. 2.5).

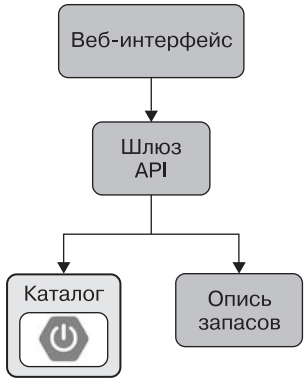


Рис. 2.5. Микросервис Catalog на основе Spring Boot

Весь исходный код этого примера можно найти в репозитории книги на GitHub (https://oreil.ly/M8ya6).

Создание проекта Spring Boot с помощью Maven

В этом случае тоже есть возможность создать проект Spring Boot с помощью Maven или Gradle. Но самый простой вариант — использовать Spring Initializr (https://start.spring.io/), онлайн-конфигуратор, помогающий создать структуру проекта со всеми необходимыми зависимостями.

В данном случае мы используем версию Spring Boot, поддерживаемую Red Hat, из репозиториев Red Hat Maven (https://oreil.ly/mAJRs), и определим метаданные проекта, перечисленные в табл. 2.2.

Таблица 2.2. Параметры проекта Spring Boot Maven

Атрибут	Значение по умолчанию	Описание
`modelVersion`	`4.0.0`	Версия модели POM (всегда 4.0.0)
`groupId`	`com.redhat.cloudnative`	Группа или организация, которой принадлежит проект. Часто выражается как перевернутое доменное имя
`artifactId`	`catalog`	Имя, которое будет присвоено артефакту библиотеки проекта (например, имя JAR- или WAR-файла)
`version`	`1.0-SNAPSHOT`	Версия проекта
`name`	`CoolStore Catalog Service`	Имя приложения
`description`	`CoolStore Catalog Service with Spring Boot`	Описание приложения

Посмотрим на содержимое файла `pom.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.redhat.cloudnative</groupId> ❶
  <artifactId>catalog</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>CoolStore Catalog Service</name>
  <description>CoolStore Catalog Service with Spring Boot</description>
  <properties>
    <spring-boot.version>2.1.6.SP3-redhat-00001</spring-boot.version> ❷
    <spring-boot.maven.plugin.version>2.1.4.RELEASE-redhat-00001
    </spring-boot.maven.plugin.version>
    <spring.k8s.bom.version>1.0.3.RELEASE</spring.k8s.bom.version>
    <fabric8.maven.plugin.version>4.3.0</fabric8.maven.plugin.version>
```

```
</properties>
<repositories>
  <repository>
    <id>redhat-ga</id>
    <url>https://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>redhat-ga-plugins</id>
    <url>https://maven.repository.redhat.com/ga/</url>
  </pluginRepository>
</pluginRepositories>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>me.snowdrop</groupId>
      <artifactId>spring-boot-bom</artifactId>
      <version>${spring-boot.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-kubernetes-dependencies</artifactId>
      <version>${spring.k8s.bom.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies> ❸
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-kubernetes-config</artifactId>
  </dependency>
```

```
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
    </dependency>
  </dependencies>
...
</project>
```

❶ Метаданные проекта, сгенерированные с помощью Initializr или вручную.

❷ Используемая версия Spring Boot.

❸ Необходимые зависимости:

- JPA: Spring Data с JPA;
- Spring Cloud (https://oreil.ly/n4oSG): поддержка и инструменты Spring для облачных приложений на Java;
- H2: база данных в памяти, которую мы будем использовать для хранения данных этого микросервиса.

Это минимальный проект Spring Boot с поддержкой сервисов RESTful и Spring Data с JPA для подключения к базе данных. Любой новый проект не содержит никакого кода, кроме основного класса, в данном случае класса `CatalogApplication`, предназначенного для запуска приложения Spring Boot.

Его можно найти в репозитории книги на GitHub (https://oreil.ly/FK15g):

```
package com.redhat.cloudnative.catalog;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication ❶
public class CatalogApplication {

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

❶ Удобная аннотация, которая добавляет автоматически сгенерированную конфигурацию и сканирование компонентов, а также позволяет определять дополнительные настройки. Эквивалентна использованию `@Configuration`, `@EnableAutoConfiguration` и `@ComponentScan` с их атрибутами по умолчанию.

Реализация предметной модели

Далее нужно предоставить некоторые данные микросервису, представляющему каталог на сайте интернет-магазина Coolstore, и определить предметную модель для взаимодействия с уровнем хранения и интерфейс, который обеспечит связь между конечной точкой REST сервиса и моделью данных (рис. 2.6).

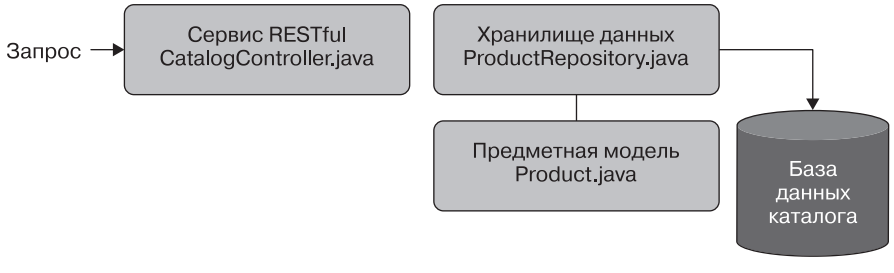


Рис. 2.6. Поток обработки данных

База данных настраивается с помощью конфигурации приложения Spring, находящейся в файле свойств `application.properties`. Посмотрим, как в этом файле настроено подключение к базе данных.

Его можно найти в репозитории книги на GitHub (https://oreil.ly/cRnE6):

```
spring.application.name=catalog
server.port=8080
spring.datasource.url=jdbc:h2:mem:catalog;DB_CLOSE_ON_EXIT=FALSE ❶
spring.datasource.username=sa
spring.datasource.password=spring.datasource.driver-class-name=org.h2.Driver ❷
```

❶ JDBC URL для базы данных H2.

❷ Используется база данных H2 в памяти.

Создадим предметную модель, следуя аналогии с предметной моделью для микросервиса Inventory, созданной выше.

Это определение можно найти в репозитории книги на GitHub (https://oreil.ly/s971w):

```
package com.redhat.cloudnative.catalog;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity ❶
@Table(name = "PRODUCT") ❷
public class Product implements Serializable {

  private static final long serialVersionUID = 1L;

  @Id ❸
  private String itemId;

  private String name;

  private String description;

  private double price;

  public Product() {
  }

  public String getItemId() {
    return itemId;
  }

  public void setItemId(String itemId) {
    this.itemId = itemId;
  }
```

```
  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }

  public double getPrice() {
    return price;
  }

  public void setPrice(double price) {
    this.price = price;
  }

  @Override
  public String toString() {
    return "Product [itemId=" + itemId + ", name=" + name
      + ", price=" + price + "]";
  }
}
```

❶ `@Entity` отмечает класс как сущность JPA.

❷ `@Table` настраивает процесс создания таблицы, определяя ее имя и ограничения базы данных. В данном случае таблица получит имя `CATALOG`.

❸ `@Id` отмечает первичный ключ в таблице.

Создание репозитория данных

Абстракция репозитория в Spring Data упрощает работу с моделями данных в приложениях Spring за счет уменьшения объема типового кода, необходимого для реализации уровней доступа к данным в различных

хранилищах. Репозиторий и его субинтерфейсы (https://oreil.ly/wUh7w) — центральные концепции в Spring Data, обеспечивающие функциональность манипулирования данными для управляемых классов сущностей. В момент запуска приложение Spring находит все интерфейсы, отмеченные как репозитории, и для каждого настраивает необходимые технологии хранения и предоставляет реализацию интерфейса репозитория.

Теперь создадим новый Java-интерфейс с именем `ProductRepository` в пакете `com.redhat.cloudnative.catalog` и унаследуем интерфейс `CrudRepository` (https://oreil.ly/gPUjj), чтобы сообщить фреймворку Spring, что нам нужен полный набор методов для управления сущностью.

Этот код можно найти в репозитории книги на GitHub (https://oreil.ly/CIGc5):

```
package com.redhat.cloudnative.catalog;

import org.springframework.data.repository.CrudRepository;

public interface ProductRepository extends CrudRepository<Product, String> { ❶
}
```

❶ `CrudRepository` (https://oreil.ly/eRvCG): интерфейс, сообщающий фреймворку Spring, что нам нужен полный набор методов для управления сущностью.

Теперь, когда у нас есть предметная модель и репозиторий для извлечения предметной модели, создадим сервис RESTful, возвращающий список товаров.

Создание RESTful-сервиса

Spring Boot использует Spring Web MVC в качестве стека RESTful по умолчанию. Теперь мы создадим для этого новый класс Java с именем `CatalogController` в пакете `com.redhat.cloudnative.catalog`, представляющий конечную точку REST. Мы будем использовать:

- путь `/api/catalog/`;
- HTTP-метод `GET`.

В ответ сервис будет возвращать каталог всех товаров, доступных в магазине, сопоставляя товары из сервиса Inventory с данными из сервиса Catalog.

Соответствующий код можно найти в репозитории книги на GitHub (https://oreil.ly/SjQ4h):

```
package com.redhat.cloudnative.catalog;

import java.util.List;
import java.util.Spliterator;
import java.util.stream.Collectors;
import java.util.stream.StreamSupport;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value = "/api/catalog") ❶
public class CatalogController {

    @Autowired ❷
    private ProductRepository repository; ❸

    @ResponseBody
    @GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
    public List<Product> getAll() {
        Spliterator<Product> products = repository.findAll().spliterator();
        return
          StreamSupport.stream(products, false).collect(Collectors.toList());
    }
}
```

❶ `@RequestMapping` указывает, что сервис REST определяет конечную точку, доступную через HTTP-метод `GET` в пути `/api/catalog`.

❷ Spring Boot автоматически добавляет реализацию `ProductRepository` во время выполнения и внедряет ее в контроллер с помощью аннотации `@Autowired` (https://oreil.ly/nuvh0).

❸ Атрибут `repository` в классе контроллера используется для получения списка товаров из баз данных.

Теперь наш второй микросервис готов к запуску. Он будет прослушивать порт 9000, чтобы избежать конфликтов с другим микросервисом:

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

В результате должен появиться такой вывод:

```
[INFO] --- spring-boot-maven-plugin:2.1.4.RELEASE-redhat-00001:run (default-cli)
  @ catalog ---
[INFO] Attaching agents: []
2020-12-02 17:12:18.528  INFO 61053 --- [           main]
trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.cloud.auto
configure.ConfigurationPropertiesRebinderAutoConfiguration' of type [org.
springframework.cloud.autoconfigure.ConfigurationPropertiesRebinder
AutoConfiguration$$EnhancerBySpringCGLIB$$e898759c] is not eligible for getting
processed by all BeanPostProcessors (for example: not eligible for auto-
proxying)
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.6.RELEASE)
StandardService   : Starting service [Tomcat]
2020-12-02 17:12:20.064  INFO 61053 --- [           main]
org.apache.catalina.core.StandardEngine  : Starting Servlet Engine:
  Apache Tomcat/9.0.7.redhat-16
2020-12-02 17:12:20.220  INFO 61053 --- [           main]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded
  WebApplicationContext
2020-12-02 17:12:20.220  INFO 61053 --- [           main]
...
```

Теперь порт 9000 прослушивает вновь настроенная конечная точка. Вы можете убедиться в этом, открыв в браузере страницу с адресом http://localhost:9000/api/catalog.

В результате вы должны увидеть следующие данные в возвращаемом объекте JSON, представляющем список товаров:


```
[
   {
      "itemId":"100000",
      "name":"Red Fedora",
      "description":"Official Red Hat Fedora",
      "price":34.99
   },
   {
      "itemId":"329299",
      "name":"Quarkus T-shirt",
      "description":"This updated unisex essential fits like a well-loved
      favorite, featuring a crew neck, short sleeves and designed
      with superior combed and ring- spun cotton.",
      "price":10.0
   },
   {
      "itemId":"329199",
      "name":"Pronounced Kubernetes",
      "description":"Kubernetes is changing how enterprises work
        in the cloud. But one of the biggest questions people have is:
        How do you pronounce it?",
      "price":9.0
   },
   {
      "itemId":"165613",
      "name":"Knit socks",
      "description":"Your brand will get noticed on these full color knit
      socks. Imported.",
      "price":4.15
   },
   {
      "itemId":"165614",
      "name":"Quarkus H2Go water bottle",
      "description":"Sporty 16. 9 oz double wall stainless steel thermal bottle
      with copper vacuum insulation, and threaded insulated lid. Imprinted.
      Imported.",
      "price":14.45
   },
   {
      "itemId":"165954",
      "name":"Patagonia Refugio pack 28L",
      "description":"Made from 630-denier 100% nylon (50% recycled / 50%
       high-tenacity) plain weave; lined with 200-denier 100% recycled
       polyester.
```

```
      ...",
      "price":6.0
   },
   {
      "itemId":"444434",
      "name":"Red Hat Impact T-shirt",
      "description":"This 4. 3 ounce, 60% combed ringspun cotton/
       40% polyester jersey t- shirt features a slightly heathered
       appearance. The fabric laundered for reduced shrinkage.
       Next Level brand apparel. Printed.",
      "price":9.0
   },
   {
      "itemId":"444435",
      "name":"Quarkus twill cap",
      "description":"100% cotton chino twill cap with an unstructured,
      low-profile, six-panel design. The crown measures 3 1/8 and this
      features a Permacurv visor and a buckle closure with a grommet.",
      "price":13.0
   },
   {
      "itemId":"444437",
      "name":"Nanobloc Universal Webcam Cover",
      "description":"NanoBloc Webcam Cover fits phone, laptop, desktop, PC,
       MacBook Pro, iMac, ...",
      "price":2.75
   }
]
```

Вывод был отформатирован, чтобы сделать его более простым для восприятия. В представленном листинге можно видеть наши товары из микросервиса Inventory, созданного на основе Quarkus, с описанием и ценой, полученными из микросервиса Catalog, созданного на основе Spring Boot. Как можно было видеть в коде выше, товар с кодом 329299 — это футболка с надписью Quarkus.

Поздравляем с созданием второго микросервиса. Теперь пришло время связать пользовательский интерфейс с внутренней реализацией. Для этого мы используем программный шлюз API с реактивной Java в следующем разделе.

Создание сервиса шлюза с помощью Vert.x

Eclipse Vert.x (https://vertx.io/) — это набор инструментов, управляемых событиями, предназначенный для создания реактивных приложений на виртуальной машине Java (JVM). Vert.x не навязывает конкретную структуру или модель упаковки; его можно использовать в существующих приложениях и платформах, чтобы добавить реактивную функциональность, просто указав JAR-файлы Vert.x в пути поиска классов приложения.

Eclipse Vert.x позволяет создавать реактивные системы, как определено в манифесте реактивных систем (https://www.reactivemanifesto.org/ru), и создает сервисы:

- отзывчивые — обрабатывающие запросы в разумные сроки;
- устойчивые — способные реагировать на сбои;
- эластичные — способные оставаться чувствительными к различным нагрузкам и имеющие возможность увеличивать и уменьшать масштаб;
- управляемые сообщениями — способные взаимодействовать с помощью асинхронной передачи.

Eclipse Vert.x предназначен для управления событиями и выполнения в неблокирующем режиме. Фактически сообщения доставляются в цикл обработки событий, который ни в коем случае нельзя блокировать. В отличие от традиционных приложений Vert.x использует очень небольшое количество потоков выполнения, ответственных за отправку событий обработчикам. Если цикл событий заблокируется, то события перестанут доставляться, поэтому код должен учитывать эту модель выполнения (рис. 2.7).

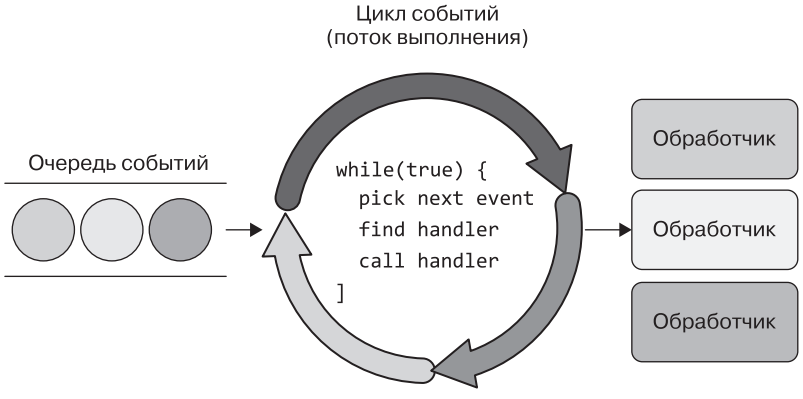


Рис. 2.7. Цикл событий Vert.x

В нашей архитектуре этот микросервис будет действовать как асинхронный программный API-шлюз, который реализован в виде реактивного микросервиса Java, эффективно маршрутизирующего и распределяющего трафик между компонентами Inventory и Catalog нашего облачного сайта интернет-магазина (рис. 2.8).

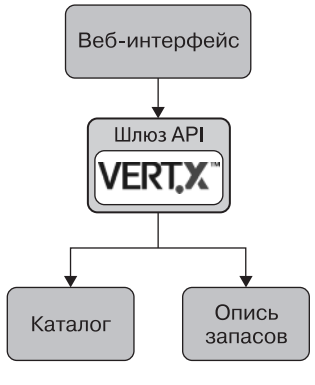


Рис. 2.8. Микросервис шлюза API на основе Vert.x

Исходный код этого микросервиса доступен в репозитории книги на GitHub (https://oreil.ly/6pe8n).

Создание проекта Vert.x Maven

Vert.x поддерживает обе системы сборки, Maven и Gradle, и самый простой способ создать новый проект Vert.x Maven — использовать структуру проекта, предлагаемую сообществом Vert.x (https://oreil.ly/fuaVI). Мы используем репозитории Red Hat Maven, поэтому добавили настройки, показанные в табл. 2.3.

Таблица 2.3. Параметры проекта Vert.x Maven

Ключ	Значение	Описание
`modelVersion`	`4.0.0`	Версия модели POM (всегда 4.0.0)
`groupId`	`com.redhat.cloudnative`	Группа или организация, которой принадлежит проект. Часто выражается как перевернутое доменное имя
`artifactId`	`gateway`	Имя, которое будет присвоено артефакту библиотеки проекта (имя JAR-файла в данном случае)
`version`	`1.0-SNAPSHOT`	Версия проекта
`name`	`CoolStore Gateway Service`	Имя приложения

Посмотрим, как будет выглядеть содержимое файла `pom.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion> ❶
    <groupId>com.redhat.cloudnative</groupId>
    <artifactId>gateway</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>CoolStore Gateway Service</name>
    <description>CoolStore Gateway Service with Eclipse Vert.x</description>
```

```
    <properties>
        <vertx.version>3.6.3.redhat-00009</vertx.version> ❷
        <vertx-maven-plugin.version>1.0.15</vertx-maven-plugin.version>
        <vertx.verticle>com.redhat.cloudnative.gateway.GatewayVerticle
        </vertx.verticle> ❸
        <fabric8.maven.plugin.version>4.3.0</fabric8.maven.plugin.version>
        <slf4j.version>1.7.21</slf4j.version>
    </properties>
...
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>io.vertx</groupId>
                <artifactId>vertx-dependencies</artifactId>
                <version>${vertx.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies> ❹
        <dependency>
            <groupId>io.vertx</groupId>
            <artifactId>vertx-core</artifactId>
        </dependency>
        <dependency>
            <groupId>io.vertx</groupId>
            <artifactId>vertx-config</artifactId>
        </dependency>
        <dependency>
            <groupId>io.vertx</groupId>
            <artifactId>vertx-web</artifactId>
        </dependency>
        <dependency>
            <groupId>io.vertx</groupId>
            <artifactId>vertx-web-client</artifactId>
        </dependency>
        <dependency>
            <groupId>io.vertx</groupId>
            <artifactId>vertx-rx-java2</artifactId>
        </dependency>
        <dependency>
            <groupId>io.vertx</groupId>
            <artifactId>vertx-health-check</artifactId>
```

```
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>${slf4j.version}</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-jdk14</artifactId>
            <version>${slf4j.version}</version>
        </dependency>
    </dependencies>
...
</project>
```

❶ Метаданные проекта.

❷ Используемая версия Vert.x

❸ `GatewayVerticle`: имя главной вертикулы; это точка входа в наше приложение.

❹ Список зависимостей:

- библиотеки Vert.x: `vertx-core`, `vertx-config`, `vertx-web`, `vertx-web-client`;
- поддержка Rx для Vert.x (https://oreil.ly/ynXXu): `vertx-rx-java2`.

Подготовка шлюза API

Далее нам нужно создать шлюз API, который будет играть роль точки входа в веб-интерфейс нашего сайта и предоставлять доступ ко всем серверным сервисам из одного места. Этот паттерн предсказуемо называется шлюзом API (https://oreil.ly/6oZaE) и считается обычной практикой в архитектуре микросервисов.

Единица развертывания в Vert.x называется *вертикулой* (verticle). Вертикула обрабатывает входящие сообщения в цикле событий, где сообщения могут быть любыми событиями, например событиями получения

сетевых буферов, синхронизации времени или сообщениями, отправленными другими вертикулами.

Основной вертикулой мы объявили `GatewayVerticle`, как указано в файле `pom.xml`, и определили конечную точку REST, которая будет пересылать запросы сервису Catalog `/api/product`:

- путь `/api/product/`;
- HTTP-метод `GET`.

Она передаст запрос сервису Catalog и вернет объект JSON, содержащий все элементы, доступные в магазине, сопоставляя элементы из сервиса Inventory с данными из сервиса Catalog.

Исходный код можно найти в репозитории книги на GitHub (https://oreil.ly/vkevU):

```
package com.redhat.cloudnative.gateway;

import io.vertx.core.http.HttpMethod;
import io.vertx.core.json.JsonArray;

import io.vertx.core.json.JsonObject;
import io.vertx.ext.web.client.WebClientOptions;
import io.vertx.reactivex.config.ConfigRetriever;
import io.vertx.reactivex.core.AbstractVerticle;
import io.vertx.reactivex.ext.web.Router;
import io.vertx.reactivex.ext.web.RoutingContext;
import io.vertx.reactivex.ext.web.client.WebClient;
import io.vertx.reactivex.ext.web.client.predicate.ResponsePredicate;
import io.vertx.reactivex.ext.web.codec.BodyCodec;
import io.vertx.reactivex.ext.web.handler.CorsHandler;
import io.vertx.reactivex.ext.web.handler.StaticHandler;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import io.reactivex.Observable;
import io.reactivex.Single;

import java.util.ArrayList;
import java.util.List;
```

```
public class GatewayVerticle extends AbstractVerticle { ❶
    private static final Logger LOG = LoggerFactory.getLogger(
        GatewayVerticle.class);

    private WebClient catalog;
    private WebClient inventory;

    @Override
    public void start() { ❷
        Router router = Router.router(vertx); ❸
        router.route().handler(CorsHandler.create("*")
        .allowedMethod(HttpMethod.GET));
        router.get("/*").handler(StaticHandler.create("assets"));
        router.get("/health").handler(this::health);
        router.get("/api/products").handler(this::products); ❹

        ConfigRetriever retriever = ConfigRetriever.create(vertx);
        retriever.getConfig(ar -> {
            if (ar.failed()) {
                // Ошибка получения конфигурации
            } else {
                JsonObject config = ar.result();

                String catalogApiHost =
                  config.getString("COMPONENT_CATALOG_HOST", "localhost");
                Integer catalogApiPort =
                  config.getInteger("COMPONENT_CATALOG_PORT", 9000);

                catalog = WebClient.create(vertx,
                    new WebClientOptions()
                        .setDefaultHost(catalogApiHost)
                        .setDefaultPort(catalogApiPort)); ❺

                LOG.info("Catalog Service Endpoint: " + catalogApiHost
                  + ":" + catalogApiPort.toString());

                String inventoryApiHost =
                  config.getString("COMPONENT_INVENTORY_HOST", "localhost");
                Integer inventoryApiPort =
                  config.getInteger("COMPONENT_INVENTORY_PORT", 8080;

                inventory = WebClient.create(vertx,
                    new WebClientOptions()
                        .setDefaultHost(inventoryApiHost)
                        .setDefaultPort(inventoryApiPort)); ❻
```

```
                LOG.info("Inventory Service Endpoint: "
                  + inventoryApiHost + ":" + inventoryApiPort.toString());

                vertx.createHttpServer()
                    .requestHandler(router)
                    .listen(Integer.getInteger("http.port", 8090)); ❼

                LOG.info("Server is running on port "
                  + Integer.getInteger("http.port", 8090));
            }
        });
    }
    private void products(RoutingContext rc) {
 ...
    }

    private Single<JsonObject> getAvailabilityFromInventory(JsonObject product) {
...
    }

    private void health(RoutingContext rc) {
...
    }
}
```

❶ Вертикула создается путем наследования класса `AbstractVerticle`.

❷ Метод `start()` создает HTTP-сервер.

❸ Для отображения конечных точек REST извлекается маршрутизатор `Router`.

❹ Создается конечная точка REST для отображения конечной точки каталога `/api/catalog` с помощью функции `product()`, которая будет извлекать содержимое.

❺ Создается HTTP-сервер, прослушивающий порт 8090.

❻ Микросервису `Inventory` передается имя хоста и порт для подключения.

❼ Микросервис поддерживает возможность замены имени хоста и порта из свойств значениями переменных среды; это важно для переносимости архитектуры между облаками.

Мы используем порт 8090, чтобы избежать конфликтов при локальной разработке. Номер порта тоже можно изменить с помощью файла свойств, как описано в документе Vert.x Config (https://oreil.ly/OgGIP). При разработке микросервисов настоятельно рекомендуем использовать переменные среды для настройки хостов и портов; мы используем их для динамического отображения конечных точек сервисов Inventory и Catalog.

Теперь можно запустить наш шлюз API:

```
mvn compile vertx:run
```

Эта команда должна вернуть такой вывод:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< com.redhat.cloudnative:gateway >-------------------
[INFO] Building CoolStore Gateway Service 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- vertx-maven-plugin:1.0.15:initialize (vmp) @ gateway ---
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ gateway
---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered
 resources, i.e. build is platform dependent!
[INFO] Copying 3 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.6.1:compile (default-compile) @ gateway ---
[INFO] Changes detected - recompiling the module!
[WARNING] File encoding has not been set, using platform encoding UTF-8,
 i.e. build is platform dependent!
[INFO] Compiling 1 source file to /home/bluesman/git/cloud-native-java2/
 labs/gateway-vertx/target/classes
...
 com.redhat.cloudnative.gateway.GatewayVerticle lambda$start$0
[INFO] INFO: Catalog Service Endpoint: localhost:9000
[INFO] dic 02, 2020 6:56:56 PM com.redhat.cloudnative.gateway.GatewayVerticle
lambda$start$0
[INFO] INFO: Inventory Service Endpoint: localhost:8080
[INFO] dic 02, 2020 6:56:56 PM com.redhat.cloudnative.gateway.GatewayVerticle
lambda$start$0
[INFO] INFO: Server is running on port 8090
[INFO] dic 02, 2020 6:56:56 PM
```

Давайте убедимся, что сервис работает и правильно маршрутизирует трафик. Для этого откроем в браузере URL http://localhost:8090/api/products.

В ответ должен появиться объект JSON, возвращаемый конечной точкой сервиса Catalog:

```
[ {
  "itemId" : "165613",
  "name" : "Knit socks",
  "description" : "Your brand will get noticed on these full color
   knit socks. Imported.",
  "price" : 4.15,
  "availability" : {
    "quantity" : 45
  }
}, {
  "itemId" : "165614",
  "name" : "Quarkus H2Go water bottle",
  "description" : "Sporty 16. 9 oz double wall stainless steel thermal
   bottle with copper vacuum insulation, and threaded insulated lid.
   Imprinted. Imported.",
  "price" : 14.45,
  "availability" : {
    "quantity" : 87
  }
}, {
  "itemId" : "329199",
  "name" : "Pronounced Kubernetes",
  "description" : "Kubernetes is changing how enterprises work in the cloud.
   But one of the biggest questions people have is: How do you pronounce it?",
  "price" : 9.0,
  "availability" : {
    "quantity" : 12
  }
}, {
  "itemId" : "100000",
  "name" : "Red Fedora",
  "description" : "Official Red Hat Fedora",
  "price" : 34.99,
  "availability" : {
    "quantity" : 0
  }
}, {
  "itemId" : "329299",
  "name" : "Quarkus T-shirt",
```

```
  "description" : "This updated unisex essential fits like a well-loved
   favorite, featuring a crew neck, short sleeves and designed with superior
   combed and ring-spun cotton.",
  "price" : 10.0,
  "availability" : {
    "quantity" : 35
  }
}, {
  "itemId" : "165954",
  "name" : "Patagonia Refugio pack 28L",
  "description" : "Made from 630-denier 100% nylon (50% recycled/
   50% high-tenacity) plain weave; lined with 200-denier 100% recycled
   polyester...",
  "price" : 6.0,
  "availability" : {
    "quantity" : 43
  }
}, {
  "itemId" : "444434",
  "name" : "Red Hat Impact T-shirt",
  "description" : "This 4. 3 ounce, 60% combed ringspun cotton/40% polyester
   jersey t- shirt features a slightly heathered appearance. The fabric
   laundered for reduced shrinkage. Next Level brand apparel. Printed.",
  "price" : 9.0,
  "availability" : {
    "quantity" : 32
  }
}, {
  "itemId" : "444437",
  "name" : "Nanobloc Universal Webcam Cover",
  "description" : "NanoBloc Webcam Cover fits phone, laptop, desktop, PC,
   MacBook Pro, iMac, ...",
  "price" : 2.75
}, {
  "itemId" : "444435",
  "name" : "Quarkus twill cap",
  "description" : "100% cotton chino twill cap with an unstructured,
   low-profile, six-panel design. The crown measures 3 1/8 and this features
   a Permacurv visor and a buckle closure with a grommet.",
  "price" : 13.0,
  "availability" : {
    "quantity" : 53
  }
} ]
```

Серверная часть готова. Теперь можно добавить некоторые данные, чтобы продемонстрировать красивый пользовательский интерфейс.

Создание пользовательского интерфейса с помощью Node.js и AngularJS

Node.js (https://nodejs.org/) — популярная платформа с открытым исходным кодом для разработки асинхронного кода на JavaScript, управляемого событиями. Эта книга посвящена современным технологиям разработки на Java, однако в микросервисных архитектурах обычно используется гетерогенная среда, содержащая код и фреймворки на нескольких языках программирования. Проблема в том, как организовать эффективное взаимодействие между ними. Одно из решений заключается в создании общего интерфейса, такого как шлюз API, который пересылает сообщения через вызовы REST или системы очередей.

AngularJS — это интерфейсный веб-фреймворк на JavaScript, цель которого — упростить разработку и тестирование таких приложений. Он предоставляет платформу для клиентских архитектур «модель — представление — контроллер» (Model-View-Controller, MVC) и «модель — представление — модель представления» (Model-View-ViewModel, MVVM) (рис. 2.9). При использовании в паре с Node.js он обеспечивает быстрый и простой способ загрузки пользовательского интерфейса.

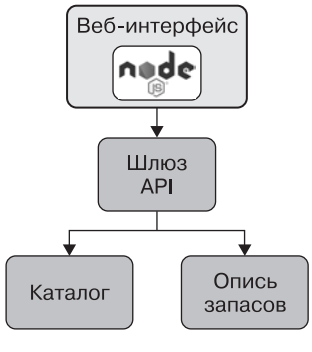


Рис. 2.9. Node.js + AngularJS Dashboard

Исходный код этого микросервиса можно найти в репозитории книги на GitHub (https://oreil.ly/fv5aa).

Запуск интерфейса

Код HTML и JavaScript готов, и мы можем связать этот интерфейс с серверными сервисами, чтобы убедиться в работоспособности приложения Coolstore.

Получение NPM

NPM (https://oreil.ly/aN4J3) — это диспетчер пакетов для JavaScript, похожий на Maven. Он поможет нам загрузить все зависимости и запустить наш интерфейс.

Установка зависимостей

Разрешить все зависимости можно, выполнив команду `npm` в каталоге web-nodejs:

```
npm install
```

Она должна вывести что-то наподобие этого:

```
...
added 1465 packages from 723 contributors and audited 1471 packages in 26.368s

52 packages are looking for funding
  run `npm fund` for details

found 228 vulnerabilities (222 low, 6 high)
  run `npm audit fix` to fix them, or `npm audit` for details
```

Запуск приложения

Теперь мы готовы проверить, сможет ли наш интерфейс правильно использовать серверные сервисы через шлюз API, сопоставляя изображения с полученными данными. Поскольку разработка выполняется локально, мы используем переменную среды, чтобы изменить порт по умолчанию Node.js во избежание конфликтов. Мы также используем переменную среды для настройки адреса конечной точки REST шлюза API (табл. 2.4).

Таблица 2.4. Переменные среды с настройками пользовательского интерфейса

Переменная	Значение	Описание
`PORT`	`3000`	Глобальная переменная среды для Node.js, определяющая номер используемого порта; в данном случае мы задействуем порт 3000
`COOLSTORE_GW_ENDPOINT`	`http://localhost:8090`	Переменная среды определяется в пользовательском интерфейсе и содержит имя хоста сервиса шлюза API

Запустим приложение с помощью команды:

```
COOLSTORE_GW_ENDPOINT=http://localhost:8090 PORT=3000 npm start
```

Введем в браузере адрес приложения Node.js http://localhost:3000.

Поздравляем! Ваш облачный сайт интернет-магазина Coolstore запущен и работает; вы можете убедиться в этом, взглянув на рис. 2.10.

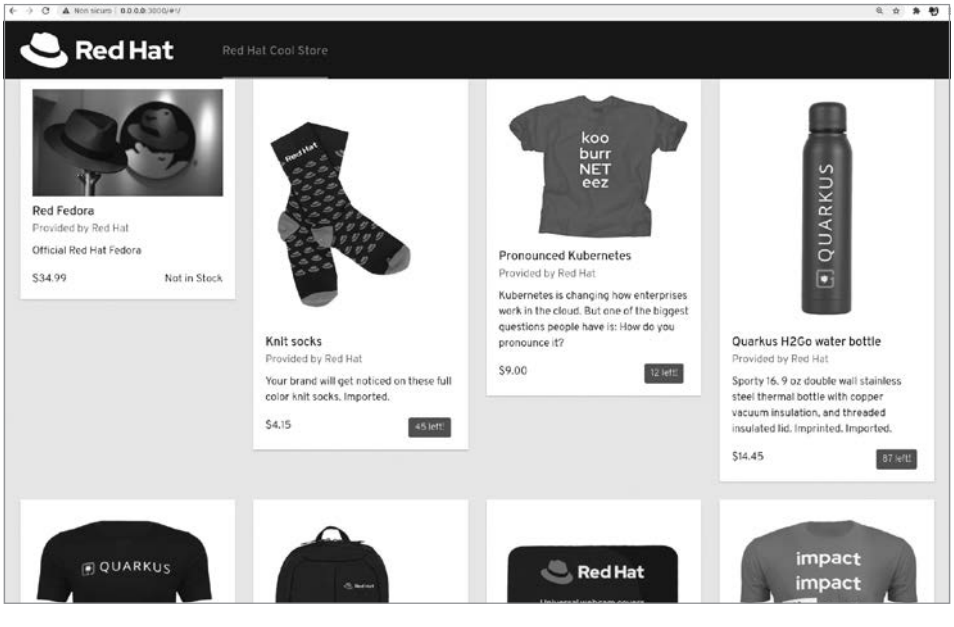


Рис. 2.10. Демонстрация работоспособности сайта магазина Coolstore

Резюме

В этой главе мы рассмотрели полную реализацию микросервисной архитектуры, использовав разные платформы Java для различных компонентов. Мы показали, как преобразовать типичное монолитное решение в более разнообразную и гетерогенную среду, легкую и готовую к работе в различных контекстах, таких как локальная разработка или промышленные системы. Это пример того, что мы называем облачной разработкой.

ГЛАВА 3

Путешествуйте налегке

> Тот, кто хочет путешествовать счастливым, должен путешествовать налегке.
>
> *Антуан де Сент-Экзюпери*

В предыдущей главе мы создали систему на основе микросервисов и показали некоторые шаги миграции существующих приложений. Но чтобы сделать примеры более понятными, в них обычно маскируют сложности. То, что может показаться очевидным в небольших примерах, становится чрезвычайно сложным в реальных бизнес-системах. Например, представьте сложную устаревшую систему. Как отмечалось в главе 1, технологии и методологии, разрабатывавшиеся на протяжении многих лет, привели к сегодняшним передовым методам и инструментам разработки современных корпоративных систем. Тот факт, что в нашей отрасли теперь есть более богатый набор инструментов, открывающих новые возможности, не означает, что их следует использовать всегда. Задумавшись об этом и о растущем количестве фреймворков, методологий и технологий, вы неизбежно зададите себе вопрос: какие инструменты и какую архитектуру использовать для разработки следующей системы

и как и где их запускать? Прежде чем вы сможете принять решение, вам нужно немного поразмышлять о наиболее известных архитектурных стилях для корпоративных приложений, которые появились за последние пару лет (трехуровневая интеграция, корпоративная интеграция, сервис-ориентированная архитектура, микросервисы и архитектура, управляемая событиями).

Трехуровневая или распределенная система

В мире Enterprise Java доминируют монолитные приложения. Они часто разрабатываются как отдельные выполняемые модули, которые масштабируются путем запуска дополнительных экземпляров сервера и с помощью функций кластеризации. Их также часто называют трехуровневыми системами, чтобы отразить три их составные части: пользовательский интерфейс на стороне клиента, реализацию бизнес-логики на стороне сервера и уровень хранения или интеграции данных на стороне сервера. Серверные части называются *монолитом*, поскольку они упакованы в один большой выполняемый файл. Любые изменения в системе обычно предполагают сборку и развертывание новой версии.

Узнать больше о создании микросервисов можно в превосходной книге Сэма Ньюмена (Sam Newman) Building Microservices (O'Reilly; https://oreil.ly/JZqsr)[1], теперь уже во втором издании.

Архитектура на основе микросервисов представляет собой подход к разработке приложений в виде наборов небольших сервисов, каждый из которых выполняется в отдельном процессе и взаимодействует с другими сервисами и механизмами через HTTP API ресурсов или как часть управляемой событиями архитектуры (Event-Driven Architecture, EDA). Эти сервисы строятся вокруг бизнес-функций и могут независимо

[1] *Ньюмен С.* Создание микросервисов. 2-е изд. — СПб.: Питер, 2023.

развертываться с помощью автоматизированного механизма развертывания. Они управляются минимальным централизованным ядром, могут разрабатываться на разных языках программирования и использовать разные технологии хранения данных.

Нет других архитектурных стилей, имеющих столь же фундаментальные различия, которые существуют между монолитами и микросервисами, как и между нефункциональными требованиями, ведущими к выбору одного из этих стилей. Наиболее важные требования вытекают из сценариев, требующих чрезвычайной гибкости в отношении масштабирования. Как опытный разработчик и архитектор, вы наверняка знаете, как оценить функциональные и нефункциональные требования к вашему конкретному проекту. В этой главе мы поможем вам сориентироваться в подходах к миграции и выборе целевой платформы. Наше путешествие мы начнем с изучения мотивации модернизации. Давайте подробнее поговорим о том, что заставляет нас задумываться о модернизации в целом и с чего начать поиск возможностей.

Технологические обновления, модернизация и трансформация

Цель разработки корпоративного программного обеспечения — придать бизнес-ценность коду, который может выполняться в рамках нефункциональных и функциональных требований. Создание ценности зависит от нашей способности быстро выводить приложения на рынок: не только лучшего качества, но и готовые к быстрым изменениям, что позволяет предприятиям реагировать на новые вызовы или нормативные изменения на рынке. И эти вызовы многогранны. В первую очередь это решение проблемы масштабируемости при использовании облачных приложений для обработки большого количества транзакций. Новые бизнес-сценарии также потребуют от вас дальнейшего анализа данных и могут быть реализованы с помощью искусственного интеллекта (ИИ) и машинного обучения (МО). И последнее, но не менее важное: наш взаимосвязанный мир генерирует все больше данных из Интернета вещей (Internet of Things,

IoT). То, что может выглядеть как естественное развитие архитектуры, таковым не является. Фактически меняющиеся бизнес-требования стимулируют модернизацию и эволюцию архитектуры за счет изменения функциональных и нефункциональных требований.

Кроме того, вы столкнетесь с операционными проблемами, влияющими на потребность в модернизации. Например, истекающие контракты на техническое обслуживание или использование устаревших технологий могут подтолкнуть к обновлению. Постоянное развитие языка Java в сочетании со все ускоряющимися циклами выпуска новых версий тоже может повлиять на решение о модернизации. Она может происходить на любом уровне вашего проекта, начиная от среды выполнения (например, от виртуальных машин до контейнеров) и заканчивая виртуальной машиной Java (версия или производитель JVM), отдельными зависимостями, пользовательскими интерфейсами и сервисами.

Здесь важно различать три аспекта модернизации. *Обновление технологий* в рамках существующих процессов и границ — хорошо знакомая задача для программных проектов, но модернизация — это нечто иное. Часто в сочетании со словом «цифровой» термин *«модернизация»* относится к внедрению новых технологий. Он предполагает обновление систем, платформ и программного обеспечения и добавление новых функций. Это может быть простое преобразование существующего ручного процесса в цифровой с помощью нового программного и аппаратного обеспечения или более сложное, такое как поэтапный отказ от существующей инфраструктуры и миграция в облако.

Иногда в разговорах о современных системах также можно услышать термин *«трансформация»*. Цифровая трансформация подразумевает использование преимуществ современных технологий для переосмысления процессов организации, культуры и опыта клиентов. Такое переосмысление может привести к новым бизнес-моделям, потокам доходов, правилам и ценностям. Трансформация — своего рода линза в организации, фокусирующаяся на коренном изменении эффективности бизнеса. Модернизация — часть трансформации и центральный

элемент, которым должны руководствоваться разработчики программного обеспечения и архитекторы.

Независимо от причин, образовавшихся в вашем проекте, прежде чем начинать модернизацию вашего приложения, помните, что сама по себе она не предъявляет никаких конкретных требований к выбору целевых сред выполнения или технологий. Это постоянно меняющийся и растущий набор технологий-кандидатов, которые позволяют компаниям конкурировать и расти в своей отрасли. Некоторые примеры можно найти в отчетах о технологических тенденциях (например, в ThoughtWorks Technology Radar (https://oreil.ly/SWvEH)) или в Gartner Hype Cycle (https://oreil.ly/JT4jE). Но, как было показано в главе 1, двумя самыми сильными мотивами постоянных инноваций являются скорость и необходимость сокращения расходов. В обоих случаях используется современная облачная архитектура на основе микросервисов.

Концепция 6R

Определившись с мотивами модернизации приложений, желательно выяснить общие подходы к модернизации и классифицировать существующие приложения. Это поможет вам управлять множеством различных приложений, особенно в проекте модернизации платформы. Вместо того чтобы исследовать элементы отдельного приложения, взгляните на полную архитектуру времени выполнения традиционных корпоративных приложений на Java. В ней вы наверняка выявите локальное оборудование, которое обычно виртуализировано и доступно для проектов в виде отдельного набора экземпляров. Учитывая, что отдельные проекты редко рассматриваются как изолированные, без каких-либо интегрированных систем, вы обязательно окажетесь в ситуации, когда вам понадобится найти скоординированное решение для нескольких проектов.

Сначала посмотрим, что такое 6R и откуда взялась эта концепция. По сути, о каждой R можно думать как о доступной стратегии миграции приложений. Каждая стратегия четко определяет результат преобразования приложения, но может не определять фактические этапы миграции.

Эта концепция была впервые упомянута аналитиком из Gartner Ричардом Уотсоном (Richard Watson) в 2011 году (https://oreil.ly/tk08O). Пять первоначальных стратегий, а именно: Rehost (перенос), Refactor (рефакторинг), Revise (пересмотр), Rebuild (пересборка) и Replace (замена), были возрождены и адаптированы в 2016 году в популярной статье (https://oreil.ly/CAalp) блога Стивена Орбана (Stephen Orban) из AWS. Орбан сохранил некоторые стратегии Gartner и добавил новые. В результате концепция 5R превратилась в 6R. Сегодня концепция 6R используется в качестве основного руководства практически для любой облачной трансформации. До сих пор ведутся споры о том, следует ли добавлять дополнительные стратегии, и в Интернете можно даже найти упоминания о концепции 7R, но в книге мы будем придерживаться 6R (рис. 3.1).

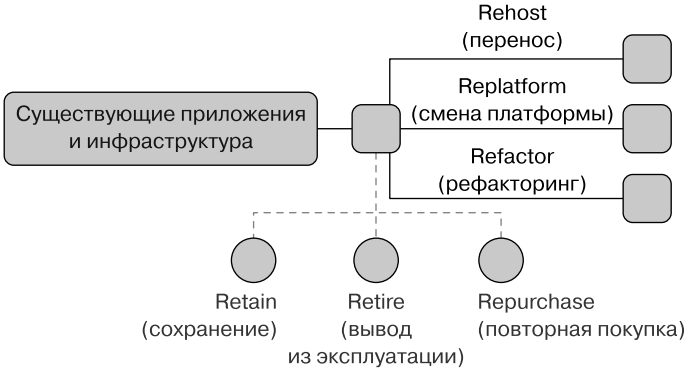


Рис. 3.1. Шесть стратегий модернизации, обзор 6R

Retain (сохранение) — модернизировать позже или не модернизировать вообще

Многие слышали историю о мейнфрейме в подвале какой-то очень известной компании, где хранятся все ее коммерческие секреты. Часто эти мейнфреймы программируются с помощью CICS (Customer Information Control System — семейство серверов приложений на разных языках, которые обеспечивают оперативное управление транзакциями и связь для приложений в системах мейнфреймов IBM), а данные хранятся в IMS (IBM Information Management System, одна из первых баз данных). И это

не обязательно плохо. Существующая система может идеально подходить для бизнеса и не требовать модернизации. Чтобы правильно оценить потребность в трансформации и модернизации, необходимо определить подобные системы и исключить их из процесса модернизации. Системы этого класса нуждаются в особом подходе к интеграции, который следует спроектировать явно. Представьте хорошо масштабируемую серверную часть мобильного приложения, которая напрямую подключается к мейнфрейму. В этом сценарии большое количество запросов от мобильных устройств может привести к перегрузке дорогостоящего мейнфрейма. Сохранение в данном случае означает не «не прикасаться», а скорее «не перемещать».

Retire (вывод из эксплуатации) — выключение системы

Возможно, найдутся такие кандидаты, которые явно достигли конца своего существования и уже были перенесены и заменены или просто устарели и не понадобятся в будущем. Путешествуйте налегке и не забудьте отметить эти системы. Последующий учет столь же важен, как и создание чего-то нового. Время, затраченное на проверку и принятие решения о выводе системы из эксплуатации, так же ценно, как и время, необходимое на перепроектирование.

Repurchase (повторная покупка) — покупка новой версии

В некоторых случаях вполне допустимо приобрести готовое программное обеспечение для новой среды выполнения. Это кажется простым, но нередко данный этап предполагает проектирование миграции и переоценку списков возможностей, поскольку маловероятно, что у вас будет возможность выполнить обновление, не изменяя версию продукта или его API. В редких случаях процесс миграции может даже застопориться из-за отсутствия документации по интеграции. Вот почему важно относиться к покупке новой версии как к проекту модернизации, а не как к простому обновлению программного обеспечения.

Rehost (перенос) — контейнеризация

Один из вариантов контейнеризации приложения, который часто называют lift and shift («поднять и перенести»), представляет собой простой перенос существующей архитектуры как есть внутрь контейнера. Это может оказаться так же просто, как звучит, но здесь есть свои проблемы. В частности, могут возникнуть трудности с оптимизацией JVM в целях выполнения в контейнерах, которым присущи свои ограничения. Некоторые существующие серверы приложений промежуточного слоя поставляются с базовыми образами, поддерживаемыми поставщиком, что упрощает переключение среды выполнения.

Особое внимание следует уделить хранилищу для сред выполнения приложений с состоянием (stateful). Серверам приложений Java требуются некоторые данные, чтобы пережить перезапуск контейнеров, а также постоянное отображение томов. Транзакции, балансировка нагрузки и репликация сеансов в памяти требуют расширенных настроек, чтобы обеспечить правильное поведение при завершении работы и обмен данными между узлами. Зарезервируйте достаточное количество времени для исследований и испытаний и старайтесь неукоснительно придерживаться рекомендаций поставщика. Этот шаг направлен на модернизацию инфраструктуры и не связан напрямую с кодом приложения. Такой подход можно применять к приложениям, которые необходимо перевести в контейнерную среду, прежде чем можно будет выполнить рефакторинг, или как промежуточный шаг на пути к смене концепции центра обработки данных.

Мартин Фаулер (Martin Fowler) ввел термин Strangler pattern (паттерн «Душитель» [https://oreil.ly/0otPb]), обозначающий способ извлечения функциональности из монолитного приложения. Он назван в честь австралийского инжира-душителя, корни которого вырастают из семян, попадающих на верхние ветви дерева-«хозяина», и продолжают расти, пока не коснутся земли.

Replatform (смена платформы) — внесение небольших корректив

Как дополнение к стратегии Rehost (перенос), стратегия Replatform (смена платформы) классифицирует приложения, которые претерпевают концептуальные или функциональные изменения при переключении среды выполнения. На эту стратегию также можно ссылаться, используя ее собственный вариант имени lift («поднять»), lift and adjust («поднять и отрегулировать»). Это может относиться к заблокированной функциональности, которая может быть реализована поверх нового стека технологий или путем изменения систем хранения данных или интеграции. Мы рекомендуем использовать данную стратегию как первый шаг к рефакторингу и разделению монолитного приложения. Предварительное выполнение этого шага облегчает выполнение операций на последующих этапах расширения и разделения. Выбирая стратегию смены платформы, вы получаете возможность плавной модернизации своих приложений и их прагматичного развития.

Refactor (рефакторинг) — создание нового

Рефакторинг (https://refactoring.com/) — это метод реструктуризации и реорганизации существующего кода, изменение его внутренней структуры без изменения внешнего поведения. Рефакторинг — наиболее трудоемкий и дорогостоящий способ переноса существующих приложений в новую среду выполнения или на новую платформу. Он может включать или не включать переход на другие архитектурные стили, локальный или облачный хостинг.

Разделяй и контейнеризируй

Теперь, рассмотрев различные стратегии модернизации существующих приложений и узнав, как и когда их применять, мы можем подумать о других предпосылках выбора целевой платформы.

Kubernetes как новый сервер приложений

Слово «платформа» в мире Enterprise Java обычно относится к серверу приложений. Серверы приложений следуют защищенному подходу к разработке программного обеспечения со стандартизированными API. Вертикальные уровни определяются тем, что обычно называют техническими уровнями в трехуровневой системе. Представление поверх бизнес-логики поверх доступа к данным и/или интеграции. По горизонтали обычно располагаются бизнес-компоненты или предметные области. Вертикальные уровни обычно хорошо разделены и слабо связаны, но горизонтальные компоненты нередко имеют общие классы и нарушения правил доступа. Если такое часто происходит в кодовой базе, то мы говорим о запутанных проектах, которые со временем превращаются в неподдерживаемые монолиты. Однако независимо от того, насколько запутан код приложения, он по-прежнему выигрывает от использования стандартных функций сервера приложений, отвечающих функциональным и нефункциональным требованиям, таким как безопасность, изоляция, отказоустойчивость, управление транзакциями, управление конфигурацией и т. д.

Если мы перенесемся к современным распределенным архитектурам, где приложения состоят из множества небольших сервисов, то обнаружим два момента: быстрого пути к хорошему дизайну компонентов больше нет и стандартные функции сервера приложений недоступны для наших компонентов.

Первое наблюдение приводит к обязательному требованию. Распределенные сервисы должны тщательно проектироваться, состоять из слабосвязанных и строго инкапсулированных компонентов. Подробнее о принципах проектирования и подходах к модернизации монолитов мы поговорим в главе 5. Второе наблюдение включает список отсутствующих функций в облачных средах. Если сервер приложений не поддерживает часто используемые функции, то, как мы упоминали, остается только два места. Одним из них может быть выбранный фреймворк

микросервисов (например, Quarkus), а другим могут быть дополнительные платформы или продукты, работающие поверх Kubernetes.

Давайте подробно рассмотрим некоторые из наиболее важных функций, обсуждаемых в следующих главах. Мы называем их *microservicilities* (микросервисные средства). Этот термин относится к списку сквозных задач, которые сервис должен реализовать, помимо бизнес-логики, предназначенной для их решения (рис. 3.2).

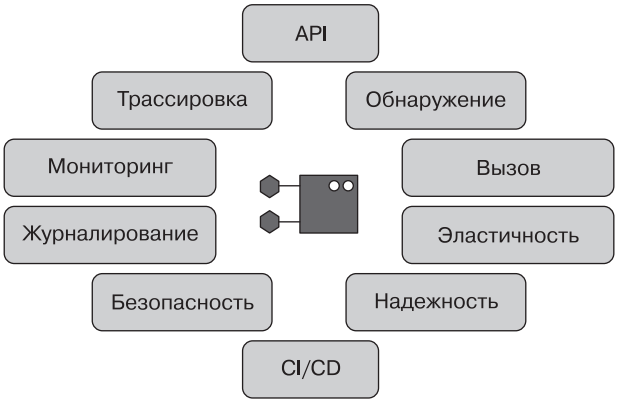


Рис. 3.2. Микросервисные средства для распределенных приложений

Обнаружение и настройка

Образы контейнеров не могут изменяться. И мы не рекомендуем хранить в них конфигурационные параметры для разных сред или этапов. Конфигурация должна быть отделена и настроена под экземпляры. Отделение конфигурации также является одним из важнейших принципов облачных приложений. Обнаружение сервисов — один из способов, позволяющих получить информацию о конфигурации из среды выполнения, вместо того чтобы жестко ее определять в приложении. Другие подходы включают использование ресурсов ConfigMaps и Secrets. Kubernetes поддерживает обнаружение сервисов по умолчанию, но этой поддержки может быть недостаточно для нужд конкретного приложения. Вы можете управлять настройками для каждой среды выполнения с помощью

файлов YAML, но дополнительные пользовательские интерфейсы или интерфейсы командной строки могут упростить разделение ответственности между командами DevOps.

Простой вызов

Доступ к приложениям, работающим внутри контейнеров, осуществляется через входные (Ingress) контроллеры. Они предоставляют маршруты для трафика HTTP и HTTPS, поступающего из-за пределов кластера, к сервисам внутри кластера. Маршрутизация трафика контролируется правилами, определенными в ресурсах Ingress. Традиционно их можно сравнить с балансировщиками HTTP-нагрузки для Apache. Другие альтернативы включают такие проекты, как HAProxy (http://www.haproxy.org/) или Nginx. Вы можете использовать возможности маршрутизации для последовательности развертываний как основу для сложной стратегии CI/CD. Для одноразовых заданий, таких как пакетные процессы, Kubernetes предоставляет поддержку заданий и заданий cron.

Эластичность

Ресурсы ReplicaSet в Kubernetes контролируют масштабирование подов и предоставляют способ достижения желаемого состояния: вы сообщаете фреймворку Kubernetes, в каком состоянии должна находиться система, чтобы он мог выяснить, как его достичь. ReplicaSet контролирует количество реплик (точных копий контейнера), которые должны работать в любое время. То, что выглядит как в значительной степени статическая операция, может быть автоматизировано. Балансировщик Horizontal Pod Autoscaler масштабирует количество подов на основе наблюдаемой нагрузки на процессор. В качестве входных данных можно использовать почти любую метрику, предоставляемую приложением.

Журналирование

Один из наиболее сложных аспектов распределенного приложения — корреляция журналов всех активных частей. В этой области разница с традиционными серверами приложений становится особенно заметной,

поскольку раньше это было настолько же просто, насколько сложно в новом мире. Мы не рекомендуем хранить журналы отдельно для каждого контейнера, так как есть риск упустить из виду общую картину и усложнить отладку побочных эффектов и поиск основных причин проблем. Существуют различные подходы к организации журналирования, в большинстве из них широко используется стек ELK (https://oreil.ly/XflXI) — Elasticsearch (https://oreil.ly/FKoKx), Logstash (https://oreil.ly/YLtNc), Kibana (https://oreil.ly/h2nIX) — или его вариант. В этих стеках Elasticsearch выступает в роли хранилища объектов, в котором хранятся все журналы. Logstash собирает журналы с узлов и передает их в Elasticsearch. А Kibana — это веб-интерфейс к Elasticsearch, который можно использовать для поиска агрегированных файлов журналов, скомпонованных из различных источников.

Мониторинг

Мониторинг в распределенном приложении — важнейший компонент, позволяющий убедиться, что все части приложения продолжают работать. В отличие от журналирования мониторинг предполагает активное наблюдение, которое часто сочетается с отправкой уведомлений, а не просто с записью событий. Стандартом де-факто для хранения сгенерированной информации является Prometheus (https://prometheus.io/). По сути, это всеобъемлющая система мониторинга с открытым исходным кодом, включающая базу данных временных рядов. Веб-интерфейс Prometheus позволяет выполнять запросы для получения метрик, оповещений и создания визуализаций, а также помогает получить представление о состоянии ваших систем.

Конвейеры сборки и развертывания

CI/CD (Continuous Integration/Continuous Delivery — непрерывная интеграция/непрерывная доставка) не является чем-то новым для приложений Enterprise Java или распределенных приложений. В соответствии с передовыми практиками разработки программного обеспечения любой промышленный код должен следовать строгому и автоматизи-

рованному циклу выпуска. При потенциально большом количестве сервисов, составляющих приложение, автоматизация должна стремиться к 100%-ному охвату. Традиционно эта задача решалась с помощью инструмента с открытым исходным кодом Jenkins (https://www.jenkins.io/), современные контейнерные платформы отошли от централизованной системы сборки и используют распределенный подход к CI/CD. Одним из примеров является Tekton (https://tekton.dev/). Цель состоит в том, чтобы создавать надежные версии программного обеспечения с помощью сборки, тестирования и развертывания. Мы углубимся в эту тему в главе 4.

Способность к восстановлению и отказоустойчивость

Психологи определяют способность к восстановлению как результат хорошей адаптации человека в условиях невзгод, травм, трагедий, угроз или больших источников стресса. В распределенных приложениях эта концепция означает возможность восстановления после сбоев или загрузки без участия человека. Kubernetes предоставляет варианты обеспечения надежности для самого кластера, но лишь частично поддерживает надежность и отказоустойчивость приложений. Например, надежность на уровне приложений можно обеспечить с помощью ресурсов PersistentVolumes, поддерживающих реплицируемые тома, или с помощью ReplicaSet, обеспечивающих постоянное количество реплик подов в кластере. На уровне приложений надежность и отказоустойчивость поддерживаются с помощью Istio или различных фреймворков, таких как Cloudstate (https://cloudstate.io/). Старайтесь использовать такие возможности, как правила повторных попыток, размыкатель цепи (circuit breaker) и сброс пула (pool ejection).

Istio (https://istio.io/) — это сервисная сетка с открытым исходным кодом, которая прозрачно накладывается на существующие распределенные приложения. Это также платформа, включающая API, которые интегрируются в любую платформу журналирования, телеметрии или систем политик.

Безопасность

Аутентификация и авторизация при взаимодействиях между сервисами не являются частью самого Kubernetes. Есть два способа реализации этих механизмов. При использовании Istio каждому сервису присваивается строгая идентичность, которая представляет роль сервиса и обеспечивает взаимодействие между кластерами и облаками. Она обеспечивает защиту взаимодействий между сервисами, а также управление ключами для автоматизации создания, распространения, ротации и отзыва ключей и сертификатов. В качестве альтернативы, более ориентированной на приложения, можно использовать компонент единого входа, такой как Keycloak (https://www.keycloak.org/) или Eclipse MicroProfile JSON Web Token — JWT (https://oreil.ly/bVETR).

Трассировка

Трассировка дает возможность отслеживать пути движения запросов и событий по всей системе в отдельных частях приложения и позволяет определить источник. В настоящее время в сообществе выработано несколько разных подходов к трассировке. Независимо от используемых языков, платформ или технологий Istio может обеспечить распределенную трассировку. Существуют также другие коммерческие проекты и проекты с открытым исходным кодом, помогающие выполнять распределенную трассировку компонентов приложения. Zipkin (https://zipkin.io/) и Jaeger (https://www.jaegertracing.io/) — два таких решения.

Определение целевой платформы

Важно отметить, что девять элементов, упомянутых выше, ориентированы на разработку приложений и охватывают не все потребности современных контейнерных платформ. Простого взгляда на этот короткий список достаточно, чтобы понять, что он оставляет без внимания важные области. Контейнерная платформа должна предоставлять функции и возможности для всей команды, от разработки до эксплуатации.

К сожалению, не существует универсального решения, подходящего для всех случаев жизни. Комплексный способ определить целевую платформу — начать с трех основных уровней: ядра, взаимодействия с клиентами и интеграции, а затем строить ландшафт приложений на основе оптимизированного стека технологий. То, что выглядит как готовый контрольный список, на самом деле таковым не является.

Компании различаются культурой, технологиями и требованиями, и следующий список является лишь рекомендуемой отправной точкой без каких-либо претензий на полноту. Мы советуем использовать его пункты для расстановки оценок и определения отдельных функциональных и нефункциональных требований с оценкой выполнения от нуля («недоступно») до трех («полностью поддерживается») и с двойкой («можно заставить это работать») в качестве средней оценки. Наконец, добавьте логику взвешивания, чтобы получить полную оценку на основе сравнения продуктов. Это может помочь сформировать основу для прямого сравнения готовых продуктов и продуктов, сделанных своими руками, а также послужить отправной точкой для создания документации платформы.

Определение ядра

Начните с оценки основной части платформы. К этой категории относятся базовые возможности, такие как оркестрация контейнеров, отображение хранилищ, частота обновления, требования к обеспечению надежности информационной системы, готовая поддержка желаемых моделей развертывания и даже дополнительная поддержка виртуальных машин. Эта категория определяет техническую основу для вашей целевой платформы, включающую:

- существующие основные возможности;
- оценку функциональных недостатков;
- поддержку гибридного облака;
- интеграцию безопасности;

- поддержку управляемых сервисов;
- доступность операторов/торговой площадки (например, OperatorHub (https://operatorhub.io/), Red Hat Marketplace (https://oreil.ly/sFuDg));
- доступные уровни поддержки;
- целевую модель развертывания;
- основной подход к модернизации.

Определение уровня взаимодействия с клиентами

В разговорах о платформах слишком мало внимания уделяется уровню взаимодействий с клиентами, который содержит техническое определение каналов связи клиентов с платформой. Канал может быть одним из порталов B2X или других специфических пользовательских интерфейсов. Тесно связанная платформа, на которой могут размещаться различные приложения, также должна включать четкое определение технического состава отдельных сервисов:

- выявление клиентоориентированных требований;
- сравнение существующего cx-фреймворка со сборкой;
- микроинтерфейсы (например, Entando (https://dev.entando.org/));
- требования к интеграции;
- анализ при наличии отсутствующих данных;
- мобильную поддержку.

Определение интеграции

В мире контейнеров интеграция превращается в новую проблему. В традиционном корпоративном ландшафте интеграция осуществляется с помощью централизованного решения (Enterprise Service Bus или чего-то подобного) или некоего общего интеграционного фреймворка, такого как Apache Camel. Ни один из подходов не является идеальным для контейнерной платформы без сохранения состояния (stateless). Вам нужна

плавная интеграция между компонентами обмена сообщениями, логикой преобразования данных и интеграцией сервисов. Все соответствующие части должны хорошо масштабироваться в среде без сохранения состояния, характерной для распределенных систем, и составное приложение должно легко расширяться с помощью новых возможностей:

- существующих возможностей интеграции;
- оценки партнерской экосистемы решений;
- определения требований к интеграции (источники данных, интеграция сервисов, обмен сообщениями, API);
- определения стандартов и фреймворков (например, Camel K (https://oreil.ly/kfXw1));
- оценки бессерверной интеграции (например, Camel K).

Определение стека технологий

Последняя категория фокусируется на отдельных технологиях и фреймворках. Рассматривайте ее как общий план, определяющий соответствующие технологии, сервисы и методологии для промышленной среды. Недооцененное влияние на требования в этой категории оказывает имеющийся в организации опыт разработки. Учитывая традиционный опыт Enterprise Java, будет непросто полностью переключиться на реактивный подход к разработке и проектирование stateless-приложений. Кроме того, знакомство с существующими API и время перехода на новую платформу играют решающую роль в выборе наиболее подходящего стека технологий. В этой категории мы рассматриваем такие возможности, как:

- оценка стека технологий для основных, клиентских и внешних сервисов;
- поддержка фреймворка микросервисов (например, Quarkus, Spring Boot);
- рекомендации по внедрению (реактивные, императивные, управляемые сообщениями и т. д.);

- модель развертывания (IaaS, PaaS, гибридная);
- определение целевой платформы разработки;
- анализ пробелов в навыках разработки.

Оценка всех этих пунктов поможет вам хорошо подготовиться к переходу на контейнерную платформу приложений. Затем вам нужно будет наметить и спланировать стратегию контейнеризации.

Обязательные шаги и инструменты миграции

Следуя базовому предположению о наличии готового ландшафта приложений и невозможности начать все с нуля, мы подчеркиваем важность переноса существующих приложений в контейнеры. Возвращаясь к рассмотренным выше стратегиям 6R, первое приложение, на которое вы обратите внимание, должно подпадать под одно из следующих R: Rehost (перенос), Replatform (смена платформы) или Refactor (рефакторинг) (рис. 3.3). Несмотря на сходство их описаний, наиболее существенное различие между этими тремя подходами заключается в ценности для бизнеса, в количестве времени и стоимости миграции.

Какие действия предпринять и с чего начать модернизацию, зависит от приложения. Конкретные шаги могут различаться, но первое, что нужно сделать, — выбрать правильных кандидатов. Поэтому необходимо проанализировать существующие приложения, классифицировать их и сгруппировать, чтобы назначить им подходящие паттерны миграции. Последний шаг — реализация отдельных проектов миграции.

Создание портфеля приложений. Существует много способов создать такой каталог (или портфель) приложений. И у вас, скорее всего, уже есть способ выбрать приложения, релевантные для определенной предметной области. Если нет, то смело переходите к главе 5, в которой мы рассказываем о проекте Konveyor (https://oreil.ly/1wPUF).

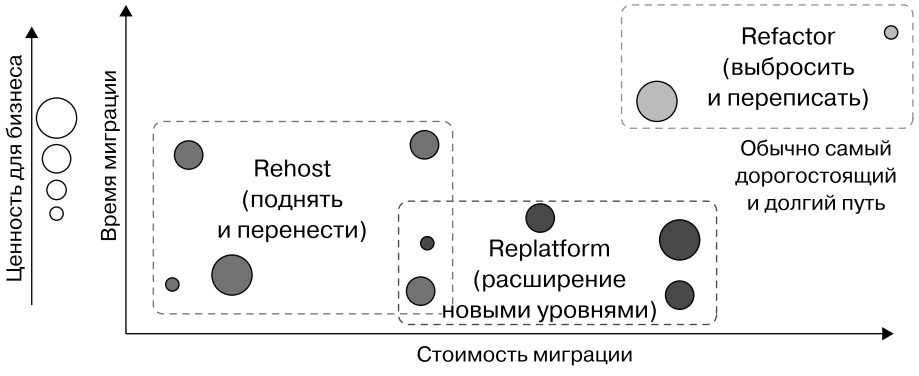


Рис. 3.3. Паттерн миграции рабочей нагрузки

Готовьтесь к большим делам

Самый престижный процесс модернизации — рефакторинг существующих приложений. Проверенный метод переноса существующей монолитной системы на микросервисную архитектуру описан в книге Сэма Ньюмена (Sam Newman) *Monolith to Microservices* (O'Reilly; https://oreil.ly/0x6oq)[1]. Она знакомит с множеством различных подходов и подробно описывает процессы для различных ситуаций.

Но есть и более простые подходы, например описанный Брентом Фраем (Brent Frye), сотрудником института разработки программного обеспечения Университета Карнеги — Меллона. Фрай предлагает гораздо более общий подход к разделению существующих приложений на модули (https://oreil.ly/YKYUY) и рекомендует восемь простых шагов, помогающих раздробить монолит. В своем описании Фрай фокусируется на компонентах и группах компонентов. *Компоненты* — это логические наборы объектов данных и действий, которые система выполняет с этими объектами. Группы компонентов становятся тем, что Фрай называет

[1] *Ньюмен С.* От монолита к микросервисам.

макросервисами (macroservices). Макросервис похож на микросервис, но имеет важные отличия. Во-первых, он может использовать хранилище данных совместно с устаревшей монолитной системой или другими макросервисами. Во-вторых, в отличие от микросервиса макросервис может предоставлять доступ к нескольким объектам данных. На последнем шаге макросервисы подвергаются дальнейшей декомпозиции и преобразуются в микросервисы.

Логические шаги по разрушению монолита, предлагаемые Фраем, представлены ниже.

1. Определите логические компоненты.
2. Упростите и реорганизуйте компоненты.
3. Определите зависимости компонентов.
4. Определите группы компонентов.
5. Создайте API для удаленного пользовательского интерфейса.
6. Перенесите группы компонентов в макросервисы:
   1) перенесите группы компонентов в отдельные проекты;
   2) создайте отдельные развертывания.
7. Преобразуйте макросервисы в микросервисы.
8. Повторяйте шаги 6–7, пока процесс не будет завершен.

Еще одна общая рекомендация предложена Крисом Ричардсоном (Chris Richardson). Как он подчеркивал в своем выступлении на O'Reilly SACON London (https://oreil.ly/CZn61) и много раз после этого, Крис предпочитает поэтапный подход, начинающийся с извлечения наиболее многообещающей функциональности.

Делайте это поэтапно и повторяйте шаги извлечения до тех пор, пока монолит окончательно не исчезнет или не будут решены первоначальные проблемы с доставкой программного обеспечения (рис. 3.4).

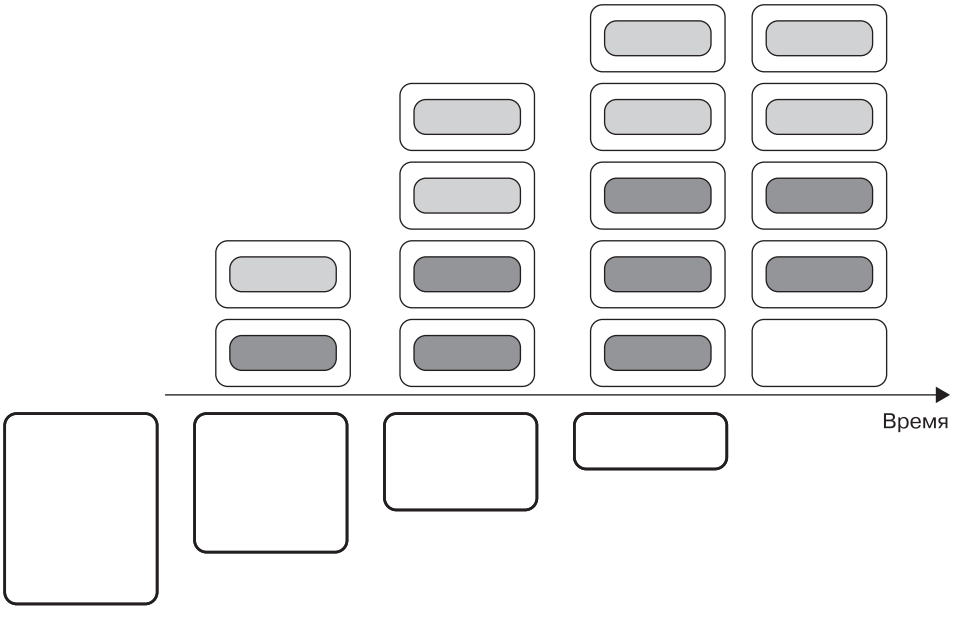


Рис. 3.4. Перемещение монолитов в сервисы с течением времени путем постепенного извлечения функциональности

Эти три подхода различаются по глубине и составляющим. Если Ричардсон говорит о наиболее ценных функциях и предлагает извлекать их в первую очередь, то Фрай создал простую методологию, которую можно применять в любых ситуациях. Наконец, Ньюмен разработал наиболее подробное руководство для различных ситуаций на пути модернизации. Все три пригодятся вам в вашем путешествии. Однако мы убеждены, что подход Ричардсона — лучшая отправная точка. Мы твердо верим в то, что сказал Томас Хейскенс (Thomas Huijskens), выступая перед специалистами по данным: «Код, который вы пишете, полезен только в том случае, если является промышленным».

Все усилия по модернизации должны соответствовать бизнес-требованиям и поддерживать промышленную функциональность. Как следствие, весь проект модернизации может быть успешным, только если вы определите правильных кандидатов.

Резюме

В этой главе вы познакомились с некоторыми определениями основных стратегий миграции и путями оценки целевой платформы разработки. Мы рассмотрели технические рекомендации, и теперь вы знаете, как оценивать существующие приложения для переноса (rehosting), смены платформы (replatforming) и рефакторинга (refactoring).

ГЛАВА 4

Платформа разработки программного обеспечения на основе Kubernetes

В предыдущей главе мы описали нашу методологию модернизации и шаги, необходимые для проектирования и разработки современных архитектур. Мы обсудили потребность в такой платформе, как Kubernetes, которая может помочь сделать ваши приложения облачными, готовыми к масштабированию пропорционально потребностям вашего бизнеса.

Мы также показали, что архитектура на основе микросервисов обычно реализуется с помощью технологии контейнеров, которая обеспечивает переносимость и согласованность приложений. Теперь посмотрим, как Kubernetes может помочь модернизировать наши Java-приложения и какие шаги нужно предпринять для этого, используя декларативный подход с помощью богатого API этого фреймворка.

Разработчики и Kubernetes

Kubernetes (https://kubernetes.io/), что в переводе с греческого означает «пилот» или «управляющий», — это проект с открытым исходным кодом, который в настоящее время де-факто является целевой средой для современных архитектур и самой популярной платформой оркестрации контейнеров. Простая схема кластера представлена на рис. 4.1. Начинавшийся с экспериментов Google по управлению сложными распределенными приложениями для их программного стека еще в 2015 году, в настоящее время этот проект превратился в одно из крупнейших сообществ в мире свободного программного обеспечения. Он управляется фондом Cloud Native Computing Foundation (CNCF) и поддерживается производителями и отдельными участниками.

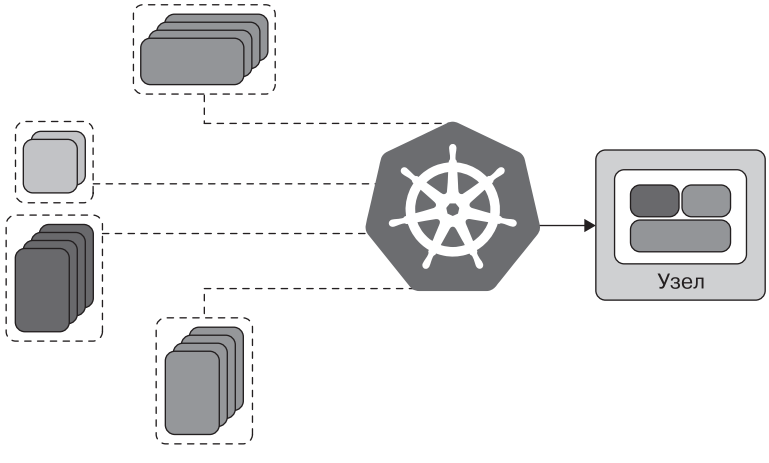


Рис. 4.1. Кластер Kubernetes с запущенными приложениями в узлах

Основное внимание в этой платформе оркестрации контейнеров уделяется обеспечению правильной работы приложений, предоставлению готовых функций самоисправления и восстановления, а также эффективного API, предназначенного для управления этим механизмом. Вы можете спросить: почему я как разработчик должен заботиться о Kubernetes, если он самостоятельный?

Хороший вопрос, и, возможно, хорошим ответом на него будет аналогия. Представьте, что у вас есть болид «Формулы-1» с автопилотом. Но если вы хотите выиграть гонку, то вам нужно подготовить и настроить свою машину, чтобы она могла конкурировать с другими хорошими машинами. То же относится и к приложениям, которые могут пользоваться всеми возможностями платформы для оптимальной работы.

Что делает Kubernetes

Используя Kubernetes в роли целевой платформы для запуска ваших приложений, вы можете положиться на экосистему API и компонентов, созданных для упрощения развертывания, чтобы разработчики могли сосредоточиться только на самой важной части: программировании. Kubernetes предоставляет платформу для бесперебойной работы распределенных систем (https://oreil.ly/DNRQS).

На практике это означает, что не нужно повторно реализовывать пользовательские решения, когда речь идет о таких аспектах, как:

- *обнаружение сервисов* — Kubernetes использует внутренний DNS для обнаружения ваших приложений; этот механизм действует автоматически и позволяет направлять трафик нескольким экземплярам вашего приложения;
- *балансировка нагрузки* — Kubernetes заботится об управлении нагрузкой на ваши приложения, балансировке трафика и распределении пользовательских запросов;
- *самовосстановление* — Kubernetes автоматически обнаруживает и заменяет неисправные контейнеры, предоставляя механизмы по умолчанию для проверки работоспособности и самовосстановления;
- *развертывание и откат* — Kubernetes гарантирует стабильную работу вашего приложения в желаемом состоянии, обеспечивая контроль над масштабированием рабочих нагрузок. Кроме того, он предлагает возможность развертывания или отката до определенной версии вашего приложения.

Чего не делает Kubernetes

Многие проблемы, с которыми часто приходится сталкиваться разработчикам в промышленной среде, уже решены и делегированы платформе, основной целью которой является обеспечение работы приложений. Но обеспечивает ли эта платформа все, что необходимо для модернизации приложений? Вероятно, нет.

Как отмечалось в предыдущей главе, шаги модернизации в сторону облачного подхода тесно связаны с методологией, а не с конкретной технологией. Изменив свое мышление с создания монолитных приложений на создание микросервисов, вы сможете начать мыслить более масштабно. В настоящее время многие приложения работают на облачных платформах, основанных на Kubernetes, и именно они поддерживают глобальные рабочие нагрузки. Ниже описаны некоторые моменты, которые следует учитывать.

- Kubernetes не знает, как обращаться с вашим приложением. Она может перезапустить его в случае сбоя, но не знает причин происходящего, поэтому нам важно иметь полный контроль над нашей архитектурой на основе микросервисов и возможность отладить каждый контейнер. Это особенно важно в случае крупномасштабного развертывания.
- Kubernetes не предоставляет промежуточного ПО или сервисов прикладного уровня. Задачи обнаружения сервисов необходимо решать путем взаимодействия с Kubernetes API или полагаться на какой-либо сервис, работающий поверх Kubernetes, например, на фреймворк сервисной сетки. Не существует готовой экосистемы для разработчиков, «из коробки».
- Kubernetes не создает приложений. Вы сами должны предоставить свои приложения, скомпилированные и упакованные в образы контейнеров, или использовать дополнительные компоненты поверх Kubernetes.

Учитывая все это, начнем путешествие по Kubernetes для разработчиков, чтобы сделать первый шаг к переносу приложения в облачную промышленную среду.

Инфраструктура как код

Kubernetes предоставляет набор API для управления желаемым состоянием приложения, а также всей платформы. Каждый компонент в Kubernetes имеет API, доступный для использования. Kubernetes предлагает паттерн декларативного развертывания (https://oreil.ly/cURvG), позволяющий автоматизировать обновление и откат групп подов (pod). Декларативный подход является гранулированным и используется для расширения Kubernetes API с помощью концепции *пользовательских ресурсов*.

Пользовательские ресурсы — это расширения Kubernetes API. Пользовательский ресурс определяет настройку конкретной установки Kubernetes, добавляя дополнительные объекты, расширяющие возможности кластера. Дополнительную информацию об этом можно получить в официальной документации Kubernetes (https://oreil.ly/cVBnl).

Часть основных объектов, с помощью которых вы сможете управлять приложением в Kubernetes, перечислены ниже.

- *Pod (под)* — группа из одного или нескольких контейнеров, развернутых в кластере Kubernetes. Это сущность, которой управляет Kubernetes, поэтому любое приложение, упакованное как контейнер, должно быть объявлено как под.
- *Service (сервис)* — ресурс, отвечающий за обнаружение сервисов и балансировку нагрузки. Чтобы под можно было обнаружить и использовать, его необходимо отобразить в объект Service.

- *Deployment (развертывание)* — описывает жизненный цикл приложения, управляет созданием подов с точки зрения того, какие образы использовать для развертывания приложения, сколько должно быть подов и как их следует обновлять. Кроме того, этот ресурс помогает определить способы проверки работоспособности и ограничения ресурсов для вашего приложения.

Каждый из этих объектов, наряду с другими ресурсами в кластере, можно определить и контролировать с помощью представления YAML или Kubernetes API. Существуют также другие полезные объекты API, например связанные с хранилищем (PersistentVolume) или используемые специально для управления приложениями с состоянием (StatefulSet). В этой главе мы сосредоточимся на самых основных объектах, необходимых для запуска вашего приложения на платформе Kubernetes.

Образы контейнеров

Первый шаг в этом путешествии — контейнеризация микросервисов, чтобы их можно было развернуть в Kubernetes в виде подов. Его можно реализовать в виде файла YAML, вызова API или с помощью Java-клиента Kubernetes.

Для создания своего первого образа контейнера вы можете использовать Quarkus-микросервис Inventory из приложения Coolstore. Контейнеры определяются манифестом — так называемым файлом Dockerfile или Containerfile, в котором определяется программный стек уровней от уровня операционной системы до уровня двоичного файла вашего приложения. Этот подход имеет многочисленные преимущества: легко отслеживать версии, наследовать существующие уровни, добавлять уровни и расширять контейнеры. Схема такой многоуровневой организации показана на рис. 4.2.

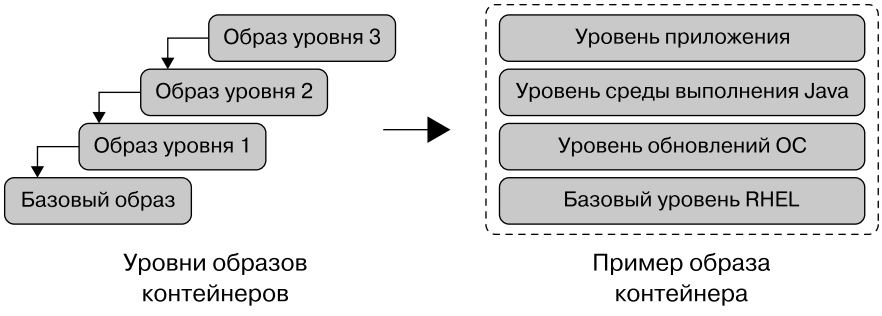


Рис. 4.2. Уровни в образе контейнера

Dockerfile

Файл Dockerfile для простых случаев использования выглядит просто. Несколько основных директив, называемых *инструкциями*, перечислены в табл. 4.1.

Таблица 4.1. Инструкции для использования в файлах Dockerfile

Инструкция	Описание
`FROM`	Задает базовый образ. Это может быть, например, дистрибутив Linux, такой как `fedora`, `centos`, `rhel`, `ubuntu`
`ENV`	Определяет переменную среды для контейнера. Определяемые переменные будут видны приложению и могут устанавливаться во время выполнения
`RUN`	Определяет команду для выполнения на текущем уровне. Это может быть установка пакета или запуск приложения
`ADD`	Копирует файлы с рабочей станции на уровень контейнера, например файл JAR или файл конфигурации
`EXPOSE`	Если приложение прослушивает какой-либо порт, то эта инструкция позволяет открыть порт для доступа из сети контейнеров, чтобы Kubernetes мог отобразить его в под или в объект Service
`CMD`	Команда запуска приложения: последний шаг в процессе создания образа контейнера, который выполняется, когда имеются все необходимые зависимости в слоях и можно безопасно запустить приложение

Процесс создания контейнера из файла Dockerfile также показан на рис. 4.3.

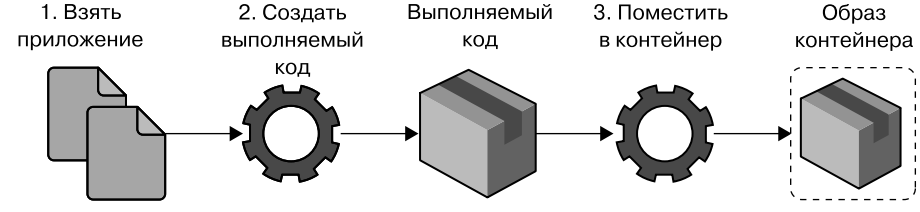


Рис. 4.3. Создание образа контейнера

Ниже приводится пример файла Dockerfile для Quarkus-микросервиса Inventory на Java, созданного в главе 2. Его также можно найти в репозитории GitHub (https://oreil.ly/D9u1k):

```
FROM registry.access.redhat.com/ubi8/openjdk-11 ❶
ENV PROFILE=prod ❷
ADD target/*.jar app.jar ❸
EXPOSE 8080 ❹
CMD java -jar app.jar ❺
```

❶ Сборка образа контейнера начинается со слоя OpenJDK 11.

❷ Задается переменная среды, с помощью которой в приложении можно различать загружаемые профили или конфигурации.

❸ Артефакт JAR, созданный во время компиляции, копируется в образ контейнера. Предполагается, что перед этим приложение было скомпилировано в файл `fat-jar` или `uber-jar`, содержащий все зависимости.

❹ Открывается порт 8080 для доступа из сети контейнеров.

❺ Запуск приложения путем вызова артефакта, скопированного в уровень.

В этом подразделе мы определили файл Dockerfile с минимальным набором инструкций, необходимым для создания образа контейнера. Теперь посмотрим, как создавать образы контейнеров из Dockerfile.

Сборка образов контейнеров

Теперь создадим образ контейнера. Docker (https://www.docker.com/) — популярный проект с открытым исходным кодом для создания контейнеров. Вы можете скачать версию для своей операционной системы и начать создавать и запускать контейнеры. Podman (https://podman.io/) — еще одна альтернатива с открытым исходным кодом, которая тоже может генерировать объекты Kubernetes.

Если на вашей рабочей станции установлен Docker или Podman, можете запустить сборку контейнера из Dockerfile с помощью следующей команды:

```
docker build -f Dockerfile -t docker.io/modernizingjavaappsbook/
  inventory-quarkus:latest
```

Она создаст образ контейнера, прочитав инструкции из файла Dockerfile. Затем добавит в образ метку в формате `<репозиторий>/<имя>:<тег>`, в данном случае `docker.io/modernizingjavaappsbook/inventory-quarkus:latest`. В консоли появится следующий вывод:

```
STEP 1: FROM registry.access.redhat.com/ubi8/openjdk-11
Getting image source signatures
Copying blob 57562f1b61a7 done
Copying blob a6b97b4963f5 done
Copying blob 13948a011eec done
Copying config 5d89ab50b9 done
Writing manifest to image destination
Storing signatures
STEP 2: ENV PROFILE=prod
STEP 3: ADD target/*.jar app.jar
STEP 4: EXPOSE 8080
STEP 5: CMD java -jar app.jar
STEP 6: COMMIT inventory-quarkus:latest
Getting image source signatures
Copying blob 3aa55ff7bca1 skipped: already exists
Copying blob 00af10937683 skipped: already exists
Copying blob 7f08faf4d929 skipped: already exists
Copying blob 1ab317e3c719 done
Copying config b2ae304e3c done
Writing manifest to image destination
Storing signatures
--> b2ae304e3c5
b2ae304e3c57216e42b11d8be9941dc8411e98df13076598815d7bc376afb7a1
```

Собранный образ контейнера, готовый к локальному использованию, будет сохранен в локальном хранилище Docker или Podman, называемом *кэшем Docker* или *кэшем контейнеров*.

Вы можете создать файл uber-jar, который будет использоваться в промышленной среде, с помощью команды ./mvnw package -Dquarkus.profile=prod. Кроме того, можно позволить Docker или Podman самим скомпилировать ваше программное обеспечение и создать контейнер, использовав особый тип сборки образов контейнеров, который называется многоэтапным (https://oreil.ly/HzhDj). Для примера взгляните на этот файл Dockerfile: https://oreil.ly/UK3c2.

Запуск контейнеров

Под *запуском контейнеров* подразумевается извлечение образов контейнеров из кэша контейнеров и запуск приложений в них. Этот процесс будет изолирован внутри среды выполнения контейнеров (например, Docker или Podman) от других процессов на вашей рабочей станции и представляет переносимое приложение со всеми зависимостями, управляемыми внутри контейнера, а не на рабочей станции.

Чтобы начать тестирование микросервиса Inventory, упакованного в образ контейнера, можно запустить следующую команду:

```
docker run -ti docker.io/modernizingjavaappsbook/inventory-quarkus:latest
```

После этого вы сможете убедиться, что микросервис запущен и работает в контейнере, прослушивая порт 8080. Оба фреймворка, Docker и Podman, позаботятся об отображении сети контейнера в сеть вашей рабочей станции; откройте браузер, введите адрес http://localhost:8080, и вы увидите страницу приветствия Quarkus (см. рис. 2.4).

Дополнительную информацию об отображении портов и сетей внутри контейнеров и хостов, на которых работает Docker, можно найти в документации Docker Network (https://oreil.ly/ja9Iu).

Реестр

Как отмечалось в предыдущем подразделе, образы контейнеров хранятся в локальном кэше. Однако если вы хотите сделать их доступными за пределами вашей рабочей станции, то нужно иметь возможность передавать их каким-либо удобным способом. Размер образа контейнера обычно составляет сотни мегабайт. Вот почему требуется реестр образов контейнеров.

Реестр, по сути, служит местом хранения образов контейнеров и совместного их использования с помощью процесса выгрузки и загрузки. Как только образ окажется в другой системе, приложение, содержащееся в нем, может быть запущено в ней.

Реестры могут быть общедоступными или частными. В числе популярных общедоступных реестров можно назвать Docker Hub (https://hub.docker.com/) и Quay.io (https://quay.io/). Они предлагаются как SaaS (Software as a Service — программное обеспечение как сервис) в Интернете и позволяют хранить общедоступные образы с аутентификацией или без нее. Частные реестры обычно предназначены для конкретных пользователей и недоступны для общего пользования. Однако их можно сделать доступными для частных сред, таких как частные кластеры Kubernetes.

В этом примере мы создали в DockerHub организацию для книги под названием `modernizingjavaappsbook`, отображаемую (map) в репозиторий этого общедоступного реестра, куда мы попробуем отправить образ нашего контейнера.

Для этого нужно войти в реестр (пройти аутентификацию), чтобы иметь возможность отправлять новый контент, после чего вы сможете выгрузить образ контейнера, который будет считаться общедоступным:

```
docker login docker.io
```

Успешно пройдя аутентификацию, можно начать выгрузку образа контейнера Inventory в реестр:

```
docker push docker.io/modernizingjavaappsbook/inventory-quarkus:latest
```

Эта команда отправляет образы в реестр, а в подтверждение вы должны получить примерно такой вывод:

```
Getting image source signatures
Copying blob 7f08faf4d929 done
Copying blob 1ab317e3c719 done
Copying blob 3aa55ff7bca1 skipped: already exists
Copying blob 00af10937683 skipped: already exists
Copying config b2ae304e3c done
Writing manifest to image destination
Storing signatures
```

Микросервис Quarkus, упакованный в образ контейнера, теперь готов к развертыванию в любом месте!

Развертывание в Kubernetes

Развертывание приложений в Kubernetes осуществляется путем взаимодействия с Kubernetes API и создания объектов, представляющих желаемое состояние приложения в кластере Kubernetes. Как уже говорилось, поды, объекты Service и Deployment — это минимальный набор объектов, которые требуется создать, чтобы Kubernetes мог управлять жизненным циклом приложения и подключением к нему.

Если у вас еще нет кластера Kubernetes, то можете скачать и использовать minikube (https://oreil.ly/n2Kgx), автономный кластер Kubernetes, предназначенный для локальной разработки.

Каждый объект в Kubernetes содержит следующие значения:

- `apiVersion` — версия Kubernetes API, с помощью которой должен создаваться этот объект;
- `kind` — тип объекта (например, Pod, Service);

- `metadata` — фрагменты информации, помогающие однозначно идентифицировать объект, например имя или UID;
- `spec` — желаемое состояние объекта.

Итак, мы определили базовую структуру любых объектов Kubernetes, а теперь рассмотрим основные объекты, необходимые для запуска приложений поверх Kubernetes.

Под

Под (https://oreil.ly/KQk2T) — группа из одного или нескольких контейнеров, имеющих общее хранилище и сетевые ресурсы, а также спецификацию запуска контейнеров. На рис. 4.4 показано представление двух подов в кластере Kubernetes с IP-адресами, назначаемыми каждому из них.

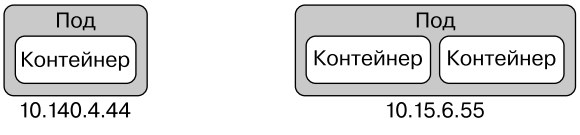


Рис. 4.4. Поды и контейнеры

Kubernetes не работает с контейнерами напрямую, а опирается на концепцию пода. Поэтому вы должны предоставить определение пода, соответствующее вашему контейнеру:

```
apiVersion: v1
kind: Pod
metadata:
  name: inventory-quarkus ❶
  labels:
    app: inventory-quarkus ❷
spec:
  containers: ❸
    - name: inventory-quarkus
      image: docker.io/modernizingjavaappsbook/inventory-quarkus:latest ❹
      ports:
        - containerPort: 8080 ❺
```

❶ Имя объекта Pod, уникальное для каждого пространства имен.

❷ Список пар «ключ — значение», применяемых к этому объекту.

❸ Список контейнеров, используемых в этом поде.

❹ URI образа контейнера, в данном случае общедоступный репозиторий в Docker Hub.

❺ Порт, предоставляемый этим контейнером, для отображения в поде.

Как правило, один под содержит один контейнер, поэтому под соответствует одному приложению. В некоторых случаях в один под можно включить несколько контейнеров (например, sidecar-контейнеров), но наилучшей практикой считается помещать одно приложение в один под, поскольку это упрощает масштабирование и повышает удобство сопровождения.

Любой из объектов Kubernetes, описанных выше, можно создать в виде файла YAML и с помощью `kubectl` — инструмента командной строки Kubernetes. Запустите следующую команду, чтобы развернуть свой первый микросервис в виде отдельного пода. Его можно найти в репозитории книги на GitHub (https://oreil.ly/YF8bT):

```
kubectl create -f pod.yaml
```

Убедитесь, что он работает:

```
kubectl get pods
```

Вы должны увидеть примерно такой вывод:

```
NAME               READY  STATUS   RESTARTS  AGE
inventory-quarkus  1/1    Running  0         30s
```

Обратите внимание на столбец `STATUS`, он показывает, что под работает правильно и все проверки работоспособности, применяемые по умолчанию, завершаются успехом.

Дополнительные сведения о том, как выполнять более детальные проверки работоспособности, вы найдете в официальной документации Kubernetes, в разделе, описывающем настройку проверок работо- и жизнеспособности (https://oreil.ly/sOdnL).

Объект Service

Объекты Service в Kubernetes (https://oreil.ly/fOfpN) используются для предоставления доступа к приложению, работающему в наборе подов. Это очень полезные объекты, поскольку под получает случайный IP-адрес в сети Kubernetes, который может измениться, если перезапустить под или переместить на другой узел в кластере Kubernetes, а объекты Service предлагают более последовательный способ связи с подами, выступая в роли DNS-сервера и балансировщика нагрузки.

Объект Service отображается в один или несколько подов, использует внутренний DNS для определения внутреннего IP-адреса по мнемоническому короткому имени хоста (например, `inventory-quarkus`) и балансирует трафик между подами (рис. 4.5). Каждый объект Service получает свой IP-адрес из выделенного диапазона IP-адресов, который отличается от диапазона IP-адресов подов.

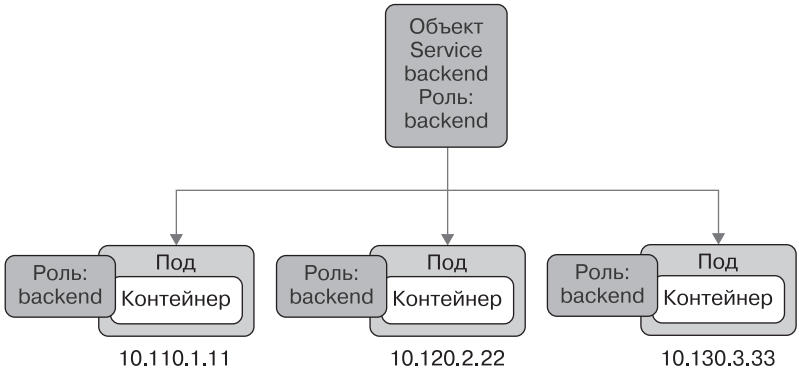


Рис. 4.5. Объект Service в Kubernetes

Метод балансировки, предлагаемый объектами Service в Kubernetes, — это уровень 4 (TCP/UDP). Поддерживаются только две стратегии балансировки: круговой алгоритм и по IP-адресу источника. Для балансировки прикладного уровня (например, HTTP) используются другие объекты, такие как Ingress, которые не рассматриваются в этой книге, но вы можете найти информацию о них в документации (https://oreil.ly/VBvOu).

Рассмотрим объект Service, отображающий наш под:

```
apiVersion: v1
kind: Service
metadata:
  name: inventory-quarkus-service ❶
spec:
  selector:
    app: inventory-quarkus ❷
  ports:
    - protocol: TCP ❸
      port: 8080 ❹
      targetPort: 8080 ❺
```

❶ Имя объекта Service.

❷ Метка, предоставляемая подом для сопоставления с объектом Service.

❸ Используемый протокол L4, TCP или UDP.

❹ Порт, используемый этим объектом Service.

❺ Порт, используемый подом и отображаемый объектом Service.

Чтобы создать свой объект Service, запустите следующую команду. Его также можно найти в репозитории книги на GitHub (https://oreil.ly/e13Dd):

```
kubectl create -f service.yaml
```

Убедитесь, что он работает:

```
kubectl get svc
```

Вы должны увидеть примерно такой вывод:

```
NAME                        TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)   AGE
inventory-quarkus-service   ClusterIP   172.30.34.73  <none>       8080/TCP  6s
```

Вы только что определили объект Service, соответствующий поду. Он будет доступен только из внутренней сети Kubernetes, если не предоставить объект, принимающий трафик извне, такой как Ingress.

Объект Deployment

Deployment — это объект Kubernetes, предназначенный для управления жизненным циклом приложения. Объект описывает желаемое состояние, и Kubernetes реализует его, используя стратегии *rolling* (постепенного обновления) или *re-create* (повторного создания). Жизненный цикл объекта Deployment включает состояния выполнения, завершения и сбоя. Deployment выполняется во время обновления, например, при обновлении или масштабировании подов.

Объекты Deployment в Kubernetes предлагают набор дополнительных возможностей сверх базовых концепций Pod и Service (рис. 4.6):

- развертывание ReplicaSet или Pod;
- обновление подов и наборов реплик ReplicaSet;
- откат к предыдущим версиям развертывания;
- масштабирование развертывания;
- приостановку или возобновление развертывания;
- определение проверок работоспособности;
- определение ограничений ресурсов.

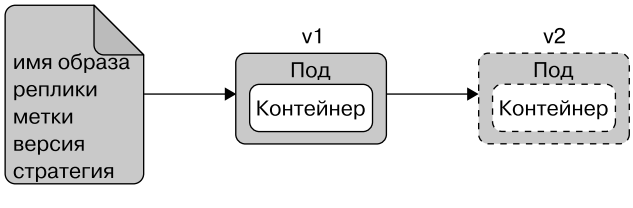


Рис. 4.6. Объекты Deployment управляют жизненным циклом приложения и обновлениями

Управление приложениями с помощью объектов Deployment Kubernetes дает возможность определить способ обновления. Основное преимущество Deployment — предсказуемость запуска и остановки набора подов. В Kubernetes поддерживается две стратегии развертывания приложений.

- *Постепенное обновление (rolling)*. Обеспечивает контролируемую поэтапную замену подов приложения и гарантирует, что всегда будет доступно некоторое минимальное их количество. Это помогает обеспечить непрерывную работу приложения, когда трафик не направляется в новую версию приложения до тех пор, пока не будут выполнены проверки работоспособности на желаемом количестве развернутых подов.
- *Повторное создание (re-create)*. Удаляет все существующие поды перед созданием новых. Следуя этой стратегии, Kubernetes сначала завершает работу всех контейнеров текущей версии, а затем, когда все старые контейнеры остановятся, одновременно запускает все новые контейнеры. При этом возникает интервал, когда приложение простаивает, но гарантируется, что несколько версий не будут работать одновременно.

Ниже приводится пример объекта Deployment, управляющего развертыванием подов:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: inventory-quarkus-deploy ❶
  labels:
    app: inventory-quarkus ❷
spec:
  replicas: 1 ❸
  selector:
    matchLabels:
      app: inventory-quarkus ❹
  template: ❺
    metadata:
      labels:
        app: inventory-quarkus
    spec:
      containers:
      - name: inventory-quarkus
```

```
image: docker.io/modernizingjavaappsbook/inventory-quarkus:latest ❻
ports:
- containerPort: 8080
readinessProbe: ❼
  httpGet:
    path: /
    port: 8080
    scheme: HTTP
  periodSeconds: 10
  successThreshold: 1
  failureThreshold: 3
livenessProbe: ❽
  httpGet:
    path: /
    port: 8080
    scheme: HTTP
  periodSeconds: 10
  successThreshold: 1
  failureThreshold: 3
```

❶ Имя объекта Deployment.

❷ Метка этого объекта.

❸ Желаемое количество реплик подов.

❹ Селектор для поиска подов с помощью меток.

❺ Используемый шаблон пода, включая метки для наследования или контейнеры для создания.

❻ Используемый образ контейнера.

❼ Kubernetes использует проверки готовности, чтобы узнать, когда контейнер готов начать принимать трафик, а под считается готовым, когда готовы все его контейнеры. Здесь проверка готовности определяется как работоспособность корневого пути HTTP.

❽ Kubernetes использует проверки работоспособности, чтобы узнать, когда перезапустить контейнер. Здесь проверка работоспособности определяется как работоспособность корневого пути HTTP.

Выполните следующую команду, чтобы создать объект Deployment. Его также можно найти в репозитории книги на GitHub (https://oreil.ly/PWucG):

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

Выполните следующую команду, чтобы убедиться, что объект Deployment создан, и получить его состояние:

```
kubectl get deploy
```

Вы должны увидеть примерно такой вывод:

```
NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
inventory-quarkus-deploy   1/1     1            1           10s
```

Обратите внимание на столбец `READY`, он верно отображает желаемое состояние — Deployment запрашивает одну реплику для микросервиса Inventory, работающего в Kubernetes. При желании можете повторно проверить, создан ли под:

```
kubectl get pods
```

Вы должны увидеть примерно такой вывод:

```
NAME                                       READY   STATUS    RESTARTS   AGE
inventory-quarkus                          1/1     Running   0          1m
inventory-quarkus-deploy-5cb46f5d8d-fskpd  1/1     Running   0          30s
```

Как видите, был создан новый под со случайно сгенерированным именем, начинающимся с имени объекта Deployment `inventory-quarkus-deploy`. Если приложение даст сбой или мы удалим под, управляемый объектом Deployment, то Kubernetes автоматически создаст его заново. Но имейте в виду, что это не относится к подам, сгенерированным без использования объекта Deployment:

```
kubectl delete pod inventory-quarkus inventory-quarkus-deploy-5cb46f5d8d-fskpd
```

Можете сами убедиться, что Kubernetes всегда старается поддерживать желаемое состояние:

```
kubectl get pods
```

Вы должны увидеть примерно такой вывод:

```
NAME                                       READY   STATUS    RESTARTS   AGE
inventory-quarkus-deploy-5cb46f5d8d-llp7n  1/1     Running   0          42s
```

Kubernetes и Java

Kubernetes обладает огромным потенциалом в управлении жизненным циклом приложений, поэтому разработчики и архитекторы довольно активно ищут и изучают передовые приемы внедрения возможностей Kubernetes в архитектуру, такие как паттерны. Паттерны Kubernetes представляют собой паттерны проектирования приложений и сервисов на основе контейнеров.

С точки зрения Java-разработчика, первым шагом является переход от монолитного решения к решению на основе микросервисов. Следующий шаг — вход в контекст Kubernetes и максимальное использование преимуществ, предлагаемых этой платформой, таких как: расширяемость API, декларативная модель и стандартизированный процесс, к которым стремится IT-индустрия.

Существуют фреймворки Java, которые помогают разработчикам преобразовывать свои приложения в контейнеры и запускать их в Kubernetes. Мы уже контейнеризировали Quarkus-микросервис Inventory с помощью Dockerfile. Теперь запустим контейнеризацию из Java, сгенерировав образ контейнера для микросервиса Catalog на основе Spring Boot с помощью Maven и Gradle.

Jib

Jib (https://oreil.ly/N2jRr) — это фреймворк с открытым исходным кодом, разработанный компанией Google для создания образов контейнеров, совместимых с форматом образов Open Container Initiative (OCI), без использования Docker или какой-либо другой среды выполнения контейнеров. Контейнеры можно создавать из любого кода на Java, благодаря плагину для Maven и Gradle. Это означает, что разработчики на Java могут контейнеризировать свое приложение, обходясь без создания и поддержки каких-либо файлов Dockerfile, делегируя эту работу Jib.

Основные преимущества этого подхода (https://oreil.ly/2y92D) представлены ниже.

- *Код исключительно на Java.* Не требуется знать особенности Docker или Dockerfile; достаточно добавить Jib как плагин, и он сам создаст образ контейнера. Полученный образ обычно называют «бездистрибутивным», поскольку он не наследует никакой другой базовый образ.
- *Скорость.* Приложение разделено на несколько уровней по зависимостям от классов. Нет необходимости повторно собирать образ контейнера, как это приходится делать при использовании файлов Dockerfile; фреймворк Jib сам позаботится о развертывании изменившихся уровней.
- *Воспроизводимость.* Не производится никаких ненужных обновлений, и одно и то же содержимое всегда генерирует один и тот же образ.

Самый простой способ запустить сборку образа контейнера с помощью Jib — добавить плагин Maven через командную строку:

```
mvn compile com.google.cloud.tools:jib-maven-plugin:2.8.0:build
  -Dimage=<MY IMAGE>
```

То же самое можно сделать, добавив Jib как плагин в `pom.xml`:

```
<project>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>com.google.cloud.tools</groupId>
        <artifactId>jib-maven-plugin</artifactId>
        <version>2.8.0</version>
        <configuration>
          <to>
            <image>myimage</image>
          </to>
        </configuration>
      </plugin>
      ...
```

```
    </plugins>
  </build>
  ...
</project>
```

При этом есть возможность управлять другими настройками, такими как аутентификация или параметры сборки. Запустите следующую команду, если хотите создать сервис Catalog и отправить его непосредственно в Docker Hub:

```
mvn compile com.google.cloud.tools:jib-maven-plugin:2.8.0:build
 -Dimage=docker.io/modernizingjavaappsbook/catalog-spring-boot:latest
 -Djib.to.auth.username=<USERNAME>
 -Djib.to.auth.password=<PASSWORD>
```

Управление аутентификацией здесь производится с помощью параметров командной строки, но Jib может также осуществлять аутентификацию с помощью Docker CLI или читать учетные данные из файла `settings.xml`.

Сборка занимает несколько минут, и в результате получается бездистрибутивный образ контейнера, который отправляется непосредственно в реестр, в данном случае Docker Hub:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< com.redhat.cloudnative:catalog >-------------------
[INFO] Building CoolStore Catalog Service 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ catalog ---
[INFO] Copying 4 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.6.1:compile (default-compile) @ catalog ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- jib-maven-plugin:2.8.0:build (default-cli) @ catalog ---
[INFO]
[INFO] Containerizing application to modernizingjavaappsbook/catalog-spring-boot
  ...
[WARNING] Base image 'gcr.io/distroless/java:11' does not use a specific image
  digest - build may not be reproducible
[INFO] Using credentials from <to><auth> for modernizingjavaappsbook/
  catalog-spring-boot
```

```
[INFO] Using base image with digest:
  sha256:65aa73135827584754f1f1949c59c3e49f1fed6c35a918fadba8b4638ebc9c5d
[INFO]
[INFO] Container entrypoint set to [java, -cp, /app/resources:/app/classes:/app/
  libs/*, com.redhat.cloudnative.catalog.CatalogApplication]
[INFO]
[INFO] Built and pushed image as modernizingjavaappsbook/catalog-spring-boot
[INFO] Executing tasks:
[INFO] [==============================] 100,0% complete
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  27.817 s
[INFO] Finished at: 2021-03-19T11:48:16+01:00
[INFO] ------------------------------------------------------------------------
```

Образ контейнера не сохраняется в локальном кэше, поскольку для создания образов с помощью Jib не нужна среда выполнения контейнеров. Вы не увидите его, выполнив команду docker images, но сможете загрузить его из Docker Hub, и тогда он сохранится в вашем кэше. Если вам нужно, чтобы образ сохранялся локально с самого начала, то можно настроить Jib, чтобы он подключался к хостам Docker и загружал образы в локальный кэш.

JKube

Eclipse JKube (https://oreil.ly/Km2ci), проект сообщества, поддерживаемый Eclipse Foundation и Red Hat, — это еще и фреймворк Java с открытым исходным кодом, помогающий взаимодействовать с Kubernetes. Он поддерживает создание образов контейнеров с использованием Docker/Podman, Jib и Source-to-Image (S2I; сборка из исходного кода в образ). Кроме того, Eclipse JKube предоставляет набор инструментов, позволяющих выполнять автоматическое развертывание в Kubernetes и управлять приложением с помощью вспомогательных средств для отладки и журналирования. Eclipse JKube создан на основе плагина Fabric8 Maven, переименованного и усовершенствованного в целях поддержки Kubernetes.

JKube поддерживает Kubernetes и OpenShift. OpenShift предлагает механизм Source-to-Image (https://oreil.ly/4z2Zn), действующий поверх Kubernetes, предназначенный для автоматической компиляции образов контейнеров из исходного кода. Сама сборка выполняется в Kubernetes, поэтому разработчики могут тестировать и развертывать свои приложения непосредственно на целевой платформе.

Как и в случае с Jib, фреймворк JKube поддерживает режим Zero Configuration (без конфигурации) для быстрого запуска, в котором автоматически выбираются настройки по умолчанию. JKube предоставляет встроенную конфигурацию внутри конфигурации плагина в формате XML. Кроме того, он предоставляет паттерны внешней конфигурации реальных дескрипторов развертывания, которые дополняются плагином.

JKube предлагается в трех формах.

- *Плагин Kubernetes* — работает в любом кластере Kubernetes, обеспечивая бездистрибутивную сборку или сборку с использованием Dockerfile.
- *Плагин OpenShift* — работает в любом кластере Kubernetes или OpenShift, обеспечивая бездистрибутивную сборку, сборку с использованием Dockerfile или сборку Source-to-Image (S2I).
- *Набор инструментов JKube* — набор инструментов и интерфейс командной строки для взаимодействия с JKube Core. Действует еще и как клиент Kubernetes и предоставляет Enricher API для расширения манифестов Kubernetes.

JKube предлагает больше возможностей, чем Jib; на самом деле JKube можно считать надмножеством Jib. С его помощью можно создавать бездистрибутивные сборки Jib, а также работать с Dockerfile и развертывать манифесты Kubernetes из Java. В этом случае нет необходимости определять свои объекты Deployment и Service; JKube сам позаботится о создании контейнера и его развертывании в Kubernetes.

Включим JKube в файл POM сервиса Catalog и настроим сборку Jib с развертыванием в Kubernetes. Исходный код примера можно найти в репозитории книги на GitHub (https://oreil.ly/Ba4Ro).

Сначала добавим JKube как плагин:

```
<project>
  ...
  <build>
    <plugins>
      ...
    <plugin>
       <groupId>org.eclipse.jkube</groupId>
       <artifactId>kubernetes-maven-plugin</artifactId>
       <version>1.1.1</version>
    </plugin>
      ...
    </plugins>
  </build>
  ...
</project>
```

Далее настроим сборку образа контейнера. В данном случае создать образ и отправить его в Docker Hub можно с помощью Jib. А затем развернем его в Kubernetes:

```
...
<properties>
...
    <jkube.build.strategy>jib</jkube.build.strategy>
    <jkube.generator.name>docker.io/modernizingjavaappsbook/catalog-spring-boot:
      ${project.version}</jkube.generator.name>
</properties>
...
```

Соберем образ:

```
mvn k8s:build
```

Вы должны увидеть примерно такой вывод:

```
JIB>... modernizingjavaappsbook/catalog-spring-boot/1.0-SNAPSHOT/build/
  deployments/catalog-1.0-SNAPSHOT.jar
JIB>    :
```

```
JIB>... modernizingjavaappsbook/catalog-spring-boot/1.0-SNAPSHOT/build/Dockerfile
...
JIB> [=======================      ] 80,0% complete > building image to tar file
JIB> Building image to tar file...
JIB> [=======================      ] 80,0% complete > writing to tar file
JIB> [==============================] 100,0% complete
[INFO] k8s: ... modernizingjavaappsbook/catalog-spring-boot/
  1.0-SNAPSHOT/tmp/docker-build.tar successfully built
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  36.229 s
[INFO] Finished at: 2021-03-19T13:03:19+01:00
[INFO] ------------------------------------------------------------------------
```

JKube создал локальный образ контейнера с помощью Jib и теперь готов отправить его в Docker Hub. Вы можете указать учетные данные одним из трех способов.

- *С помощью Docker* — вы можете выполнить вход в свой реестр, в данном случае в Docker Hub, и JKube извлечет учетные данные из файла `~/.docker/config.json`.
- *Задав учетные данные внутри POM* — учетные данные для подключения к реестру можно включить в конфигурацию XML.
- *Задав учетные данные в настройках Maven* — учетные данные можно указать в файле `~/.m2/settings.xml`, и плагин прочитает их оттуда.

В нашем примере мы используем третий вариант и определим учетные данные в настройках Maven; соответственно, вы можете скопировать этот файл к себе и подставить в него свои учетные данные. Исходный код примера можно найти в репозитории книги на GitHub (https://oreil.ly/uxAxW):

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
          http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <servers>
    <server>
      <id>https://index.docker.io/v1</id>
```

```
      <username>USERNAME</username>
      <password>PASSWORD</password>
    </server>
  </servers>
</settings>
```

Чтобы отправить образ в Docker Hub, достаточно просто запустить эту цель Maven:

```
mvn k8s:push
```

Вы должны увидеть примерно такой вывод:

```
JIB> [=========================] 81,8% complete > scheduling pushing manifests
JIB> [=========================] 81,8% complete > launching manifest pushers
JIB> [=========================] 81,8% complete > pushing manifest for latest
JIB> Pushing manifest for latest...
JIB> [=========================] 90,9% complete > building images to registry
JIB> [=========================] 90,9% complete > launching manifest list pushers
JIB> [=========================] 100,0% complete
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:08 min
[INFO] Finished at: 2021-03-19T13:21:28+01:00
```

Теперь можно развернуть сервис Catalog в Kubernetes. Позвольте JKube подключиться к вашему кластеру Kubernetes, прочитав файл `~/.kube/config` на вашей рабочей станции:

```
mvn k8s:resource k8s:apply
```

Вы должны увидеть примерно такой вывод:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< com.redhat.cloudnative:catalog >-------------------
[INFO] Building CoolStore Catalog Service 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- kubernetes-maven-plugin:1.1.1:resource (default-cli) @ catalog ---
[INFO] k8s: Running generator spring-boot
  ...
```

```
[INFO] k8s: Creating a Service from kubernetes.yml namespace default name catalog
[INFO] k8s: Created Service: target/jkube/applyJson/default/service-catalog.json
[INFO] k8s: Creating a Deployment from kubernetes.yml namespace default name
  catalog
[INFO] k8s: Created Deployment: target/jkube/applyJson/default/deployment-
  catalog.json
[INFO] k8s: HINT: Use the command `kubectl get pods -w` to watch your pods start
  up
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  7.464 s
[INFO] Finished at: 2021-03-19T13:38:27+01:00
[INFO] ------------------------------------------------------------------------
```

Как видите, приложение было успешно развернуто в Kubernetes с помощью сгенерированных манифестов:

```
kubectl get pods

NAME                         READY   STATUS    RESTARTS   AGE
catalog-64869588f6-fpjj8     1/1     Running   0          2m2s

kubectl get deploy

NAME      READY   UP-TO-DATE   AVAILABLE   AGE
catalog   1/1     1            1           3m54s
```

Чтобы убедиться, что все действительно сделано как надо, посмотрим список сервисов:

```
kubectl get svc

NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
catalog      ClusterIP   10.99.26.127   <none>        8080/TCP   4m44s
```

По умолчанию Kubernetes предоставляет доступ к приложению только изнутри кластера, определяя тип объекта Service как ClusterIP. Вы можете открыть к нему доступ извне, указав этот тип как NodePort или определив объект Ingress. В этом примере мы используем kubectl port-forward для отображения открытого порта Kubernetes в порт нашей рабочей станции.

Откроем доступ к нашему приложению с помощью команды `kubectl port-forward`:

```
kubectl port-forward deployment/catalog 8080:8080
```

Если вы сейчас откроете браузер и введете адрес http://localhost:8080/api/catalog, то увидите данные в формате JSON, возвращаемые сервисом Catalog приложения Coolstore.

Резюме

В этой главе мы поговорили о том, как с помощью возможностей Kubernetes разработчики на Java могут модернизировать и улучшать свои приложения. Рассмотрели внутренний цикл разработки с помощью Kubernetes. Узнали, как создавать и развертывать образы контейнеров в Kubernetes. Кроме того, мы прошли этапы создания и развертывания контейнеров непосредственно из Java с помощью Maven, Jib и JKube.

Модернизация важна для разработчиков, поскольку помогает сделать приложения облачными и переносимыми, готовыми предоставлять сервисы с высокой доступностью. В следующей главе мы подробнее рассмотрим модернизацию существующих приложений на Java и необходимые для этого шаги.

ГЛАВА 5

Больше чем простой перенос: работа с наследием

> Наследие — это не то, что я делаю для себя. Это то, что я делаю для следующего поколения.
>
> *Витор Белфорт (Vitor Belfort)*

Многие организации сталкиваются с проблемой поддержания существующих бизнес-приложений в рабочем состоянии, одновременно пытаясь внедрять инновации. При этом, как правило, организации ожидают более быстрого ввода новых возможностей и снижения затрат, что кажется труднодостижимым при взгляде на существующий ландшафт приложений и распространенность устаревших систем.

Мы часто используем термин «унаследованная система» для описания старых методологий, технологий или приложений, разработанных не в соответствии с новейшими методами или применяющих устаревший стек технологий. Многие системы, созданные нами в начале нашей

карьеры, можно смело отнести к этой категории. Мы знаем, что большинство из них все еще используются. Благодаря некоторым из них возникли новые подходы и даже последовавшие за ними стандарты. Обычно мы также подразумеваем, что эти системы нуждаются в замене, вследствие чего в конечном итоге они воспринимаются с негативным оттенком. К счастью, это не всегда так. Слово «наследие» также позволяет успешно описывать достижения. Название «унаследованный» не означает «устаревший» или «непригодный для использования». Существует множество причин для сохранения устаревших систем, в том числе:

- система работает в точности как задумано, и нет необходимости что-то менять;
- внедренные бизнес-процессы позабыты и не документированы, а их замена стоит дорого;
- стоимость замены системы превышает выгоду от сохранения ее неизменной.

Книга Майкла Физерса (Michael Feathers) *Working Effectively with Legacy Code* (O'Reilly; https://oreil.ly/iogGC)[1] описывает методы экономного решения распространенных проблем с унаследованным кодом, позволяющие обходиться без чрезвычайно дорогостоящего переписывания всего существующего кода.

Физерс сказал: «Для меня унаследованный код — это просто код без тестов». В настоящее время термин «унаследованный» в первую очередь относится к монолитным приложениям. В современной корпоративной среде существует множество подходов к работе с унаследованными приложениями, и выбор правильного является первой и наиболее важной частью процесса модернизации.

До сих пор мы говорили только об отдельных системах. И разработчики обычно заботятся только об этой конкретной системной области, тогда

[1] *Физерс М.* Эффективная работа с унаследованным кодом.

как планы модернизации должны следовать всеобъемлющим целям компании и учитывать общекорпоративную IT-стратегию. Особенно интересный подход к миграции в облако представлен в книге Грегора Хоупа (Gregor Hohpe) *Cloud Strategy: A Decision-Based Approach to Successful Cloud Migration* (https://oreil.ly/EuW9J). Мы рекомендуем прочитать ее всем, кому интересно познакомиться с обобщенным опытом миграции.

Управление наследием

Всякое успешное путешествие начинается с первого шага. В отношении миграции приложений таковым будет оценка существующих приложений. Мы предполагаем, что вы знакомы с целями всей компании. Теперь мы можем сопоставить их с оценочными категориями. Еще один источник оценочных категорий — технические требования, например существующие чертежи или рекомендуемые основные решения и версии фреймворков. Создание и обновление этого списка оценочных категорий должно быть повторяющейся задачей, включенной в ваш процесс управления. В конечном счете критерии миграции можно вывести из этих оценочных критериев и использовать в качестве краеугольных камней при принятии решений в ходе модернизации.

Оценка приложений для миграции

При оценке усилий, которых может потребовать миграция или модернизация, важно учитывать конкретные мотивирующие сложности, характерные для вашей организации. Некоторые примеры таких сложностей представлены ниже.

- *Ограниченный бюджет на разработку*. Команды разработчиков вынуждены работать быстрее и эффективнее. Вместо того чтобы работать со сложными спецификациями, они стремятся использовать

облегченные фреймворки и готовые реализации. Модернизация обычно должна планироваться как часть текущего проекта разработки или технического обслуживания.

- *Утрата навыков*. Профессиональные навыки работы с существующими технологиями постепенно утрачиваются. Примерами могут служить низкоуровневое программирование или программирование на ранних версиях спецификаций Enterprise Java, которые больше не изучаются и считаются устаревшими. Изменение существующих систем, основанных на старых технологиях, может означать необходимость освоения определенных навыков, требующихся для разработки проекта.
- *Предполагаемые риски*. Следуя известной пословице, ставшей популярной примерно в 1977 году, «Если что-то работает, то пусть работает — не трогай», мы действительно видим много предполагаемых рисков, связанных с изменением хорошо зарекомендовавшего себя и работающего программного обеспечения. Причин для этого может быть множество: начиная от отсутствия знаний о системе и заканчивая страхом перед остановкой производства на фабриках. Необходимо рассматривать эти риски отдельно и предусматривать в плане миграции конкретные действия по их смягчению.
- *Отсутствие известного предсказуемого процесса*. Наша книга поможет вам в этом конкретном вопросе. Навигация по неизвестному процессу может представлять серьезную проблему. Наличие проверенного и воспроизводимого процесса, который признают и которому следуют все стороны, имеет решающее значение для успеха модернизации.
- *Оценка реальных усилий*. Оценка усилий по миграции не должна быть чем-то магическим. К сожалению, многие компании имеют минимальные представления о реальных усилиях, которых требует модернизация корпоративных Java-приложений. Использование предсказуемого и оптимизированного подхода к их оценке устранит эту проблему.

Превращение этих сложностей в действенные параметры оценки может выглядеть следующим образом:

- прогнозирование уровня усилий и затрат;
- планирование миграции приложений и устранение конфликтов;
- выявление всех потенциальных рисков на уровне кода, инфраструктуры, процессов или знаний;
- прогнозирование окупаемости инвестиций для бизнес-обоснования;
- выявление и снижение рисков для бизнеса;
- минимизация нарушений существующих бизнес-операций.

Если речь идет о единственном приложении, то достаточно все эти пункты свести в одну электронную таблицу или документ. Однако для средне- и крупномасштабных мероприятий требуется более подходящее решение. Крупномасштабные проекты нуждаются в автоматизированных процедурах и правилах, позволяющих оценить базу установки и привязать приложения к бизнес-сервисам в целях надежного планирования следующих шагов. Проект Konveyor (https://www.konveyor.io/) с открытым исходным кодом предлагает простой способ сбора всей необходимой информации и управления ею. Он сочетает в себе набор инструментов, призванных помочь при модернизации и миграции на Kubernetes.

Forklift (часть проекта Konveyor) предоставляет возможность миграции виртуальных машин на KubeVirt, сопровождаемой минимальным временем простоя. Подпроект Crane сосредоточен на переносе приложений между кластерами Kubernetes. Кроме того, в наборе имеется инструмент Move2Kube, помогающий ускорить перевод приложений на основе Swarm и Cloud Foundry на Kubernetes.

В частности, для модернизации приложений Konveyor предлагает проект Tackle (https://oreil.ly/u99Gf). Он оценивает и анализирует приложения с точки зрения пригодности для рефакторинга в контейнеры и проводит стандартную инвентаризацию.

Tackle Application Inventory

Этот инструмент позволяет пользователям поддерживать свой портфель приложений, связывать их с бизнес-сервисами, которые они поддерживают, и определять взаимозависимости. Application Inventory использует расширяемую модель маркировки для добавления метаданных, что является отличным способом привязки категорий миграции, как обсуждалось ранее. Application Inventory используется для выбора приложения, которое затем должно оцениваться с помощью Pathfinder.

Tackle Pathfinder

Это интерактивный инструмент, оценивающий пригодность приложений для модернизации и развертывания в контейнерах на корпоративной платформе Kubernetes. Pathfinder (https://oreil.ly/K4V4u) генерирует отчет о пригодности приложения для Kubernetes, включая связанные риски, и создает план внедрения. В своих рекомендациях Pathfinder опирается на информацию из реестра приложений и ответов на дополнительные оценочные вопросы. Если приложение зависит от прямого подключения к хост-системе, то Pathfinder может признать его непригодным для миграции в Kubernetes, поскольку это приведет к перегрузке частей хоста. Вот примеры оценочных вопросов.

- Поддерживаются ли компоненты сторонних поставщиков в контейнерах?
- Продолжается ли активное развитие приложения?
- Есть ли у приложения какие-либо юридические требования (например, связанные с обработкой платежной или медицинской информации)?
- Предоставляет ли приложение метрики?

Мы настоятельно рекомендуем использовать Pathfinder для управления крупномасштабной модернизацией целых ландшафтов. Он поможет вам классифицировать приложения и расставить приоритеты сегодня и затем постоянно следить за оценкой необходимости миграции в будущем.

Tackle Controls

Tackle Controls — набор сущностей, добавляющих дополнительные баллы к оценкам Application Inventory и Pathfinder, учитывающие ценность бизнес-сервисов, мнение заинтересованных лиц и групп, должностные функции, типы меток и метки. Кроме того, есть возможность зафиксировать атрибуты, характерные для компании или проекта, внедрив собственные сущности, и отфильтровать портфель, составленный Application Inventory, например, чтобы выбрать все приложения, используемые определенной «должностной функцией», такой как отдел кадров.

Tackle DiVA

Наконец, DiVA (https://oreil.ly/UGzn2) — это инструмент анализа приложений, ориентированный на данные. Как преемник проекта Windup (https://oreil.ly/sjiNq), это самый интересный проект, на который обязательно должны обратить внимание все, кто хочет оценить отдельные приложения. Он ориентирован на оценку традиционных монолитных приложений и в настоящее время поддерживает приложения на основе сервлетов и Spring Boot. DiVA позволяет импортировать набор исходных файлов приложения (Java/XML), а затем предоставляет:

- опись сервисов (экспортируемый API);
- опись баз данных;
- опись транзакций;
- зависимости между кодом и базой данных (графы вызовов);
- зависимости между базами данных;
- зависимости между транзакциями;
- рекомендации по рефакторингу транзакций.

В настоящее время DiVA продолжает активно разрабатываться, и пока реализованы не все возможности оригинального проекта Windup. Тем не менее проект DiVA достаточно сильно облегчает модернизацию.

Кроме того, он дает разработчикам отличную возможность поделиться своими открытиями и стать частью большого сообщества, занимающегося проблемами автоматизации миграции.

Migration Toolkit for Applications

Пока не завершилась интеграция Windup в DiVA, можно использовать автоматизированную оценку миграции для приложений Enterprise Java с помощью набора инструментов миграции для приложений (Migration Toolkit for Applications, MTA (https://oreil.ly/SIOSR)).

MTA объединяет инструменты поддержки крупномасштабных проектов модернизации и миграции корпоративных Java-приложений. Он позволяет импортировать двоичные файлы или архивы приложения и автоматически выполняет анализ кода, включая портфель и зависимости приложений, определяет возможные проблемы миграции и дает оценку усилий по миграции. Первоначально этот набор разрабатывался для поддержки миграции серверов Java EE (например, с WebSphere или WebLogic на JBoss EAP). Однако он имеет легко расширяемый механизм, позволяющий добавлять свои наборы правил и даже адаптировать существующие к своим потребностям. В настоящее время MTA также поддерживает миграцию Spring Boot в Quarkus.

Например, вот так выглядит выдержка из правила на Java:

```
//...
JavaClass.references("weblogic.servlet.annotation.WLServlet")
    .at(TypeReferenceLocation.ANNOTATION)
        )
        .perform(
            Classification.as("WebLogic @WLServlet")
               .with(Link.to("Java EE 6 @WebServlet",
                             "https://some.url/index.html"))
               .withEffort(0)
               .and(Hint.withText("Migrate to Java EE 6 @WebServlet.")
               .withEffort(8))
        );
//...
```

Это правило отыскивает в классах Java аннотации `@WLServlet` и увеличивает оценку усилий, необходимых для миграции. Узнать больше о правилах и об их разработке можно в документации Windup (https://oreil.ly/FbQKL).

Кроме того, оно поддерживает варианты использования без миграции в рамках процесса сборки (через подключаемый плагин Maven (https://oreil.ly/T8mom) или интерфейс командной строки (https://oreil.ly/U7Dsk)) либо в ходе регулярной проверки кода на соответствие стандартам организации или его переносимости.

Вот некоторые из паттернов, которые может обнаруживать MTA:

- собственные библиотеки;
- собственные конфигурации;
- механизмы поиска сервисов;
- веб-сервисы;
- EJB-дескрипторы;
- унаследованный код Java;
- диспетчеры транзакций;
- внедренные фреймворки;
- механизмы объединения потоков выполнения в пулы;
- сервисы таймеров;
- дескрипторы WAR/EAR;
- статические IP-адреса.

MTA и DiVA — два эффективных инструмента, позволяющих определить общий технический долг, что приводит к классификации потребностей и рисков миграции. Однако они не могут подсказать, какую функциональность следует перенести или модернизировать в первую очередь. Для этого вы должны более основательно изучить дизайн и функциональность приложения.

Оценка функциональности для миграции

Традиционные монолиты могут иметь разные формы и размеры. Когда используется термин «монолит», под ним обычно подразумевается артефакт развертывания. В Enterprise Java это традиционно корпоративные архивы (Enterprise Archives, EAR) или веб-архивы (Web Archives, WAR). Их также можно рассматривать как приложения, выполняющиеся в единственном процессе. Они могут быть разработаны в соответствии с рекомендациями по модульной организации, такими как OSGi (Open Services Gateway Initiative), или техническими подходами, такими как трехуровневая архитектура без значимых бизнес-модулей. Общее направление усилий по модернизации сильно зависит от типа монолита. Как правило, чем больше модулей в существующем приложении, тем проще оно поддается модернизации. В идеальном мире модули можно напрямую преобразовать в сервисы. Но в реальной жизни это случается редко.

Если монолит имеет вид гигантского блока кода, то к нему следует применить логическую модель. Вероятнее всего, внутренне этот блок организован вокруг деловых и технических функций, таких как управление заказами, отображение PDF, рассылка уведомлений клиентам и т. д. Но даже если код не организован вокруг этих понятий, они все равно существуют в кодовой базе. Эти предметные области, называемые в предметно-ориентированном проектировании (Domain-Driven-Design, DDD) ограниченными контекстами, становятся новыми сервисами.

Более подробную информацию можно узнать в книге Эрика Эванса (Eric Evans) Domain-Driven Design: Tackling Complexity in the Heart of Software (O'Reilly; https://oreil.ly/kgLPl)[1], считающейся стандартным введением в DDD.

[1] *Эванс Э.* Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем.

Когда границы модулей и функциональные возможности будут определены, можно начинать думать о модернизации. Но прежде обязательно оцените компромиссы между затратами на модернизацию каждого модуля и преимуществами, которые появятся в результате, и начинайте с лучшего кандидата. На рис. 5.1 показано, как может выглядеть такая оценка на примере приложения с шестью модулями. Предположим, что мы говорим о вымышленном интернет-магазине. Модули с сильными взаимозависимостями, такие как Order и Customer, будет сложно извлечь по отдельности. Если дополнительно учесть необходимость масштабируемости, то извлечение их из монолита может быть не очень выгодным. Эти два модуля находятся внизу слева на графике. В противоположном углу координатной плоскости находится сервис Catalog (на рис. 5.1 обозначен зеленым блоком). Он возвращает список доступных товаров, имеет немного зависимостей и позволяет только читать данные. В периоды увеличения посещаемости сайта этот модуль запрашивается чаще других, поэтому его отделение будет наиболее выгодным. Выполните аналогичный анализ компромиссов для всех модулей вашего приложения и сравните затраты и выгоды.

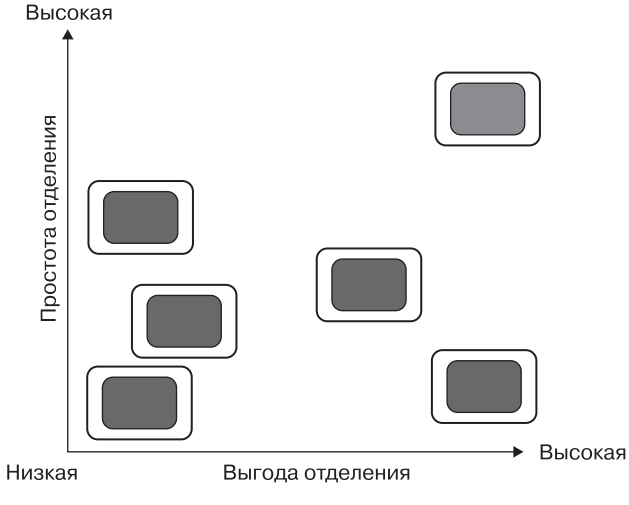


Рис. 5.1. Сравнение затрат и выгод

Теперь вы достигли последней контрольной точки, в которой должны подтвердить свою предыдущую стратегическую оценку приложения. Превышают ли ожидаемые выгоды модернизации предполагаемые затраты? К сожалению, здесь нет общеприменимой рекомендации, поскольку многое зависит от самого приложения, бизнес-требований, а также общих целей и задач компании. Документируйте свои решения и выводы, поскольку сейчас самое время определиться с будущим направлением ваших усилий по модернизации. Помните концепцию 6R, о которой рассказывалось в главе 3? Retain (сохранение), Retire (вывод из эксплуатации), Repurchase (повторная покупка новой версии), Rehost (перенос в контейнеры), Replatform (смена платформы, небольшие корректировки) и Refactor (рефакторинг, добавление чего-то нового).

Определив функциональные возможности и оценив затраты на миграцию, вы будете знать, какие аспекты приложения можно начинать модернизировать. Допустим, вы решили не создавать новое приложение, а аккуратно модернизировать существующее. В следующем разделе мы подробнее рассмотрим некоторые подходы к миграции в этом случае.

Подходы к миграции

Вышеупомянутые инструменты и оценки помогли вам определить наиболее подходящие приложения и сервисы, и теперь пришло время углубиться в стратегии и сложности миграции единственного приложения.

Защита наследия (Replatform)

Когда в обновлении или расширении нуждаются только один или два модуля, самый простой способ — сосредоточиться на двух модулях и сохранить остальную часть приложения как есть, сделав ее работоспо-

собной в современной инфраструктуре. Для этого, помимо изменений в соответствующих модулях, необходимо также оценить среду выполнения, библиотеки или даже целевую инфраструктуру с точки зрения минимизации изменений в коде.

Этого можно достичь, просто поместив приложение и базы данных в контейнеры и изменив соответствующие модули монолита или полностью отделив определенные функции и повторно интегрировав их в частично распределенную систему, как показано на рис. 5.2.

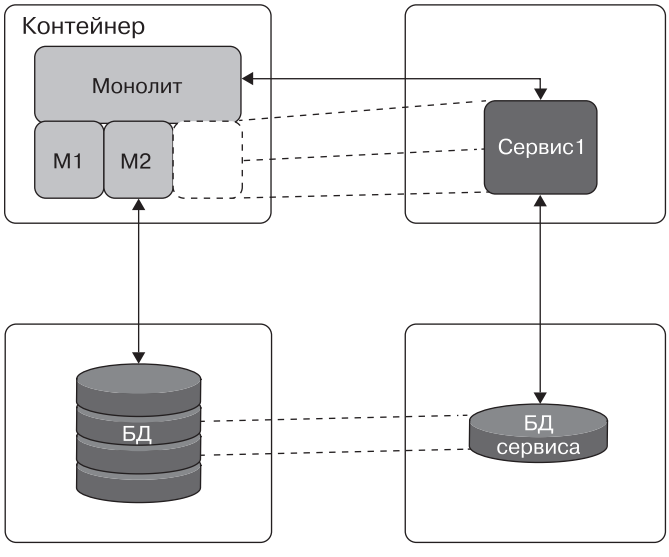


Рис. 5.2. Вновь объединяем все части

Легко сказать, да сложно сделать. Существует множество нефункциональных требований, которые необходимо перенести с платформы сервера приложений на внешнюю архитектуру (https://oreil.ly/rrcZG). В следующей главе мы сосредоточимся на наиболее важных элементах, а здесь поговорим только о миграции самого приложения и базы данных.

Связь сервиса с приложением

Когда определенная функциональность будет извлечена, самый насущный вопрос будет заключаться в интеграции оставшегося монолита с вновь извлеченным сервисом. Если предположить, что вы переходите на использование среды выполнения контейнеров, то вам следует использовать шлюз API для балансировки нагрузки и переключения трафика на основе URL. Подробнее об этом рассказывается в главе 6.

Другой подход заключается в использовании прокси-сервера HTTP. Важно, чтобы прокси был запущен в работу до извлечения компонентов из монолита. Убедитесь, что он не нарушает работу существующего монолита. Кроме того, может потребоваться некоторое время, чтобы регулярно запускать новый сервис в промышленной среде, даже если он не используется конечными пользователями. Переключайтесь постепенно, перенаправляя трафик, и следите за тем, нет ли проблем.

Чтобы упростить взаимодействия сервиса с монолитом, можете прибегнуть к реализации простой прямой связи JAX-RS. Однако этот подход применим, только когда имеется очень небольшое количество сервисов. Обязательно рассматривайте извлеченный сервис как систему интеграции с точки зрения монолита.

Все три варианта (шлюз API, HTTP-прокси и интерфейс JAX-RS) позволяют создать ваш первый успешный микросервис. Все они реализуют паттерн Strangler («Душитель») (см. главу 3) и помогают реорганизовать монолит в отдельные системы, выступая в качестве первого шага.

Перехват — потенциально опасный путь: если начать создавать собственный уровень трансляции протокола, который используется несколькими сервисами, возникает риск придать слишком много интеллектуальных возможностей общему прокси. Этот подход уводит от независимых микросервисов и приводит к сервис-ориентированной архитектуре

со слишком сложной логикой на уровне маршрутизации. Лучшей альтернативой является так называемый паттерн Sidecar («Прицеп»), суть которого состоит в создании дополнительного контейнера в поде. В таком случае дополнительная логика становится частью нового сервиса. Оформленная как дополнительный компонент Kubernetes, эта логика становится частью среды выполнения и может обслуживать как старых, так и новых клиентов.

Sidecar («Прицеп») — это просто контейнер, работающий в том же поде, что и контейнер приложения. Он использует те же том и сеть и может помочь расширить возможности приложения. Типичными примерами могут служить журналирование или другие функциональные возможности агента.

Связь базы данных с другими базами данных

Определив функциональную границу и метод интеграции, нужно решить, как осуществить разделение базы данных. Монолитные приложения обычно полагаются на одну большую БД, однако каждый извлеченный сервис должен работать с собственными данными. Правильный способ решения этой головоломки снова зависит от существующей структуры данных и транзакций.

Относительно простой первый шаг выглядит так: разделить таблицы, используемые сервисом, на представления, доступные только для чтения, и таблицы для записи и настроить монолит на использование интерфейса с операциями чтения и записи. Впоследствии такие интерфейсы будет проще абстрагировать в обращения к сервисам. Этот подход требует внесения изменений только в монолит и должен оказывать минимальное влияние на существующий код. Таблицу можно переместить в отдельную базу данных и на следующем шаге настроить соответствующие запросы.

Все эти подготовительные изменения вносятся исключительно в старый монолит. Преобразование существующего кода в модульную структуру может быть рискованным. В частности, риск увеличивается по мере усложнения модели данных. На последнем шаге можно переместить извлеченные таблицы в новую базу данных и настроить монолит на использование вновь созданного сервиса для взаимодействия с бизнес-объектом. Эта задача относительно просто решается на бумаге, но быстро достигает предела практичности, если для доступа к данным требуется много соединений между таблицами. Простыми кандидатами являются объекты основных данных, например «Пользователь». Более сложные кандидаты могут быть комбинированными объектами, такими как «Заказ». Все, что говорилось о модульности кода приложения, в еще большей степени относится к базе данных. Чем лучше она организована, тем проще будет выделить функциональность и данные в отдельный сервис. Иногда можно обнаружить, что нет хорошего решения для извлечения объектов из модели данных или что другие подходы не обеспечивают подходящей производительности. В таких случаях следует пересмотреть выбранный вами путь модернизации.

Если все пройдет благополучно, вы получите две отдельные базы данных и два очень неравных «сервиса», составляющих систему. После этого можно подумать о стратегиях синхронизации данных между сервисами. Большинство БД предлагают возможность запустить некоторую логику при изменении данных. Это могут быть простые триггеры, копирующие изменившиеся записи в другие таблицы, или обработчики, вызывающие функции более высокого уровня (например, веб-сервисы). Но часто эта функциональность сильно зависит от используемой базы данных. Задействовать эти возможности допустимо, если внутри компании издан официальный документ, регламентирующий использование определенных функций, или если вы вполне уверены в дальнейшем изменении исходной БД.

В противном случае можно прибегнуть к пакетной синхронизации на основе заданий. Необходимость репликации можно определять по изменениям отметок времени, номеров версий или значений в столбцах

состояния. Это достаточно устоявшийся и хорошо известный подход к синхронизации данных, который можно найти во многих унаследованных системах. Основным его недостатком является неизбежное расхождение в точности данных в целевой системе, независимо от реализации. Кроме того, более короткие интервалы репликации могут привести к дополнительным затратам на транзакции или увеличить нагрузку на исходную систему. Этот подход пригоден, только когда данные обновляются редко, а промежуточный этап обработки между обновлениями в идеале не должен зависеть от времени. Но он не подходит, когда обновления должны копироваться в режиме реального времени или по крайней мере достаточно быстро.

Современный подход к решению проблемы синхронизации данных основан на средствах чтения журналов. Реализованные в форме сторонних библиотек, они идентифицируют изменения, сканируя файлы журналов транзакций базы данных. Эти файлы предназначены для операций резервного копирования и восстановления и гарантированно фиксируют все изменения данных, включая удаление. Эта концепция также известна как захват изменений данных. Одним из самых заметных проектов в этой области является Debezium (https://debezium.io/). Использование средств чтения журналов — наименее разрушительный вариант синхронизации изменений между базами данных, поскольку они не требуют модификации исходной БД и не нагружают исходные системы дополнительными запросами. События изменения данных генерируют уведомления для других систем с помощью паттерна Outbox («Исходящий ящик»).

Создание чего-то нового (рефакторинг)

Если по каким-то причинам вы пришли к решению заново реализовать и реорганизовать всю свою систему и перенести ее на новую распределенную архитектуру, то, скорее всего, вы думаете о синергии и способах разделения усилий на небольшие и управляемые задачи. Учитывая сложность полного стека микросервисов, это очень непросто. Один из

важнейших факторов при таком подходе — знания, которыми обладает команда. После многих лет разработки на сервере приложений Enterprise Java команде должны пригодиться накопленные знания API и стандартов. Существуют различные способы реализации сервисов на JVM, помогающие командам повторно использовать наиболее важные функции, которые им уже известны из стандартов Enterprise Java/Jakarta EE. Обсудим некоторые из этих методов реализации сервисов на JVM.

Jakarta EE (https://jakarta.ee/about) — это набор спецификаций, позволяющий разработчикам Java работать с приложениями Java Enterprise. Спецификации разработаны известными в отрасли лидерами, которые пользуются доверием у разработчиков и потребителей технологий. Это версия Java Enterprise Edition с открытым исходным кодом.

MicroProfile

Проект MicroProfile (https://microprofile.io/) создан в 2016 году и быстро присоединился к фонду Eclipse. Основная цель MicroProfile — создать фреймворк Java Enterprise, предназначенный для реализации переносимых микросервисов независимым от поставщика способом. MicroProfile состоит из модели программирования, конфигурации и сервисов, которые не зависят от поставщика и обеспечивают трассировку, отказоустойчивость, работоспособность и метрики, характеризующие работу приложения. Компоненты MicroProfile API основаны по модели Jakarta EE, что делает переход к микросервисам более естественным для Java-разработчиков за счет возможности использовать уже имеющиеся знания о Jakarta EE. MicroProfile определяет 12 спецификаций, как показано на рис. 5.3, а лежащая в их основе модель компонентов использует подмножество существующих стандартов Jakarta EE. По сравнению с полной спецификацией Jakarta EE в MicroProfile отсутствуют особенно тяжелые спецификации. Больше всего для больших монолитных приложений подходят Enterprise JavaBeans (EJB) и Jakarta XML Web Services.

Рис. 5.3. Обзор технологий MicroProfile

Существуют различные реализации спецификаций MicroProfile: Open Liberty, Thorntail, Paraya Server, TomEE, SmallRye и т. д. Поскольку MicroProfile основан на принципах и компонентах, близких к Jakarta EE Web Profile, перенос существующих приложений происходит сравнительно просто.

Quarkus

Quarkus (http://quarkus.io/) — относительно новый член клуба микросервисных фреймворков. Это полный стек, Java-фреймворк для Kubernetes и JVM. Он оптимизирован для работы в контейнерах и ограниченных средах выполнения. Его основная цель — служить идеальной средой выполнения для бессерверных и облачных сред, а также сред Kubernetes.

Он поддерживает популярные стандарты, фреймворки и библиотеки Java, такие как Eclipse MicroProfile, Spring Boot, Apache Kafka, RESTEasy (JAX-RS), Hibernate ORM (JPA), Infinispan, Camel и многие другие.

Поддержка внедрения зависимостей основана на решении CDI (Contexts and Dependency Injection — внедрение контекстов и зависимостей), заимствованном из Jakarta EE, что делает его совместимым с распространенными моделями компонентов. Интересной частью является фреймворк расширений, помогающий добавлять новые функциональные возможности, предназначенные для настройки, загрузки и интеграции библиотек, используемых в компании. Он работает на JVM и поддерживает GraalVM (универсальную виртуальную машину для многих языков).

Модели компонентов для сервисов

Один из наиболее типичных вопросов, которые задают разработчики, звучит следующим образом: как перенести существующие модели компонентов приложений Enterprise Java в микросервисы? Обычно этот вопрос относится к компонентам Enterprise Java Beans или CDI Beans,

особенно к хранимым компонентам bean, управляемым контейнером (до EJB3), которые необходимо воссоздавать с помощью Java Persistence API (JPA). Мы настоятельно рекомендуем проверить точность базового отображения данных/объектов, его соответствие новым требованиям и затем полностью воссоздать его. Это не самая трудоемкая и затратная часть модернизации. Намного сложнее реализовать бизнес-требования. Компоненты CDI технически совместимы с MicroProfile, но решение о целесообразности простой миграции кода зависит от новых бизнес-требований. Важно отыскать существующие границы транзакций, чтобы убедиться в том, что нет необходимости задействовать нижестоящие ресурсы. В общем случае рекомендуется повторно использовать как можно меньше исходного кода. Причина здесь в основном в разнице подходов к проектированию систем между двумя технологиями. В свое время нам сошел с рук наполовину модульный монолит, однако с микросервисами это не пройдет. Особое внимание к определению ограниченных контекстов окупится производительностью и качеством дизайна окончательного решения.

Преобразование Spring-приложений в сервисы

Аналогичный подход можно применить к приложениям, основанным и на других фреймворках, таких как Spring. Технически вы легко сможете обновлять и копировать существующие реализации, но недостатки никуда не исчезают. В частности, командам разработчиков, использующим Spring, может быть полезно использовать API, совместимые с другими фреймворками, такими как Quarkus.

В число Spring API, совместимых с Quarkus, входят Spring DI, Spring Web и Spring Data JPA. Частично совместимы Spring Security, Spring Cache, Spring Scheduled и Spring Cloud Config. Цель реализации поддержки Spring API в Quarkus состояла не в том, чтобы обеспечить полную совместимость, позволяющую переносить существующие приложения Spring, а в том, чтобы обеспечить совместимость Spring API, достаточную для разработки новых приложений с помощью Quarkus.

Проблемы

Тщательно оценивая и планируя каждый шаг, можно реструктурировать и модернизировать существующие монолитные приложения. В большинстве случаев этот процесс требует приличного объема ручной работы, и есть ряд типичных проблем, на которые следует обратить внимание.

Избегайте двойной записи

Создав несколько микросервисов, вы быстро поймете, что самое сложное — это данные. Выполняя свою бизнес-логику, микросервисы часто должны обновлять свои данные в локальном хранилище. При этом им также необходимо оповещать другие сервисы об изменениях. Эта проблема не столь очевидна в мире монолитных приложений или устаревших распределенных транзакций, работающих с одной моделью данных. Решить эту проблему непросто. Переход на распределенную архитектуру, скорее всего, приведет к потере согласованности, как определено в теореме CAP.

Теорема CAP (https://oreil.ly/TVwYw), она же концепция «два из трех», утверждает, что одновременно можно гарантировать только две из следующих трех характеристик: согласованность (consistency), доступность (availability) и устойчивость к разделению (partitition tolerance).

Современные распределенные приложения используют шину событий, например Apache Kafka, для передачи данных между сервисами. Миграция транзакций с двухфазной фиксацией (2PC — two-phase commit) в монолите в распределенный мир значительно меняет поведение приложения и его реакцию на сбои, и возникает острая необходимость в управлении продолжительными и распределенными транзакциями.

Продолжительные транзакции

Паттерн Saga («Сага») предлагает решение проблем двойной записи и продолжительных транзакций. Паттерн Outbox («Исходящий ящик») решает более простую проблему организации взаимодействий между сервисами, но его недостаточно для решения более сложных проблем, связанных с продолжительными транзакциями. Последние обусловлены необходимостью выполнения нескольких операций в разных сервисах с семантикой согласования «все или ничего». Примером может служить любой многоэтапный бизнес-процесс, выполняемый несколькими сервисами. Приложение, представляющее корзину покупателя, должно генерировать электронные письма с подтверждением и печатать транспортную этикетку. Эти действия либо должны выполняться все вместе, либо не должно выполняться ни одно из них. В мире унаследованных монолитов эта проблема практически отсутствует, поскольку координация между модулями осуществляется в рамках одного процесса и одного транзакционного контекста. Распределенный мир требует другого подхода.

Паттерн Saga («Сага») предлагает решать эту проблему путем разделения общей бизнес-транзакции на несколько транзакций, которые выполняются в локальных базах данных сервисов, участвующих в транзакциях. Как правило, существует два способа реализации распределенных саг:

- хореография — в этом подходе каждый сервис отправляет сообщение следующему после завершения своей локальной транзакции;
- оркестрация — в этом подходе предполагается наличие одного центрального сервиса, который координирует выполняемые действия и вызывает участвующие сервисы. Связь между участвующими сервисами может быть синхронной, осуществляемой через HTTP или gRPC, или асинхронной, осуществляемой через механизм обмена сообщениями, такой как Apache Kafka.

Слишком быстрое удаление старого кода

Выделив сервис, вы можете почувствовать труднопреодолимое желание избавиться от старого исходного кода, затрат на его обслуживание и дублирование. Но будьте осторожны. Старый код можно рассматривать как эталон и тестировать изменения в поведении в обеих базах кода. Кроме того, может быть полезно периодически проверять время работы только что созданного сервиса. Обычно рекомендуется в течение некоторого времени эксплуатировать старый и новый варианты параллельно и сравнивать результаты, после чего можно удалить старую реализацию. Однако не стоит делать этого слишком рано.

Особенности интеграции

Традиционные монолиты тесно связаны со сложной интеграционной логикой. Часто она прячется за логикой взаимодействий с пользователем или синхронизации данных. Каждая отдельная система, интегрированная в общий бизнес-процесс, должна рассматриваться как отдельный сервис. Извлекая части данных из существующей модели, можно применять те же принципы и делать это постепенно, шаг за шагом. Другой подход — рассматривать интеграционную логику как сервис с самого начала. Метод, изначально предназначенный для поддержки микросервисов, называется Camel K (https://oreil.ly/JiOwc). Он основан на известной интеграционной библиотеке Apache Camel и заключает маршруты интеграции в контейнеры или отдельные сервисы. С его помощью можно разделить всю интеграционную логику монолитного приложения на отдельные сервисы.

Резюме

Современные корпоративные системы Java подобны поколениям в семье: развиваются, опираясь на унаследованные системы. Проверенные паттерны, стандартизированные инструменты и ресурсы с открытым

исходным кодом помогут вам создавать долговечные системы, способные расти и изменяться в соответствии с вашими потребностями. По сути, выбираемый подход к миграции напрямую связан с проблемами, которые необходимо решить сегодня и завтра. Какую цель вы преследуете? Какого уровня масштабирования текущей архитектуры вы хотите достичь? Возможно, вам помогут микросервисы, а может быть, что-то иное. Вы должны понимать, к какой цели идете, поскольку иначе будет сложно определить, как реализовать миграцию существующих систем. Понимание конечной цели поможет определить, как разделить систему и как расставить приоритеты в этой работе.

ГЛАВА 6

Создание приложений для Kubernetes

В предыдущей главе мы рассказали, как мигрировать с традиционного корпоративного паттерна Java к подходу, ориентированному на контейнеры. В этой главе вы увидите, какие компоненты необходимы для миграции на микросервисную архитектуру и как Kubernetes может помочь в этом.

Ранее вы также узнали, что подход, основанный на микросервисах, помогает сделать программное обеспечение надежным, переносимым и готовым к масштабированию по требованию. Современные архитектуры проектируются с учетом того, что масштабирование возможно с самого начала, и это предполагает не только новые возможности, но и новые проблемы. Разработчики корпоративных приложений на Java знают, что их код является частью бизнес-логики, и полагаются на фреймворки, обеспечивающие надежность и совместимость с общепризнанными паттернами проектирования программного обеспечения. В настоящее время не редкость, когда одно и то же приложение обслуживает миллионы запросов, выполняясь в общедоступном облаке, порой распределенном географически. Для этого приложение необходимо спроектировать так, чтобы оно соответствовало этой модели: разделять функции, избегать создания единой точки отказа и распределять нагрузку между несколькими частями архитектуры, чтобы исключить перерывы в обслуживании.

Поиск правильного баланса между масштабируемостью и сложностью

В идеальном мире приложения не имеют состояния и могут масштабироваться независимо друг от друга. Они не терпят сбоев, а сетевые соединения всегда обеспечивают надежную передачу данных. Но реальность выглядит иначе. Миграция монолитов на микросервисные архитектуры открывает возможность развертывания в облаке, и мы уже видели некоторые преимущества, которые она дает. Однако при этом возникают некоторые новые проблемы: управление несколькими точками входа в приложение, поддержание согласованности между несколькими микросервисами и управление распределенными базами данных и схемами.

На рис. 6.1 показано, как переход от монолитных приложений к микросервисам приводит к появлению нового подхода, предусматривающего наличие нескольких взаимосвязей или даже нескольких баз данных.

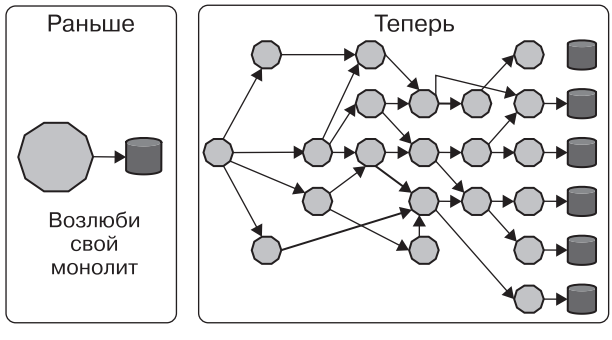


Рис. 6.1. От монолита к микросервисной архитектуре

По аналогии с теоремой CAP (https://oreil.ly/nruM5) очень сложно одновременно обеспечить масштабируемость и простоту системы. Вот почему фреймворк Kubernetes получил такую популярность: он вездесущ, работает в любом облаке и ему можно делегировать бóльшую часть сложности. Он позволяет сосредоточиться «только» на разработке приложений. С другой стороны, нам нужно найти решение и для приложений *с состоянием*, и, как мы увидим далее, Kubernetes способен помочь нам и в этом вопросе.

Функциональные требования к современным архитектурам

Kubernetes очень удобно использовать при определении распределенных приложений, как было показано на рис. 3.1. Любой разработчик на Java должен хорошо знать паттерны проектирования (https://oreil.ly/V4aqC), описанные в книге «Банды четырех»[1], шедевре разработки программного обеспечения, в которой авторы определяют наиболее часто используемые паттерны проектирования ПО. Kubernetes расширяет этот набор паттернов, создавая новый набор специфических облачных требований, помогающих сделать приложения устойчивыми к различным нагрузкам (рис. 6.2). Далее рассмотрим некоторые из них.

Рис. 6.2. Функциональные требования к современным архитектурам

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2022.

Управление через API

Мантра микросервисов — «API прежде всего». Если еще раз взглянуть на рис. 6.1, то можно заметить, что разделение монолитного приложения на набор микросервисов приводит к первой важной проблеме: как обеспечить взаимодействие всех этих компонентов программного обеспечения друг с другом? В монолитах мы полагаемся на области видимости модулей, пакетов. Микросервисы обычно взаимодействуют друг с другом через вызовы REST, в которых микросервис может быть поставщиком или потребителем услуг. Это не единственный способ связывания микросервисов. Не менее распространенный вариант основан на использовании очередей, сообщений или кэшей. Но в любом случае каждый микросервис открывает доступ к своим примитивам или к функции через API, доступ к которому может осуществляться с помощью шлюза API, как обсуждалось в примере с Coolstore в главе 2.

Сам фреймворк Kubernetes представляет собой программное обеспечение, управляемое через API. Управление всеми компонентами платформы, такими как поды, объекты Service и Deployment, осуществляется через REST API. Все операции и обмен данными между компонентами и внешними пользовательскими командами являются вызовами REST API, которые обрабатывает сервер API (https://oreil.ly/PauGF). Взаимодействуя с Kubernetes через `kubectl` или JKube, мы просто вызываем API через HTTPS, отправляя и получая данные в формате JSON. Такая экосистема API — идеальная среда для архитектуры, управляемой через API, например для микросервисов. Теперь, узнав, как взаимодействуют микросервисы, посмотрим, как обнаруживать новые сервисы.

Обнаружение

Позволить микросервисам взаимодействовать друг с другом с помощью вызовов REST несложно. Однако было бы неплохо иметь также удобный способ вызывать другие компоненты и функции, как, например, при импорте модуля или пакета в наше приложение. В современных архитектурах может быть довольно много микросервисов, доступных

для взаимодействий, поэтому иногда недостаточно просто хранить список конечных точек сети, таких как IP-адреса или имена хостов. Как обсуждалось в главе 4, Kubernetes упрощает сетевые взаимодействия, предлагая объект `Service`, позволяя двум или более подам взаимодействовать друг с другом во внутренней сети платформы. Kubernetes также предлагает API, позволяющий перечислять объекты внутри кластера из приложения. Java-разработчики могут использовать такие фреймворки, как JKube, и создавать для этой цели Java-клиенты для Kubernetes.

Список объектов Service и подов, представляющих отдельные микросервисы, дает начало инвентаризации компонентов программного стека в режиме реального времени, дополнительно помогая поддерживать и расширять приложения во время выполнения. Кроме того, Kubernetes поддерживает интеграцию с внешними инструментами и фреймворками, такими как Service Mesh, который реализует протокол обнаружения сервисов для выявления новых сервисов по мере их появления.

Для микросервисных архитектур все чаще выбирают сервисную сетку (service mesh (https://oreil.ly/jGh80)). Она предоставляет панель управления, которая взаимодействует с Kubernetes, и поддерживает обнаружение сервисов, взаимную аутентификацию, A/B-тестирование, маршрутизацию и паттерн размыкателя цепи (circuit breaker). Более подробную информацию можно найти в Интернете (https://oreil.ly/ECIF4).

Безопасность и авторизация

Еще одна проблема, которую должны учитывать современные разработчики приложений, — это безопасность всего стека, от приложения до платформы. Рекомендуемые приемы также применимы к современным архитектурам, но сложность и необходимые усилия могут значительно возрасти, когда имеется много взаимодействующих сервисов, баз данных и конечных точек. Kubernetes может помочь и здесь.

Kubernetes обеспечивает безопасность всей экосистемы, поддерживает управление доступом на основе ролей (Role-Based Access Control, RBAC) и подробные правила разрешений. Кроме того, поды управляются специальным пользователем с именем *Service Account*, который имеет доступ к серверу Kubernetes API, обычно с ограниченным пространством имен пользователя. Кроме того, Kubernetes предоставляет специальный API для управления паролями и сертификатами, который называется *секретом (Secret)*. Секрет представляет собой том, который монтируется платформой в под во время выполнения, а его значение хранится в базе данных etcd вместе с состоянием и конфигурациями кластера.

etcd (https://etcd.io/) — это распределенная база данных «ключ — значение», используемая в Kubernetes для хранения состояния кластера. Содержимое БД может шифроваться, и в таком случае только администратор кластера сможет получить доступ к нему.

Как уже говорилось, микросервисы обычно взаимодействуют между собой с помощью вызовов HTTPS REST, используя сертификаты, которые управляются с помощью секретов. Контейнеры и Kubernetes обеспечивают хороший начальный уровень безопасности приложений, основываясь на котором Java-разработчики могут внедрять передовые методы обеспечения безопасности приложений.

Мониторинг

Измерение потребления ресурсов играет важную роль в современных архитектурах и тем более в облачных средах, имеющих модель оплаты «по факту использования». Трудно оценить, сколько вычислительных ресурсов потребуется вашему приложению в условиях повышенной нагрузки, а переоценка может увеличить затраты. Kubernetes позволяет осуществлять мониторинг на уровне операционной системы и приложения с помощью своей экосистемы API и инструментов.

Популярным облачным инструментом для сбора метрик на уровне платформы и приложения является Prometheus (https://prometheus.io/) — база данных временных рядов, способная экспортировать метрики из кластера Kubernetes и приложений при использовании языка запросов PromQL (https://oreil.ly/tYn9Q).

Метрики также могут помочь Kubernetes решить, когда следует масштабировать ваше приложение в ту или иную сторону в зависимости от наблюдаемой нагрузки. Масштабированием можно управлять с помощью своих метрик, таких как количество потоков выполнения JVM или размер очереди, и превратить мониторинг в упреждающий инструмент, позволяющий расширять возможности ваших сервисов. Prometheus также дает возможность отправлять оповещения и сигналы тревоги, что поможет планировать автоматические действия в приложениях, когда ситуация требует незамедлительного реагирования.

Java-разработчики могут также взаимодействовать с Prometheus и метриками внутри Kubernetes с помощью Micrometer (https://micrometer.io/) — инструмента с открытым исходным кодом, предлагающего механизм регистрации метрик основных типов. Он доступен для любых рабочих нагрузок на основе JVM и весьма популярен в проектах Spring Boot и Quarkus, служит для организации взаимодействий с Prometheus и Kubernetes. «То же, что SLF4J, но для метрик».

Трассировка

Наблюдаемость — еще одна ключевая характеристика современных архитектур, а измерение задержки между вызовами REST API — значимый аспект управления приложениями на основе микросервисов. Крайне важно обеспечить надежность взаимодействий вкупе с минимальной задержкой. По мере увеличения количества микросервисов небольшая задержка в некоторой части архитектуры может привести к прерыванию обслуживания пользователя. В таких ситуациях Kubernetes помогает от-

ладить большинство операционных проблем, возникающих при переходе на распределенную архитектуру.

Jaeger (https://www.jaegertracing.io/) — это популярный инструмент с открытым исходным кодом, который подключается к Kubernetes и обеспечивает возможность наблюдения. Он использует распределенную трассировку для отслеживания движения запросов через различные микросервисы. Jaeger позволяет визуально представить потоки вызовов в панели мониторинга и, кроме того, часто интегрируется с сервисными сетками. Jaeger помогает разработчикам наблюдать за распределенными транзакциями, оптимизировать производительность и уменьшать задержки, а также анализировать основные причины.

Журналирование

Как уже говорилось, одно обращение к приложению на основе микросервисов, такому как Coolstore, может привести к каскаду вызовов различных сервисов, взаимодействующих друг с другом. Важно следить и наблюдать за приложением, а также хранить вспомогательную информацию в журналах. Подход к журналированию в современных приложениях изменился. Если в монолитах обычно поддерживается несколько файлов журналов, хранящихся в разных каталогах на диске сервера приложений, то распределенные приложения поддерживают *поток* (stream) журналов. Распределенное приложение может быстро масштабироваться и перемещаться на другие узлы или даже в другие облака, поэтому нет смысла обращаться к какому-то одному экземпляру, чтобы получить журналы; следовательно, необходима распределенная система журналирования.

Kubernetes упрощает журналирование. По умолчанию он предлагает возможность доступа к журналам подов путем чтения стандартных потоков приложения, таких как STDOUT (стандартный вывод) и STDERR (стандартный вывод ошибок). Соответственно, приложение должно

не записывать сообщения в файл журнала, находящийся в определенном пути, а отправлять их в стандартные потоки.

Журналы по-прежнему можно хранить в определенных каталогах, которые в Kubernetes могут быть постоянными, но такой подход считается антипаттерном.

Kubernetes также поддерживает возможность взаимодействий с распределенными системами журналирования, такими как Elasticsearch (https://elastic.co/), документно-ориентированной базой данных NoSQL с открытым исходным кодом, основанной на Apache Lucene (https://lucene.apache.org/), и хранить в них журналы и события. В состав Elasticsearch обычно включаются программа пересылки, такая как Fluentd (https://fluentd.org/), и панель инструментов для визуализации журналов, такая как Kibana (https://www.elastic.co/kibana/). Все вместе они образуют стек EFK (Elasticsearch, Fluentd, Kibana). С его помощью разработчики могут просматривать журналы из нескольких микросервисов в агрегированном представлении через панель управления Kibana, а также выполнять запросы на языке запросов Kibana Query Language (KQL).

Распределенное журналирование считается стандартом де-факто для облачных приложений, и Kubernetes может интегрироваться и взаимодействовать со многими приложениями, такими как EFK, чтобы обеспечить централизованный доступ к журналам для всего кластера.

CI/CD

Непрерывная интеграция (Continuous Integration, CI) — этап разработки программного обеспечения, на котором объединяется код разных членов команды или разные функции. Обычно CI включает слияние кода (интеграцию), сборку приложения (контейнера) и выполнение основных тестов в эфемерной среде.

Непрерывная доставка (Continuous Delivery, CD) — набор методов автоматизации различных аспектов доставки программного обеспечения. Один из этих методов называется конвейером доставки и представляет собой автоматизированный процесс, определяющий шаги, которые должны пройти изменения в коде или конфигурации, чтобы в конечном итоге достичь промышленной среды.

Вместе данный этап и набор часто называют CI/CD, и это один из ключевых технологических инструментов в методологии DevOps.

Современные сервисы должны быстро реагировать на изменения или проблемы. Мы можем осуществлять трассировку, мониторинг и журналирование распределенных архитектур и также должны иметь возможность быстрее обновлять приложения на основе микросервисов. Конвейеры — лучший способ развертывания приложений в промышленной среде (рис. 6.3).

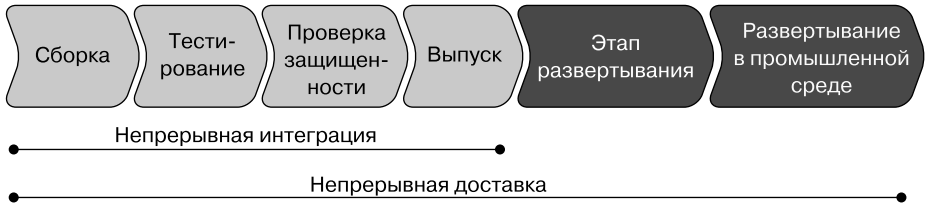


Рис. 6.3. Непрерывная интеграция и непрерывная доставка

Конвейер — это набор шагов, выполняемых последовательно или параллельно и создающих и тестирующих приложение во всех промежуточных средах, прежде чем оно наконец попадет в промышленную среду. Он может быть полностью автоматизированным или взаимодействовать с внешними инструментами, предназначенными для утверждения шагов вручную (например, Service Now, JIRA и т. д.). Kubernetes может взаимодействовать со многими внешними инструментами CI/CD, такими как Jenkins (https://jenkins.io/), а также предоставляет собственную подсистему CI/CD под названием Tekton (https://tekton.dev/).

Tekton — это система CI/CD для Kubernetes, расширяющая Kubernetes API и предоставляющая ресурсы для создания конвейеров. Она использует каталог задач (https://oreil.ly/Oxx5P), входящий в комплект Tekton, для комбинирования в конвейеры, такие как Maven или задачи Source-to-Image.

Tekton можно установить в Kubernetes с помощью оператора из OperatorHub.io (https://operatorhub.io/).

Для создания конвейеров в Kubernetes Tekton предоставляет следующие ресурсы.

- *Задачи Task* — набор повторно используемых и слабосвязанных шагов, выполняющих определенную функцию (например, создание образа контейнера). Задачи выполняются в Kubernetes как поды, а шаги в задачах отображаются в контейнеры.
- *Конвейеры Pipeline* — это списки задач, которых требует создание и/или развертывание приложения.
- *Исполнитель задачи TaskRun* — выполняет экземпляр задачи Task и представляет результат выполнения.
- *Исполнитель конвейера PipelineRun* — выполняет экземпляр конвейера Pipeline и представляет результат выполнения, включающий несколько экземпляров исполнителей задач TaskRun.

Вот пример конвейера Tekton для Quarkus-микросервиса Inventory, который также можно найти в репозитории GitHub к книге (https://oreil.ly/modernentjava):

```
apiVersion: tekton.dev/v1alpha1
kind: Pipeline
metadata:
  name: inventory-pipeline
spec:
  resources:
  - name: app-git
    type: git
```

```
- name: app-image
  type: image
tasks:
- name: build
  taskRef:
    name: s2i-java-11
  params:
    - name: TLSVERIFY
      value: "false"
  resources:
    inputs:
    - name: source
      resource: app-git
    outputs:
    - name: image
      resource: app-image
- name: deploy
  taskRef:
    name: kubectl
  runAfter:
    - build
  params:
  - name: ARGS
    value:
      - rollout
      - latest
      - inventory-pipeline
```

Java-разработчики также могут создавать конвейеры Pipeline и задачи Task для Tekton и управлять ими непосредственно из кода с помощью Java-клиента Fabric8 Tekton. Этот вариант дает полный контроль из одной точки и позволяет обходиться без использования внешних манифестов, таких как файлы YAML.

Сначала импортируйте зависимость Maven в файл POM:

```
<dependencies>
    <dependency>
        <groupId>io.fabric8</groupId>
        <artifactId>tekton-client</artifactId>
        <version>${tekton-client.version}</version>
    </dependency>
</dependencies>
<properties>
    <tekton-client.version>4.12.0</tekton-client.version>
</properties>
```

Затем используйте Tekton Java API для создания задач Task или конвейера Pipeline:

```
package io.fabric8.tekton.api.examples;

import io.fabric8.tekton.client.*;
import io.fabric8.tekton.resource.v1alpha1.PipelineResource;
import io.fabric8.tekton.resource.v1alpha1.PipelineResourceBuilder;

public class PipelineResourceCreate {

  public static void main(String[] args) {
    try ( TektonClient client = ClientFactory.newClient(args)) {
      String namespace = "coolstore";
      PipelineResource resource = new PipelineResourceBuilder()
        .withNewMetadata()
        .withName("client-repo")
        .endMetadata()
        .withNewSpec()
        .withType("git")
        .addNewParam()
        .withName("revision")
        .withValue("v4.2.2")
        .endParam()
        .addNewParam()
        .withName("url")
        .withValue("https://github.com/modernizing-java-applications-book/
          inventory-quarkus.git")
        .endParam()
        .endSpec()
        .build();

      System.out.println("Created:" + client.v1alpha1().pipelineResources().
        inNamespace(namespace).create(resource).getMetadata().getName());
    }
  }
}
```

Отладка микросервисов

Распределенные архитектуры имеют множество преимуществ, но при этом создают некоторые проблемы. Даже если в конечном итоге код будет запускаться внутри кластера Kubernetes, он все равно будет разрабаты-

ваться (преимущественно) локально, с использованием IDE, компилятора и т. д. Цикл разработки можно представить по-разному. В общем случае имеется два цикла (рис. 6.4). Тот, что ближе к разработчику, называется внутренним циклом. В нем вы итеративно программируете, тестируете и отлаживаете свои функции. Другой цикл, более далекий от разработчика, называется внешним. В нем ваш код запускается внутри образа контейнера, который вы должны создать, отправить и развернуть. И этот цикл занимает намного больше времени.

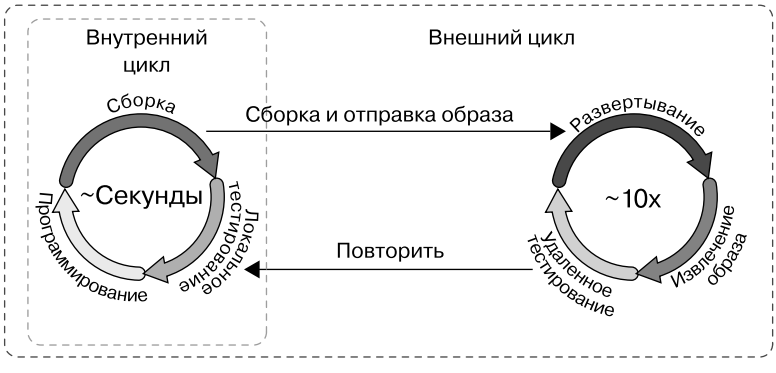


Рис. 6.4. Внутренний и внешний циклы

Внешний цикл — часть CI/CD, а внутренний — это то, где вы программируете и тестируете свое приложение, прежде чем запустить конвейер Tekton, который развернет это приложение в Kubernetes. Отладка микросервисов тоже часть внутреннего цикла.

Разработчики могут применять разные подходы к отладке микросервисов:

- использовать Docker Compose (https://oreil.ly/ULV5g) и развертывать все сервисы локально;
- использовать minikube (https://oreil.ly/1ogSc) или любые другие локальные кластеры Kubernetes и развертывать сервисы в них;
- заменять сервисы, участвующие во взаимодействиях, их фиктивными моделями.

Docker Compose помогает создавать контейнеры, способные запускаться на любых хостах Docker без Kubernetes. Он часто используется при локальной разработке, позволяя управлять несколькими контейнерами, но не отображается ни в какие целевые кластеры Kubernetes, поэтому поддержка локальной среды разработки, отдельной от целевого, может быть затруднена.

Все это допустимые подходы, но иногда во взаимодействиях участвуют внешние сервисы, доступные только в удаленном кластере Kubernetes, которые сложно или невозможно смоделировать.

Microcks (https://microcks.io/) — это инструмент отладки с открытым исходным кодом для Kubernetes, позволяющий имитировать и тестировать API. Он помогает превратить контракт API, коллекцию или проекты SoapUI в действующие имитации и позволяет ускорить разработку в Kubernetes без зависимостей.

Рассмотрим некоторые дополнительные возможности отладки микросервисов в Kubernetes.

Переадресация портов

Kubernetes позволяет входить в командные оболочки удаленных подов, чтобы решать простые задачи отладки, такие как проверка файловой системы. Кроме того, можно настроить переадресацию портов (https://oreil.ly/IAu5H) между локальным компьютером, подключенным к кластеру Kubernetes, и приложением, работающим в поде. Эта возможность может пригодиться, когда требуется соединиться с базой данных, работающей в поде, подключить административный веб-интерфейс, доступ к которому будет закрыт в промышленной среде, или, как в этом случае, подключить отладчик к JVM, под управлением которой работает наш сервер приложений.

Путем переадресации порта отладки на сервере приложений можно подключить отладчик из вашей IDE и выполнять код, работающий

в поде, в пошаговом режиме. Не забывайте, что если приложение выполняется не в режиме отладки, то вам сначала нужно включить порты отладки.

Чтобы начать отладку, следует открыть порт отладки. Например, для отладки микросервиса Inventory необходимо получить доступ к порту отладки 5005:

```
kubectl port-forward service/inventory-quarkus 5005:5005
```

Теперь любое соединение с `localhost:5005` будет переадресовываться на экземпляр Inventory, работающий в поде.

Переадресация портов действует только до тех пор, пока выполняется команда kubectl port-forward. Поскольку мы запускаем ее на переднем плане, то остановить переадресацию портов можно, нажав Ctrl+C (или Cmd+C в Mac).

Для отладки исходного кода можно использовать выбранную вами IDE, или же вы можете выполнять отладку из консоли:

```
jdb -sourcepath $(pwd)/src/main/java -attach localhost:5005
```

Режим удаленной разработки Quarkus

Quarkus предоставляет режим удаленной разработки (https://oreil.ly/rLflo), позволяющий запускать Quarkus в контейнерной среде, такой как Kubernetes, и немедленно вносить изменения в локальные файлы.

Чтобы включить его, добавьте следующий раздел в файл `application.properties`:

```
quarkus.package.type=mutable-jar ❶
quarkus.live-reload.password=changeit ❷
quarkus.live-reload.url=http://my.cluster.host.com:8080 ❸
```

❶ Если приложение определить как изменяемое, то в режиме разработки изменения можно будет оперативно применять и тестировать, не перезагружая артефакт.

❷ Пароль, используемый для защиты соединения между удаленной и локальной сторонами.

❸ URL, по которому приложение будет доступно в режиме разработки.

Создать изменяемый JAR можно с помощью Maven. Ниже показано, как можно разрешить фреймворку Quarkus развернуть приложение в Kubernetes, если вы подключены к реестру Kubernetes:

```
eval $(minikube docker-env)
```

Добавить расширение Quarkus Kubernetes можно с помощью такой команды:

```
./mvnw quarkus: add-extension -Dextensions="kubernetes"
```

Разверните приложение в Kubernetes:

```
mvn clean install -DskipTests -Dquarkus.kubernetes.deploy=true
```

И подключаетесь к нему в режиме удаленной разработки:

```
mvn quarkus:remote-dev
```

Таким образом вы можете с помощью «живого кодирования», предлагаемого фреймворком Quarkus, подключать локальный компьютер к удаленной контейнерной среде, такой как Kubernetes.

Telepresence

Telepresence (https://www.telepresence.io/) — это инструмент с открытым исходным кодом, помогающий отлаживать микросервисы в Kubernetes. Он запускает один сервис локально, подключая его к удаленному кла-

стеру Kubernetes. Telepresence не зависит от языка программирования и позволяет подключать локальную среду к любой рабочей нагрузке, выполняющейся в Kubernetes, для отладки.

Отладка приложений в Kubernetes с помощью Telepresence осуществляется очень просто. Сначала скачайте и установите Telepresence CLI (https://oreil.ly/wkwOC) и активируйте сеанс в своем кластере, после чего Telepresence прочитает файл `~/.kube/config` и подключится к Kubernetes.

Telepresence изменит настройки сети в Kubernetes, чтобы объекты Service были доступны с вашего ноутбука, и наоборот.

Когда установка и настройка Telepresence CLI на рабочей станции будут завершены, выполните следующую команду, чтобы инициализировать и проверить подключение к кластеру:

```
$ telepresence connect
```

Вы должны получить примерно такой вывод:

```
Connected to context minikube (https://192.168.39.69:8443)
```

Для примера начнем отладку микросервиса Inventory, развернутого на предыдущих шагах. Но прежде получим список приложений, доступных для отладки:

```
$ telepresence list
```

Вы должны получить примерно такой вывод:

```
inventory-quarkus-deploy: ready to intercept (traffic-agent not yet installed)
```

Чтобы начать отладку микросервиса, нужно разрешить инструменту Telepresence перехватывать внутренний трафик Kubernetes для объекта Service.

В объекте Service, представляющем микросервис Inventory, определено, что он использует порт 8080. Это можно увидеть, выполнив следующую команду:

```
$ kubectl get svc inventory-quarkus-service
```

Вы должны получить примерно такой вывод:

```
NAME                       TYPE      CLUSTER-IP     EXTERNAL-IP PORT(S)  AGE
inventory-quarkus-service ClusterIP 172.30.117.178 <none>      8080/TCP 84m
```

Теперь можно начинать перехватывать трафик, подключившись к объекту Deployment через порт, настроенный в объекте Service. Кроме того, можно указать путь к файлу, куда Telepresence должен записать переменные среды, с которыми в данный момент работает сервис:

```
$ telepresence intercept inventory-quarkus-deploy --port 8080:http --env-file
  inventory.env
```

Вы должны получить примерно такой вывод:

```
Using Deployment inventory-quarkus-deploy
intercepted
    Intercept name         : inventory-quarkus-deploy
    State                  : ACTIVE
    Workload kind          : Deployment
    Destination            : 127.0.0.1:8080
    Service Port Identifier: http
    Volume Mount Point     : /tmp/telfs-844792531
    Intercepting           : all TCP connections
```

Только что созданный файл `inventory.env` выглядит так:

```
INVENTORY_QUARKUS_SERVICE_PORT=tcp://172.30.117.178:8080
INVENTORY_QUARKUS_SERVICE_PORT_8080_TCP=tcp://172.30.117.178:8080
INVENTORY_QUARKUS_SERVICE_PORT_8080_TCP_ADDR=172.30.117.178
INVENTORY_QUARKUS_SERVICE_PORT_8080_TCP_PORT=8080
INVENTORY_QUARKUS_SERVICE_PORT_8080_TCP_PROTO=tcp
INVENTORY_QUARKUS_SERVICE_SERVICE_HOST=172.30.117.178
```

```
INVENTORY_QUARKUS_SERVICE_SERVICE_PORT=8080
INVENTORY_QUARKUS_SERVICE_SERVICE_PORT_HTTP=8080
KO_DATA_PATH=/var/run/ko
KUBERNETES_PORT=tcp://172.30.0.1:443
KUBERNETES_PORT_443_TCP=tcp://172.30.0.1:443
KUBERNETES_PORT_443_TCP_ADDR=172.30.0.1
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_SERVICE_HOST=172.30.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
LOG_LEVEL=debug
NSS_SDB_USE_CACHE=no
TELEPRESENCE_CONTAINER=inventory-quarkus
TELEPRESENCE_MOUNTS=/var/run/secrets/kubernetes.io
TELEPRESENCE_ROOT=/tmp/telfs-777636888
TERM=xterm
```

Теперь вы можете получить доступ к микросервису Inventory, как если бы были подключены к внутренней сети Kubernetes, и работать с только что полученными переменными среды:

```
curl http://inventory-quarkus-service.coolstore:8080/api/inventory/329299
```

Вы должны получить примерно такой вывод:

```
{"id":"329299","quantity":35}
```

Резюме

В этой главе вы узнали, как паттерны Kubernetes могут помочь модернизировать приложения, предоставляя платформу с множеством компонентов, расширяющих возможности приложений. Расширяемая архитектура Kubernetes, управляемая через API, позволяет использовать внешние инструменты и создавать экосистему программного обеспечения и утилит, которая напоминает модель сервера приложений для

Java Enterprise и дает возможность расширять ее. Важные задачи, такие как журналирование, мониторинг или отладка, реализованы в полном соответствии с моделью облачных вычислений, в которой приложения вездесущи и могут выполняться в нескольких местах и в нескольких облаках одновременно.

В следующей главе мы обсудим новую концепцию корпоративных приложений, экономящую ресурсы и пригодную к использованию в облаке: бессерверные вычисления.

ГЛАВА 7

Решения завтрашнего дня: бессерверные вычисления

> Вторая промышленная революция, в отличие от первой, дает нам не потрясающие воображение образы, такие как прокатные станы и расплавленная сталь, но биты в потоке информации, перемещающиеся по схемам в форме электронных импульсов. Железные машины все еще существуют, но подчиняются приказам невесомых битов.
>
> *Итало Кальвино*

Модель бессерверных вычислений получила большой импульс к развитию благодаря появлению общедоступных облаков, а в последнее время и сообществу открытого программного обеспечения, создавшему множество проектов, которые позволяют задействовать их на любой платформе.

Но что такое бессерверные вычисления? Где они используются? И как можно применять их в современных Java-приложениях?

Что такое бессерверные вычисления

Лучшее, пожалуй, определение бессерверных вычислений дается в техническом описании CNCF Serverless Whitepaper (https://oreil.ly/yYbmP):

> *«Бессерверные вычисления — это концепция сборки и запуска приложений, не требующих управления сервером. Она описывает более детализированную модель развертывания, в которой приложения, объединенные в одну или несколько функций, загружаются на платформу, а затем выполняются, масштабируются и оплачиваются в точном соответствии со спросом, имеющим место на данный момент».*

Возможность запуска приложения, которое «не требует управления сервером», — наиболее важная часть этого определения. В предыдущих главах мы видели, что Kubernetes помогает удовлетворить функциональные требования к современным архитектурам и образы контейнеров являют собой удобный способ упаковки и доставки приложений на любую облачную платформу. В бессерверных вычислениях все еще существуют серверы, однако они абстрагированы от разработки приложений. Обслуживанием и управлением таких серверов занимается третья сторона, а разработчики могут просто упаковывать свой код в контейнеры для развертывания.

Основное различие между моделью развертывания, которую мы обсуждали применительно к Kubernetes, и бессерверной моделью составляет так называемый подход *масштабирования до нуля*. При таком подходе приложение автоматически запускается по требованию при вызове и останавливается, когда не используется. Эта модель выполнения также

называется управляемой событиями и является основой бессерверных вычислений. Далее в этой главе мы обсудим бессерверные архитектуры, управляемые событиями.

Запуск приложения может инициировать целый ряд событий и приводить к результату (рис. 7.1). Это может быть одно действие или цепочка действий, в которой выходные данные одного приложения передаются на вход другого приложения. Событием может быть что угодно, например HTTP-запрос, сообщение Kafka или транзакция в базе данных. Приложение может автоматически масштабироваться до нескольких реплик, необходимых для обработки трафика, а затем останавливаться при отсутствии активности.

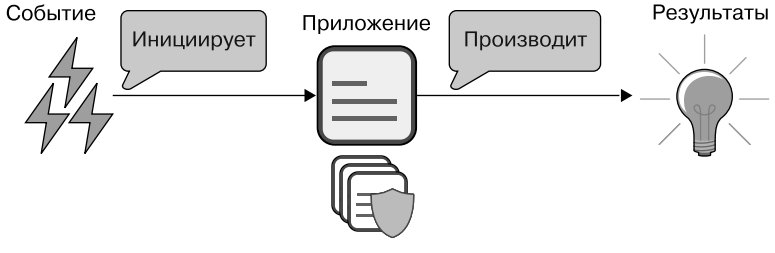


Рис. 7.1. Модель бессерверных вычислений

Архитектурная эволюция

Бессерверная модель не универсальна. В общем случае любые асинхронные приложения, легко распараллеливаемые в независимые единицы работы, хорошо укладываются в эту модель. Если посмотреть на схему, показанную на рис. 7.2, то можно заметить, что эволюция микросервисных архитектур началась с преобразования монолитных приложений в модель сервис-ориентированной архитектуры (Service-Oriented Architecture, SOA) и в настоящее время продолжила развиваться в новую модель — модель *функций*.

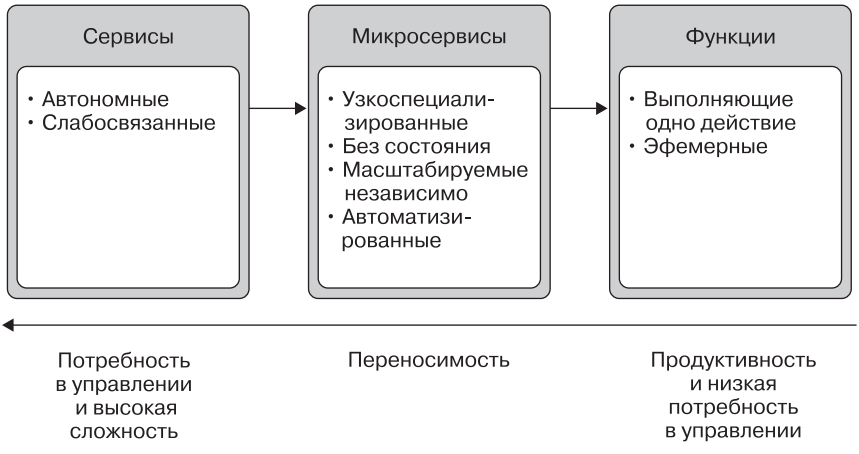


Рис. 7.2. Архитектурная эволюция

Такие функции составляют минимальные вычислительные единицы, решающие конкретные задачи. Примеры:

- обслуживание веб-обработчиков;
- преобразование данных (изображений, видео);
- генерирование PDF;
- одностраничные приложения для мобильных устройств;
- чат-боты.

Этот подход позволяет сосредоточиться на удобстве, поскольку именно оно считается *главной целью*. Неудачи допустимы, а короткие операции предпочтительнее. Вот почему бессерверная модель не подходит, например, для приложений реального времени, продолжительных задач или контекстов, где надежность и/или атомарность являются ключевыми. Разработчик должен позаботиться об успешной обработке входных и выходных данных любой задействованной бессерверной функцией. Это позволяет увеличить гибкость и масштабируемость по крайней мере для некоторой части общей архитектуры.

Варианты использования: данные, ИИ и машинное обучение

Бессерверная модель помогает избежать типичных проблем при планировании ресурсов для проектов, поскольку снижает риск избыточного и недостаточного выделения ресурсов, тем самым уменьшая затраты на поддержку IT-инфраструктуры из-за недоиспользования ресурсов. В бессерверной модели все ресурсы выделяются в строго необходимом количестве, так как приложения запускаются только при вызове и нет нужды предварительно выделять или оценивать и обновлять аппаратные ресурсы.

Это особенно ценно, когда требуется анализировать большие объемы данных в режиме реального времени, и именно поэтому бессерверные технологии привлекают внимание специалистов по данным и экспертов по машинному обучению: функции могут обрабатывать данные для анализа, являются гибкими и имеют минимальные накладные расходы. С другой стороны, бессерверные технологии плохо сочетаются с принципами проектирования существующих платформ машинного обучения. Кроме того, исследователи должны проявлять немалое терпение, запуская процессы, выполнение которых может потребовать много времени, как, например, обучение модели.

Если посмотреть на рис. 7.3, то можно увидеть пример бессерверной архитектуры для машинного обучения моделей классификации и прогнозирования. Процесс начинается с обращения к обученной модели, предназначенной для классификации группы объектов, которое приводит к вызову последовательности асинхронных функций, выполняющихся параллельно, а те, в свою очередь, определяют класс объекта на основе его характеристик и возвращают результат. Готовые к неудачам, мы допускаем, что одна из этих функций может дать сбой или не успеть выполнить задачу вовремя. Однако все они — независимые рабочие нагрузки, вследствие чего могут выполняться параллельно и без определенного порядка, и любой сбой в одной функции не влияет на всю систему. Кроме того,

механизм автоматического масштабирования бессерверной архитектуры гарантирует, что любая высокая нагрузка данных будет обрабатываться быстрее по запросу, чем при традиционных подходах.

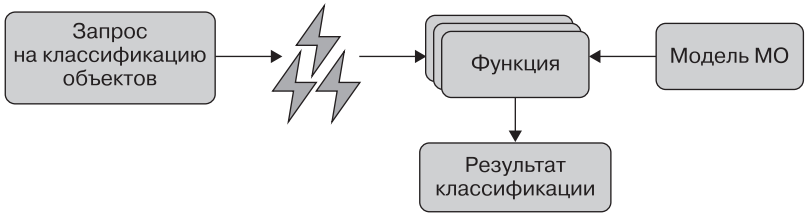


Рис. 7.3. Машинное обучение с бессерверными технологиями

Варианты использования: периферийные вычисления и Интернет вещей

Нас повсюду окружают периферийные устройства и устройства Интернета вещей (Internet of Things, IoT): от голосовых помощников до домашней автоматизации. В настоящее время чуть ли не каждый предмет в нашем доме можно подключить к Интернету и взаимодействовать с ним с помощью какого-либо управляющего приложения. Будучи разработчиком, вы можете реализовать внутреннюю логику работы устройства или логику управляющего приложения.

Примером такого сценария для Java-разработчиков может служить проект Quarkus for IoT (https://oreil.ly/0Lmiu), цель которого — сбор данных о загрязнении с датчиков на устройствах Raspberry Pi (https://raspberrypi.org/) с помощью Quarkus и контейнеров как на самих устройствах, так и на стороне сервера. При этом серверная сторона реализована как бессерверное приложение, чтобы обеспечить высокую масштабируемость для обслуживания большого количества датчиков, данные из которых могут поступать скачкообразно.

Проект также предлагает очень хорошее руководство по реализации архитектур IoT поверх Red Hat OpenShift, как показано на рис. 7.4.

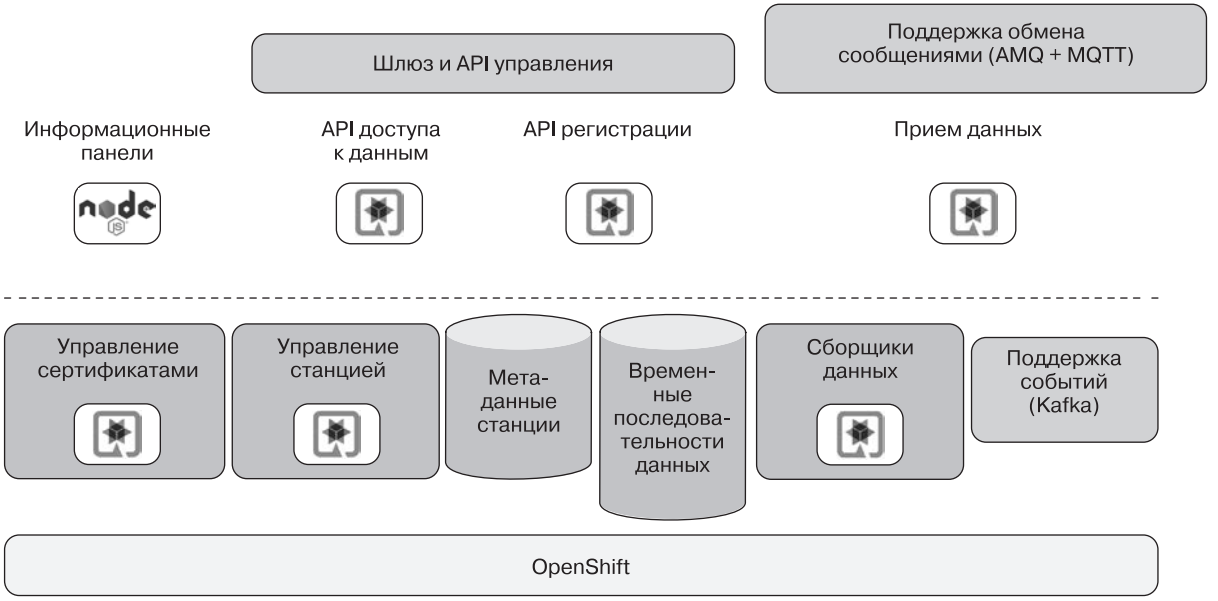


Рис. 7.4. Архитектура проекта Quarkus for IoT

Бессерверные вычисления в этой архитектуре используются для масштабирования микросервисов Quarkus, обрабатывающих сообщения, которые поступают из устройств по протоколу MQTT, потоков Kafka, а также сборщиков данных. Они делают архитектуру завершенной и надежной, а также экономически эффективной, поскольку ресурсы выделяются, только когда необходимы.

Knative: бессерверное решение для Kubernetes

По сути, бессерверные решения — это механизм функции как сервис (Functions as a Service, FaaS), предлагающий разработчикам более удобный способ упаковки и развертывания приложений. Часто, особенно в общедоступных облаках, где упаковка приложений в контейнеры также автоматизирована, бессерверные технологии и FaaS прекрасно дополняют друг друга. Однако приложение, масштабируемое до нуля, не обязательно должно быть функцией. Как уже отмечалось, бессерверные вычисления не являются чем-то, что возможно только в общедоступных облаках. Например, любой может использовать эту модель в любом кластере Kubernetes благодаря проекту с открытым исходным кодом под названием Knative (https://knative.dev/).

Мы еще вернемся к обсуждению FaaS ниже в этой главе.

Решение Knative обеспечивает поддержку бессерверных вычислений в Kubernetes и приложений, масштабируемых до нуля и управляемых событиями, а также предлагает более высокий уровень абстракции для случаев использования обычных приложений.

Knative легко установить в Kubernetes через оператор с OperatorHub.io (https://oreil.ly/ni1wkr).

Knative состоит из двух основных компонентов:

- *Knative Serving* — заботится о масштабировании до нуля, создавая все необходимые ресурсы Kubernetes (например, Pod, Deployment, Service, Ingress);
- *Knative Eventing* — компонент подписки, доставки и управления событиями в кластере и за его пределами (например, сообщения Kafka, внешние сервисы).

Knative позволяет легко превратить приложение для Kubernetes в бессерверное приложение. Ниже показан пример определения Knative Service для Quarkus-микросервиса Inventory созданного в главе 2.

Полный исходный код можно найти в репозитории книги на GitHub (https://oreil.ly/kyQfu):

```
apiVersion: serving.knative.dev/v1
kind: Service ❶
metadata:
 name: inventory-svc
spec:
 template:
   spec:
     containers:
       - image: docker.io/modernizingjavaappsbook/inventory-quarkus:latest ❷
         ports:
          - containerPort: 8080
```

❶ Это определение Knative Service — нестандартного ресурса, представляющего бессерверную рабочую нагрузку в Kubernetes.

❷ Применяется тот же образ контейнера, который мы использовали в определении объекта Deployment. При использовании Knative Service объекты Service и Deployment создаются автоматически.

Получить бессерверную версию микросервиса Inventory можно, создав объект Knative Service с помощью следующей команды:

```
kubectl create -f kubernetes/ksvc.yaml
```

Knative также предоставляет удобный интерфейс командной строки kn, позволяющий создавать объекты Knative Service и управлять всеми бессерверными компонентами Knative. Дополнительная информация по этой теме доступна в официальной документации (https://knative.dev/docs/getting-started/).

Сразу после этого можно убедиться, что новый Knative Service был создан:

```
kubectl get ksvc
```

Вы должны получить примерно такой вывод:

```
NAME           URL
LATESTCREATED        LATESTREADY          READY  REASON
inventory-svc  http://inventory-svc.coolstore.192.168.39.69.nip.io
  inventory-svc-00001  inventory-svc-00001  True
```

Как видите, на основе Knative Service были автоматически созданы все определения Kubernetes, такие как Pod, Deployment и Service. Благодаря этому отпадает необходимость поддерживать их и можно положиться на единственный объект, управляющий развертыванием и сетью:

```
kubectl get deploy,pod,svc

NAME                                          READY  UP-TO-DATE  AVAILABLE  AGE
deployment.apps/inventory-svc-00001-deployment  1/1    1           1          58m

NAME                                          READY  STATUS   RESTARTS  AGE
pod/inventory-svc-00001-deployment-58...8-81h9b 2/2    Running  0         13s

NAME                                  TYPE          CLUSTER-IP     EXTERNAL-IP
service/inventory-svc                 ExternalName  <none>
kourier-internal.kourier-system.svc.cluster.local  80/TCP         58m
service/inventory-svc-00001           ClusterIP     10.109.47.140  <none>
service/inventory-svc-00001-private   ClusterIP     10.101.59.10   <none>
```

Скрытым образом трафик к Knative Service маршрутизируется через сеть Knative. Вызов микросервиса Inventory приведет к созданию пода, если к этому моменту приложение не было запущено:

```
curl http://inventory-svc.coolstore.192.168.39.69.nip.io/api/inventory/329299
```

Вы должны получить примерно такой вывод:

```
{"id":"329299","quantity":35}
```

По прошествии некоторого времени, в течение которого не поступит новых запросов, приложение будет автоматически масштабировано до нуля и все поды будут остановлены:

```
kubectl get deploy
NAME                              READY  UP-TO-DATE  AVAILABLE  AGE
inventory-svc-00001-deployment  0/0    0           0          75m
```

Бессерверные архитектуры, управляемые событиями

События окружают нас повсюду. Как отмечалось в предыдущем разделе, HTTP-запрос может инициировать запуск приложения, бездействовавшего в этот момент, что согласуется с моделью бессерверных вычислений, представленной на рис. 7.1. Но существует множество событий, таких как появление сообщения Kafka, изменения в потоке данных или любого события из Kubernetes, на которые приложение может подписаться.

Большой популярностью в этом сценарии пользуется паттерн обмена сообщениями «публикация — подписка» (https://oreil.ly/ThcHL), позволяющий множеству отправителей посылать сообщения в сущность на сервере, часто называемую темой, а получателям — подписаться на указанную тему

для получения сообщений. В бессерверной модели приложение может регистрироваться и подключаться для обработки входящих событий. Одним из примеров для Kubernetes является компонент Knative Eventing (https://oreil.ly/NF2gB), реализующий спецификацию CloudEvents (https://cloudevents.io/), определяющую порядок описания данных в событиях, доставляемых по нескольким протоколам и в нескольких форматах (таких как Kafka, AMQP и MQTT), обобщенным способом (https://oreil.ly/oGI1q).

С помощью Knative Eventing производители и потребители событий могут действовать независимо. Запуск Knative Service инициируется источником событий через брокер, как показано на рис. 7.5. Цель фреймворка обработки событий — отделить все от всего. Отправитель не вызывает подписчиков напрямую и даже не знает, сколько у него подписчиков. Вместо этого обмен данными осуществляют брокеры и триггеры.

Чтобы не полагаться на передачу входящего запроса через все микросервисы, запустить сервис Inventory можно с помощью произвольного события HTTP.

Для этого сначала нужно создать Knative Broker. Ниже приводится пример определения Knative Broker для микросервиса Inventory, созданного в главе 2. Полный исходный код можно найти в репозитории книги на GitHub (https://oreil.ly/8Gwys):

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
 name: default
 namespace: coolstore
```

Затем создать брокер:

```
kubectl create -f kubernetes/kbroker.yaml
```

Вы должны получить примерно такой вывод:

```
NAME     URL   AGE  READY  REASON
default  http://broker-ingress.knative-eventing...coolstore/default
 11s  True
```

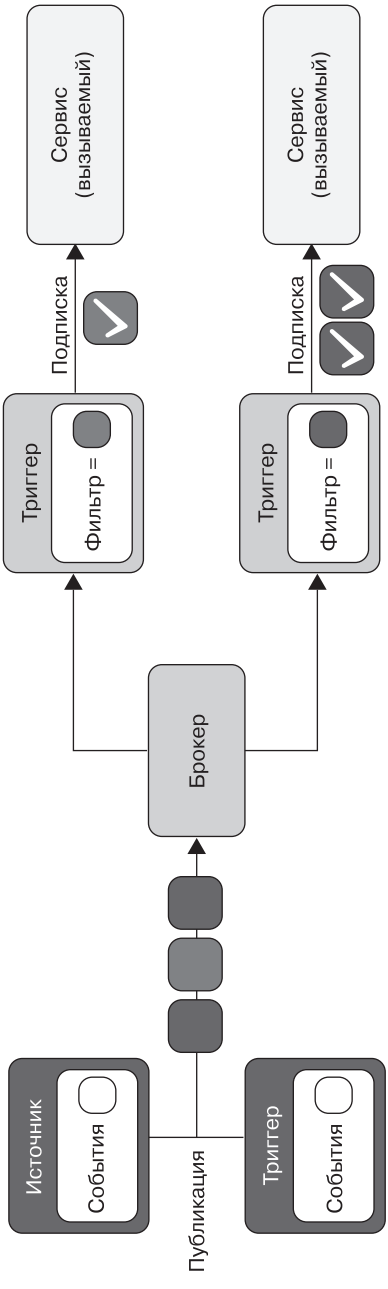


Рис. 7.5. Архитектура Knative Events

Здесь используется внутренняя сеть Kubernetes, поэтому любая конечная точка, определяемая нами, представляет Service Kubernetes в формате полного доменного имени (Fully Qualified Domain Name, FQDN), доступный только внутри кластера.

Теперь создадим триггер для запуска микросервиса Inventory. Это может быть любое событие, соответствующее спецификации CloudEvents. В данном случае используем HTTP-запрос из другого пода.

Ниже приводится пример триггера Knative Trigger для микросервиса Inventory, который мы создали в главе 2. Соответствующий исходный код можно найти в репозитории книги на GitHub (https://oreil.ly/GmG6r):

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: inventory-trigger
spec:
  broker: default ❶
  filter:
    attributes:
      type: web-wakeup ❷
  subscriber:
    ref:
     apiVersion: serving.knative.dev/v1 ❸
     kind: Service
     name: inventory
```

❶ Имя брокера.

❷ Тип атрибута. Его можно использовать для фильтрации событий, запускающих микросервис.

❸ Имя Knative Service, к которому нужно подключиться и активировать по событию.

Создадим триггер Knative Trigger:

```
kubectl create -f kubernetes/ktrigger.yaml
```

Вы должны получить примерно такой вывод:

```
NAME               BROKER   SUBSCRIBER_URI  AGE  READY  REASON
inventory-trigger  default  http://inventory.coolstore...local/  10s  True
```

Теперь можно сымитировать внешнее событие, запускающее микросервис. В данном случае это простой вызов HTTP, но таким событием может быть и передача потока информации из базы данных с помощью Debezium или сообщение Kafka.

Debezium.io (https://debezium.io/) — это платформа с открытым исходным кодом для сбора данных, которая обеспечивает потоковую передачу данных из популярных баз данных, таких как PostgreSQL, MySQL и т. д. Больше информации доступно в онлайн-документации (https://oreil.ly/bFEJi).

Загрузить минимальный образ контейнера, содержащий утилиту `curl`, готовый для запуска непосредственно в Kubernetes в виде пода, и отправить HTTP-запрос `POST` брокеру Knative Broker для запуска микросервиса можно с помощью следующей команды:

```
kubectl run curl --image=radial/busyboxplus:curl -ti \
  --curl -v \
  "http://broker-ingress.knative-eventing.svc.cluster.local/coolstore/default" \
  -X POST \
  -H "Ce-Id: wakeup" \
  -H "Ce-Specversion: 1.0" \
  -H "Ce-Type: web-wakeup" \ ❶
  -H "Ce-Source: web-coolstore" \ ❷
  -H "Content-Type: application/json" \
  -d ""
```

❶ Атрибут, который мы определили в Knative Broker выше.

❷ Имя события.

Вы должны получить примерно такой вывод:

```
> POST /coolstore/default HTTP/1.1
> User-Agent: curl/7.35.0
```

```
> Host: broker-ingress.knative-eventing.svc.cluster.local
> Accept: */*
> Ce-Id: wakeup
> Ce-Specversion: 1.0
> Ce-Type: web-wakeup
> Ce-Source: web-coolstore
> Content-Type: application/json
>
< HTTP/1.1 202 Accepted
< Date: Wed, 16 Jun 2021 11:03:31 GMT
< Content-Length: 0
<
```

Теперь посмотрим, запустился ли под с микросервисом Inventory:

```
NAME                                              READY   STATUS    RESTARTS   AGE
curl                                              1/1     Running   1          30m
inventory-svc-00001-deployment-58485ffb58-kpgdt   1/2     Running   0          7s
```

Функция как сервис для Java-приложений

Выше мы обсудили, как Knative Serving помогает упростить обслуживание нескольких объектов Kubernetes и масштабирование до нуля позволяет оптимизировать потребление ресурсов за счет уменьшения или увеличения масштабов приложений, когда это необходимо. Но есть еще один уровень абстракции, помогающий автоматически создавать и развертывать приложения в соответствии с бессерверной моделью: модель FaaS, представленная ранее.

FaaS — это модель вычислений, управляемая событиями; согласно ей разработчики пишут приложения, которые автоматически развертываются в контейнерах, полностью управляемых платформой, а затем запускаются в соответствии с моделью масштабирования до нуля. Применение этой модели освобождает вас как разработчика от необходимости писать что-то вроде манифеста Kubernetes. Вы можете просто написать логику приложения и позволить платформе самой упаковать приложение в контейнер и развернуть его в кластере как бессерверное приложение с масштабированием до нуля.

Популярные общедоступные облачные бессерверные решения, такие как AWS Lambda, Azure Functions или Google Cloud Run, предоставляют удобный SDK для разработки функций на самых популярных языках программирования, позволяющий выполнять упаковку и развертывание в модели FaaS. Кроме того, доступны решения с открытым исходным кодом, такие как Apache OpenWhisk (https://openwhisk.apache.org/) или проект Fn (https://fnproject.io/), реализующие FaaS с Docker. В следующих подразделах мы сосредоточимся на Knative и Kubernetes, поскольку на протяжении всей книги обсуждали, как Kubernetes обеспечивает полную экосистему, упрощающую миграцию корпоративных приложений Java в облако.

Развертывание функций для приложений Java

Функции — это фрагменты кода, доставляемого в соответствии с бессерверной моделью, и их можно переносить между различными конфигурациями инфраструктуры. Жизненный цикл функции, начиная с написания кода, спецификации и метаданных, показан на рис. 7.6. Сборка автоматически выполняется позже, а на этапе развертывания производится публикация функции на платформе. В этот жизненный цикл входит также механизм обновлений, который запускает новую сборку и новую публикацию, когда обнаруживаются изменения в коде.

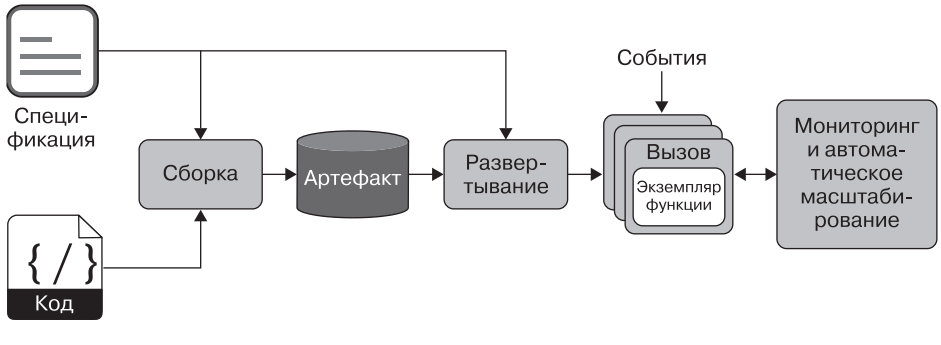


Рис. 7.6. Модель развертывания функций

Boson Function CLI (func)

Boson Function CLI (https://oreil.ly/lKYKc) — это интерфейс командной строки с открытым исходным кодом и инфраструктура, которая подключается к Knative, чтобы предоставлять возможности FaaS в Kubernetes. С помощью этого инструмента можно не писать манифесты Kubernetes и не создавать образы контейнеров самостоятельно, так как все это будет сделано автоматически:

```
$ func
...
Поддерживаемые команды:
 build      Собрать проект функции в виде образа контейнера
 completion Сгенерировать сценарий для bash, fish и zsh
 create     Создать проект функции
 delete     Удалить функцию из кластера
 deploy     Развернуть функцию
 describe   Показать информацию о функции
 emit       Отправить CloudEvent в конечную точку функции
 help       Вывести справку о выбранной команде
 list       Вывести список функций
 run        Запустить функцию локально
 version    Показать версию
 ...
```

Вы можете скачать последнюю версию утилиты func с официального сайта (https://oreil.ly/d6oXo) и настроить ее для своей системы.

Функции можно развернуть в любом кластере Kubernetes, настроенном для поддержки бессерверных рабочих нагрузок, таком как Knative.

В настоящее время утилита `func` поддерживает следующие языки программирования и фреймворки:

- Golang;
- Node.js;
- Python;

- Quarkus;
- Rust.

Создадим Quarkus-функцию в пространстве имен `coolstore`, созданном в предыдущих разделах. Полный исходный код функции можно найти в репозитории книги на GitHub (https://oreil.ly/M3yPE).

Чтобы создать новую Quarkus-функцию, выполните следующую команду, указав параметр `-l` для выбора языка:

```
$ func create -l quarkus quarkus-faas
```

Вы должны получить примерно такой вывод:

```
Project path: /home/bluesman/git/quarkus-faas
Function name: quarkus-faas
Runtime: quarkus
Trigger: http
```

Эта команда создала заготовку проекта Maven для Quarkus с файлом POM, содержащим все необходимые зависимости:

```
$ tree
.
├── func.yaml ❶
├── mvnw
├── mvnw.cmd
├── pom.xml ❷
├── README.md
└── src
    ├── main
    │   ├── java
    │   │   └── functions
    │   │       ├── Function.java ❸
    │   │       ├── Input.java
    │   │       └── Output.java
    │   └── resources
    │       └── application.properties
    └── test
        └── java
            └── functions
                ├── FunctionTest.java
                └── NativeFunctionIT.java
8 directories, 11 files
```

❶ Этот файл содержит информацию о конфигурации проекта функции.

❷ Файл POM для этого проекта Quarkus.

❸ Класс Java, содержащий аннотации и код для запуска функции.

Добавим некоторые определения в файл конфигурации функции `func.yaml`, чтобы преобразовать функцию в образ контейнера, выполняемый в Kubernetes:

```
name: quarkus-faas ❶
namespace: "coolstore" ❷
runtime: quarkus ❸
image: "docker.io/modernizingjavaappsbook/quarkus-faas:latest" ❹
imageDigest: ""
trigger: http ❺
builder: quay.io/boson/faas-quarkus-jvm-builder ❻
builderMap:
 default: quay.io/boson/faas-quarkus-jvm-builder
 jvm: quay.io/boson/faas-quarkus-jvm-builder
 native: quay.io/boson/faas-quarkus-native-builder
env: {} ❼
annotations: {} ❽
```

❶ Имя функции.

❷ Пространство имен Kubernetes, в котором будет развернута функция.

❸ Среда выполнения языка для функции, объявляемая во время создания.

❹ Имя образа для функции после ее создания.

❺ Событие, которое запускает функцию. Например, `http` для простых HTTP-запросов, как в этом случае, или `event` для функций, запускаемых событиями CloudEvent.

❻ Указывает образ построителя пакетов, который следует использовать для сборки функции.

❼ Ссылка на любые переменные среды, которые будут доступны функции во время выполнения.

❽ Аннотации для функции, которые будут использоваться для маркировки элементов.

func создает функции и преобразует их в образы контейнеров с помощью Buildpack (https://buildpacks.io/), популярного проекта с открытым исходным кодом, используемого для сборки исходного кода в образ выполняемого контейнера приложения.

Рассмотрим файл POM:

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <modelVersion>4.0.0</modelVersion>
 <groupId>org.acme</groupId>
 <artifactId>function</artifactId>
 <version>1.0.0-SNAPSHOT</version>
 <properties>
   <compiler-plugin.version>3.8.1</compiler-plugin.version>
   <maven.compiler.parameters>true</maven.compiler.parameters>
   <maven.compiler.source>1.8</maven.compiler.source>
   <maven.compiler.target>1.8</maven.compiler.target>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
   <quarkus-plugin.version>1.13.0.Final</quarkus-plugin.version>
   <quarkus.platform.artifact-id>quarkus-universe-bom</quarkus.platform.
    artifact-id>
   <quarkus.platform.group-id>io.quarkus</quarkus.platform.group-id>
   <quarkus.platform.version>1.13.0.Final</quarkus.platform.version> ❶
   <surefire-plugin.version>3.0.0-M5</surefire-plugin.version>
 </properties>
 <dependencyManagement>
   <dependencies>
     <dependency>
       <groupId>${quarkus.platform.group-id}</groupId>
       <artifactId>${quarkus.platform.artifact-id}</artifactId>
       <version>${quarkus.platform.version}</version>
       <type>pom</type>
       <scope>import</scope>
     </dependency>
   </dependencies>
 </dependencyManagement>
 <dependencies>
   <dependency>
     <groupId>io.quarkus</groupId>
     <artifactId>quarkus-funqy-knative-events</artifactId> ❷
   </dependency>
...
```

```
 </dependencies>
...
 <profiles>
   <profile> ❸
     <id>native</id>
     <activation>
       <property>
         <name>native</name>
       </property>
     </activation>
     <build>
       <plugins>
         <plugin>
           <artifactId>maven-failsafe-plugin</artifactId>
           <version>${surefire-plugin.version}</version>
           <executions>
             <execution>
               <goals>
                 <goal>integration-test</goal>
                 <goal>verify</goal>
               </goals>
               <configuration>
                 <systemPropertyVariables>
                   <native.image.path>${project.build.directory}/${project.
                     build.finalName}-runner</native.image.path>
                   <java.util.logging.manager>org.jboss.logmanager.LogManager
                     </java.util.logging.manager>
                   <maven.home>${maven.home}</maven.home>
                 </systemPropertyVariables>
               </configuration>
             </execution>
           </executions>
         </plugin>
       </plugins>
     </build>
     <properties>
       <quarkus.package.type>native</quarkus.package.type>
     </properties>
   </profile>
 </profiles>
</project>
```

❶ Версия Quarkus.

❷ Зависимость Quarkus Funqy — Java API для сред FaaS.

❸ Профиль для создания двоичных приложений Quarkus.

Quarkus Funqy (https://oreil.ly/1CNPK) — часть поддержки бессерверных рабочих нагрузок в Quarkus, предоставляющая переносимый Java API для функций, развертываемых в различных средах FaaS, таких как AWS Lambda, Azure Functions, Knative и Knative Events (Cloud Events). Funqy — это абстракция, охватывающая несколько разных поставщиков сервисов FaaS и протоколов. Оптимизирована для быстрого запуска небольших рабочих нагрузок и предоставляет для этого простой фреймворк, не требующий больших накладных расходов.

Рассмотрим исходный код функции Java, сгенерированной в файле `src/main/java/functions/Function.java`:

```
package functions;
import io.quarkus.funqy.Funq;
public class Function {
   @Funq ❶
   public Output function(Input input) { ❷
       return new Output(input.getMessage());
   }
}
```

❶ Чтобы объявить функцию, достаточно снабдить метод аннотацией `@Funq` из Quarkus Funqy API.

❷ В качестве входных и выходных данных можно использовать классы Java. Они должны соответствовать соглашению JavaBean и иметь конструктор по умолчанию. Здесь мы используем Bean-компоненты `Input` и `Output`.

Посмотрим на исходный код JavaBean-компонента `Input`, сгенерированный в файле `src/main/java/functions/Input.java`, который будет использоваться для представления входных сообщений:

```
package functions;

public class Input {
   private String message;

   public Input() {}
```

```
    public Input(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

А так выглядит исходный код JavaBean-компонента `Output`, сгенерированного в файле `src/main/java/functions/Ouput.java`:

```
package functions;

public class Output {
    private String message;

    public Output() {}

    public Output(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Теперь мы готовы собрать функцию. По умолчанию Boson CLI будет подключаться к локальному экземпляру Docker, чтобы создать контейнер с пакетами, а затем отправить его в реестр контейнеров, указанный в файле конфигурации `func.yaml`:

```
$ func build
```

В будущих версиях Boson CLI также сможет делегировать этап сборки в Kubernetes через Tekton.

Через несколько минут вы должны получить примерно такой вывод:

```
Function image built: docker.io/modernizingjavaappsbook/quarkus-faas:latest
```

Создав функцию, ее можно протестировать локально перед развертыванием в Kubernetes:

```
$ func run
```

Вы должны получить примерно такой вывод:

```
exec java -Dquarkus.http.host=0.0.0.0 -Djava.util.logging.manager=org.jboss.
logmanager.LogManager -XX:+ExitOnOutOfMemoryError -cp . -jar /layers/dev.
boson.quarkus-jvm/app/app.jar
__ ____  __  _____   ___ __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2021-06-25 16:51:24,023 INFO [io.quarkus] (main) function 1.0.0-SNAPSHOT on JVM
(powered by Quarkus 1.13.0.Final) started in 1.399s. Listening on:
http://0.0.0.0:8080
2021-06-25 16:51:24,027 INFO [io.quarkus] (main) Profile prod activated.
2021-06-25 16:51:24,028 INFO [io.quarkus] (main) Installed features: [cdi,
 funqy-knative-events]
```

Откройте другой терминал и убедитесь, что процесс запущен:

```
$ docker ps | grep modernizingjavaappsbook/quarkus-faas:latest
cd1dd0ccc9b2  modernizingjavaappsbook/quarkus-faas:latest  "/cnb/process/web"
 3 minutes ago  Up 3 minutes  5005/tcp, 127.0.0.1:8080->8080/tcp
  musing_carson
```

Попробуйте обратиться к нему:

```
$ curl \
 -X POST \
 -H "Content-Type: application/json" \
 -d '{"message":"Hello FaaS!"}' \
http://localhost:8080
```

Вы должны получить примерно такой вывод:

```
{"message":"Hello FaaS!"}
```

Теперь развернем созданный контейнер в Kubernetes и позволим Knative управлять им как приложением с масштабированием до нуля. После отправки запроса функции через HTTP Knative запустит ее автоматически и остановит, когда она не будет использоваться какое-то время:

```
$ func deploy
```

Через несколько секунд вы должны увидеть примерно такой вывод:

```
Deploying function to the cluster
  Function deployed at URL: http://quarkus-faas.coolstore.192.168.39.69.nip.io
```

Наконец, вызовите свою функцию в Kubernetes!

```
$ curl \
  -X POST \
  -H "Content-Type: application/json" \
  -d '{"message":"Hello FaaS on Kubernetes!"}' \
http://quarkus-faas.coolstore.192.168.39.69.nip.io
```

Вы должны получить примерно такой вывод:

```
{"message":"Hello FaaS on Kubernetes!"}
```

Вы можете убедиться, что новый под запущен в кластере Kubernetes в пространстве имен `coolstore`:

```
kubectl get pods -n coolstore
NAME                                          READY  STATUS   RESTARTS  AGE
curl                                          1/1    Running  3         9d
quarkus-faas-00001-deployment-5b789c84b5-kc2jb  2/2    Running  0         80s
```

Кроме того, вы можете увидеть, что был создан новый сервис Knative:

```
kubectl get ksvc quarkus-faas -n coolstore
NAME         URL                     LATESTCREATED      LATESTREADY        ...
quarkus-faas http://quarkus...nip.io quarkus-faas-00001 quarkus-faas-00001 ...
```

Теперь можно запросить информацию о вновь развернутой функции, выполнив следующую команду:

```
kubectl describe ksvc quarkus-faas -n coolstore
```

Вы должны получить примерно такой вывод:

```
Name:        quarkus-faas
Namespace:   coolstore
Labels:      boson.dev/function=true
             boson.dev/runtime=quarkus
Annotations: serving.knative.dev/creator: minikube-user
             serving.knative.dev/lastModifier: minikube-user
API Version: serving.knative.dev/v1
Kind:        Service
Metadata:
 Creation Timestamp: 2021-06-25T17:14:12Z
 Generation:         1
....
Spec:
 Template:
   Metadata:
     Creation Timestamp: <nil>
   Spec:
     Container Concurrency: 0
     Containers:
       Env:
         Name:  BUILT
         Value: 20210625T171412
       Image:   docker.io/modernizingjavaappsbook/quarkus-faas@
    sha256:a8b9cfc3d8e8f2e48533fc885c2e59f6ddd5faa9638fdf65772133cfa7e1ac40
       Name:    user-container
       Readiness Probe:
         Success Threshold: 1
         Tcp Socket:
           Port: 0
       Resources:
     Enable Service Links: false
     Timeout Seconds:      300
 Traffic:
   Latest Revision: true
   Percent:         100
Status:
 Address:
   URL: http://quarkus-faas.coolstore.svc.cluster.local
```

```
 Conditions:
   Last Transition Time:       2021-06-25T17:14:22Z
   Status:                     True
   Type:                       ConfigurationsReady
   Last Transition Time:       2021-06-25T17:14:22Z
   Status:                     True
   Type:                       Ready
   Last Transition Time:       2021-06-25T17:14:22Z
   Status:                     True
   Type:                       RoutesReady
 Latest Created Revision Name: quarkus-faas-00001
 Latest Ready Revision Name:   quarkus-faas-00001
 Observed Generation:          1
 Traffic:
   Latest Revision: true
   Percent:         100
   Revision Name:   quarkus-faas-00001
 URL:               http://quarkus-faas.coolstore.192.168.39.69.nip.io
Events:
 Type   Reason  Age   From               Message
 ----   ------  ----  ----               -------
 Normal Created 4m15s service-controller Created Configuration "quarkus-faas"
 Normal Created 4m15s service-controller Created Route "quarkus-faas"
```

Резюме

В этой главе вы увидели, как можно создавать современные приложения, следуя модели бессерверных вычислений. Познакомились с некоторыми наиболее распространенными вариантами использования и архитектурами, с которыми разработчики на Java будут работать сегодня и завтра. Периферийные вычисления, Интернет вещей, прием данных и машинное обучение — все это контексты, в которых архитектуры, управляемые событиями, являются естественным выбором, а бессерверные технологии и Java могут играть стратегическую и вспомогательную роль. Мы обсудили технологию FaaS — последний этап эволюции в разработке программного обеспечения и то, как Kubernetes может автоматизировать весь жизненный цикл приложений, развертываемых как несвязанные, асинхронные, легко распараллеливаемые процессы, называемые функциями.

В настоящее время разработчики на Java имеют полный набор фреймворков, инструментов и платформ, таких как Kubernetes, которые могут помочь модернизировать архитектуры, внедрять инновации и предугадывать решение задач, возникающих в современном IT-контексте. Этот контекст становится все более разнородным, вездесущим, крупномасштабным и облачным.

> Дал я крылья тебе, и на них высоко и свободно
> Ты полетишь над землей и над простором морей [...]
> Всем, кому дороги песни, кому они дороги будут,
> Будешь знаком ты, пока солнце стоит и земля.
>
> *Феогнид из Мегары*[1]

[1] Перевод В. В. Вересаева. http://veresaev.lit-info.ru/veresaev/stihi/feognid-elegii.htm.

Об авторах

Маркус Эйзеле (Markus Eisele) — руководитель программы адаптации разработчиков в Red Hat. Имеет более чем 14-летний опыт работы с серверами Java EE от разных производителей и часто выступает с докладами на международных конференциях по Java. Имеет звание Java Champion, одно время входил в состав экспертной группы Java EE. Основатель JavaLand — крупнейшей в Германии конференции по Java. Всегда рад обучать разработчиков интеграции микросервисных архитектур в существующие платформы, а также успешному созданию устойчивых приложений с помощью Java и контейнеров. Кроме того, является автором книги *Modern Java EE Design Patterns and Developing Reactive Microservices* (O'Reilly; https://oreil.ly/38mgZ). Вы можете пообщаться с Маркусом в Twitter (https://twitter.com/myfear) и LinkedIn (https://linkedin.com/in/markuseisele).

Натале Винто (Natale Vinto) — инженер-программист, имеющий более чем 10-летний опыт работы в области информационных и информационно-коммуникационных технологий, а также солидный опыт работы в области телекоммуникаций и операционных систем Linux. Обладая богатым опытом разработки на Java, Натале несколько лет работал архитектором решений для OpenShift в компании Red Hat. В настоящее время является представителем разработчиков (developer advocate) для OpenShift в Red Hat, помогает сообществу и клиентам успешно внедрять Kubernetes и облачные стратегии. Вы можете пообщаться с ним в Twitter (https://twitter.com/natalevinto) и LinkedIn (https://linkedin.com/in/natalevinto).

Иллюстрация на обложке

На обложке книги изображены онагры *(Equus hemionus)*. Эти животные внешне похожи на диких ослов, но они немного меньше по размерам, имеют более бледную шерсть и светло-коричневую полосу на спине. Онагры живут в равнинных районах пустынь и окружающих их предгорьях от Монголии до Саудовской Аравии. В этих суровых условиях рацион онагров состоит из скудных трав, кустарников и листвы, но животные всегда стараются держаться поблизости от открытых водоемов и рек.

Онагры по своей природе непугливы и очень любопытны. Это делает их особенно уязвимыми. В 1971 году онагры были объявлены охраняемым видом, но, несмотря на все усилия, в дикой природе осталось всего 395 взрослых онагров.

Многие животные, изображенные на обложках O'Reilly, находятся под угрозой исчезновения; все они важны для нашего мира.

Иллюстрацию для обложки нарисовала Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры из книги *Brehms Tierleben*.