O'REILLY®

Java в облаке

SPRING BOOT, SPRING CLOUD, CLOUD FOUNDRY

ПИТЕР®

Джош Лонг, Кеннет Бастани

Cloud Native Java

Designing Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry

Josh Long and Kenny Bastani

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

Джош Лонг, Кеннет Бастани

Java в облаке

SPRING BOOT, SPRING CLOUD, CLOUD FOUNDRY

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2019

ББК 32.988.02-018
УДК 004.738.2
Л76

Лонг Джош, Бастани Кеннет

Л76 Java в облаке. Spring Boot, Spring Cloud, Cloud Foundry. — СПб.: Питер, 2019. — 624 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-0713-1

Хотите потягаться с гигантами современных облачных технологий? Работать как Amazon, Netflix или Etsy? Ответ очевиден: вам нужна облачная разработка под Java/JVM, позволяющая освоить новейшие технологии, открывающие путь к облакам — в первую очередь Spring Boot и Cloud Foundry. Всему этому вы научитесь, прочитав фундаментальную книгу «Java в облаке». Вы не только узнаете, как устроены современные облачные технологии для серьезных решений, но и освоите основы микросервисной архитектуры, непрерывной интеграции и доставки, сможете целиком переработать накопившийся унаследованный код и достойно отвечать на самые сложные вызовы, которые ставит перед нами современная Java-экосистема.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.2

ISBN 978-1449374648 англ.

ISBN 978-5-4461-0713-1

Краткое содержание

Оглавление

Часть II. Веб-сервисы

Часть III. Интеграция данных

Часть IV. Промышленная эксплуатация

Посвящается Еве и Макани.

С любовью от дяди Джоша

Посвящается моему дедушке Аббасу, который ждал появления своего имени на обложке книги целых сто лет.

Кенни

Предисловие Джеймса Уоттерса

В одну и ту же реку невозможно войти дважды.

Гераклит

Когда летом 2015 года я сидел с Джошем Лонгом в кафе на Венис-Бич, у меня появилось ощущение, что мы находимся на пороге чего-то большого. В то время наша платформа, ориентированная на работу в облачной среде, Pivotal Cloud Foundry, быстро набирала популярность в качестве среды выполнения облачных приложений, и разработчики очень хотели побольше узнать о нашей новой технологии — Spring Boot. В сочетании с набиравшей популярность Spring Boot выход на сцену Spring Cloud обещал произвести настоящий фурор. Я заметил: «Они будут прекрасно сочетаться, и это уже происходит».

К работе были привлечены неимоверные силы. В тот самый момент, когда ИТ-директора отчаянно искали пути повышения производительности труда проектировщиков, среда Spring Boot предлагала микросервисы и вполне приемлемый подход DevOps к разработке корпоративных приложений. Развивающаяся интеграция Spring Boot и PCF превратила развертывание в производственной среде в простой конвейер, API был отозван, Spring Cloud поставила первую в мире сеть микросервисов, и появился стандартный облачный подход к Java.

Изменение в порядке разработки, которое совсем не казалось поверхностным, уникальная комбинация этих технологий изменили структуру поставки продукта большими организациями. Слишком долго разработчикам не удавалось выполнять развертывание в производственной среде из-за эксплуатационной сложности устаревших серверов приложений и шаблонов Java. Нередко можно было услышать мрачные клиентские истории о многодневных развертываниях. Мы знали, что наша платформа изменит жизнь клиентов. После внедрения Spring Boot на PCF они

стали присылать нам благодарственные письма с описаниями обновлений в производственной среде, занимающих минуты, а не месяцы.

С 2015 года мы наблюдаем отличные результаты, и организации, перешедшие на эту технологию, отмечают как минимум 50-процентное повышение скорости создания программ, более чем двойное улучшение показателей наработки на отказ и времени простоя при восстановлении после сбоев, а также возможность управления десятками тысяч JVM-машин с небольшими командами обслуживания платформы. Самое главное, что организации, внедрившие технологии, рассмотренные в данной книге, больше времени уделяют решению вопросов, связанных с клиентами и рынками, и существенно меньше времени тратят на переживания по поводу сложностей разработки и сопровождения программ.

Страница за страницей эта книга подробно описывает наиболее важные модели современной разработки корпоративного программного обеспечения (ПО). С большим трудом накопленный Джошем и Кенни практический опыт взаимодействия со многими ведущими предприятиями в сфере производства сквозит во множестве приведенных здесь примеров.

Я призываю каждого проектировщика и ИТ-руководителя, готового воспользоваться всей мощью адаптивности и гибкости в своих организациях, получить удовлетворение от этой работы.

Джеймс Уоттерс (James Watters),
первый вице-президент Pivotal Cloud.
Foundry, Pivotal, @wattersjames

Предисловие Рода Джонсона

Мы становимся живыми свидетелями одного из самых перспективных преобразований в истории нашей отрасли: перехода от старых архитектур к облаку и от традиционного разделения на разработчиков и операторов к унифицированному понятию DevOps. Задача преобразования решается в данной книге путем объяснения возможностей и проблем написания приложений, ориентированных на работу в облачной среде, и предоставления четкого руководства относительно того, как это делается.

Преобразования не происходят в одночасье. Одна из самых сильных сторон данной книги — ее акцентированность на том, как добраться до облака оттуда, где вы оказались сегодня, опираясь на имеющийся опыт. В частности, в разделе о достижении паритета сервисов предоставляется замечательный материал о том, как перейти от традиционных методов работы к облачным.

Эта книга предлагает выверенный баланс теории и практики, объясняя и принципы построения архитектуры современных приложений, и эффективные, опробованные способы ее реализации. Практика требует выбора не только языка программирования, но и первичной среды с открытым кодом, поскольку современные приложения неизменно создаются с опорой на оправдавшие себя решения, имеющие открытый код. Если ваш выбор пал на язык Java — данная книга для вас.

Слухи о моей смерти сильно преувеличены.

Марк Твен

Несколько лет назад сообщения о «смерти» Java были обычным явлением. Сегодня же Java процветает, и эта книга напомнит вам причину данного явления. Язык

Java обрел новую жизнь отчасти потому, что Java-технологии привели к созданию современных приложений, готовых к работе в облачной среде. Двумя решающими факторами стали проекты с открытым кодом Netflix и Spring. В этой книге проделана замечательная работа по охвату обоих факторов.

Первоначально задуманные для упрощения сложностей Java EE минувшей эпохи, основные идеи Spring выдержали испытание временем и отлично зарекомендовали себя применительно к облачным приложениям. Более десяти лет назад мы говорили о Spring-треугольнике: внедрении зависимостей, абстракции переносимых сервисов и AOP. Все это в равной степени актуально и сегодня, когда чистое разобщение бизнес-логики и ее среды стало важнее, чем когда-либо.

Основа данной книги — освещение Spring Boot, нового способа использования Spring в эпоху микросервисов, который сегодня активно внедряется на производствах. Spring Boot упрощает создание новых сервисов Spring любой степени детализации и их развертывание в современной контейнеризированной среде. В то время как традиционное «корпоративное» Java-приложение было монолитным, запускаемым на все еще большом сервере приложений, среда Spring Boot развернула все в сторону упрощения и повышения эффективности: сервис должным образом становится центральной фигурой и развертывается с сервером, обладающим ресурсами, которых вполне хватает для его запуска.

В данной книге рассматривается последняя работа команды Spring — Spring Cloud Stream и усовершенствованная поддержка комплексного тестирования, а также развивающаяся интеграция с проектами Netflix с открытым кодом.

Я рад отметить продолжение нововведений Spring и концентрацию этой среды на упрощении труда проектировщиков. Хотя последние пять лет я работал со средой Spring только в качестве пользователя, мне радостно видеть, как она процветает и побеждает все новые источники сложности. Сегодня я продолжаю решение этой задачи упрощения в среде Atomist, где мы стремимся сделать для команд разработчиков и процесса создания ПО все то, что Spring сделала для приложений Java, предоставляя возможность автоматизировать все важные процессы. Среда Spring обеспечивает простую производительную структуру и абстракцию для работы со всем, чем занимается Java-разработчик. Среда Atomist нацелена на решение тех же задач в отношении исходного кода проекта, построения систем, отслеживания проблем, развернутой среды и т. д. Сюда же включена мощная автоматизация разработки: от создания новых проектов до усовершенствования сотрудничества команд с помощью Slack и отслеживания событий развертывания.

Фундаментальный строительный блок автоматизации — тестирование. В данной книге мне особенно понравилось всестороннее освещение этого процесса, проливающее свет на сложные проблемы, которые касаются тестирования микросервисов. Мне также понравилось наличие большого количества листингов программ, снабженных подробными комментариями.

Приятно писать предисловие для книги, сотворенной друзьями и посвященной любимой мною технологии. Те, кто знаком с Джошем, знают, что он отличный собеседник. Он так же хорош во владении печатным словом, как и в живом программировании. Джош и Кенни — страстные, любознательные и хорошо информированные экскурсоводы, и я с удовольствием путешествую с ними. Я многому научился на этом пути и уверен, что и вам это удастся.

Род Джонсон (Rod Johnson),
создатель, генеральный директор Spring Framework,
Atomist, @springrod

Введение

Быстрее! Быстрее! Быстрее!!! Всем хочется двигаться быстрее, но не все знают, как этого добиться. Требования рынка растут все быстрее, появляются и новые возможности, но некоторые из нас просто не в состоянии идти в ногу со временем! Чем отличается обычное предприятие от таких, как Amazon, Netflix и Etsy? Нам известно, что эти компании разрослись до неимоверных масштабов и все же каким-то образом сохраняют свое преимущество, опережая конкурентов. Как?

Довести идею от концепции до клиента, понять, что идея превратилась во что-то полезное и ценное, можно, проделав большой объем работы. На своем пути к вводу в эксплуатацию любой продукт проходит через множество различных этапов: доведение пользовательского восприятия до проектировщиков, разработка, тестирование, запуск в производство. Исторически сложилось так, что работа замедлялась на каждом из этих пунктов. Со временем общими усилиями мы оптимизировали различные части процесса. У нас имеются облачные вычисления, поэтому отпала необходимость в многоярусных сервисах. Мы используем разработку на основе тестирования и непрерывную интеграцию, предназначенную для автоматизации проверок. Программы выпускаются в небольших пакетах — микросервисах, что позволяет сократить область изменений и их стоимость. Мы принимаем идеи, заложенные в devops, чтобы способствовать всестороннему охвату системы и чувству товарищества между разработчиками и операторами, сокращая тем самым неоправданные затраты, связанные с рассогласованностью приоритетов. Все собранное вместе и есть то, что мы подразумеваем под *облачными технологиями*.

Разработчики программных продуктов, являясь специалистами и практиками в своей отрасли, оказались сегодня на перепутье. Существуют надежные, стабильные самообслуживающиеся решения с открытым кодом для инфраструктуры, тестирования, связки, непрерывной интеграции и поставки, сред разработки и облачных платформ. Эти элементы позволяют организациям сосредоточиться на менее затратной поставке ценностей более высокого порядка и в более широком масштабе.

Для кого эта книга

В основном данная книга предназначена для разработчиков Java- и JVM-машин, которые ищут способы создания более качественного ПО в короткие сроки с помощью Spring Boot, Spring Cloud и Cloud Foundry. Она для тех, кто уже слышал шум, поднявшийся вокруг микросервисов. Возможно, вы уже поняли, на какую стратосферную высоту взлетела среда Spring Boot, и удивляетесь тому, что сегодня предприятия используют платформу Cloud Foundry. Если так и есть, то эта книга для вас.

Зачем мы ее написали

В компании Pivotal мы помогаем клиентам превращать их компании в передовые организации цифровых технологий, обучая их непрерывной поставке и применению Cloud Foundry, Spring Boot и Spring Cloud. Мы видим, что срабатывает (а что — нет), и хотим зафиксировать состояние дел, как это определено нашими пользователями, и поделиться своим опытом. Мы не претендуем на всесторонний охват, но при этом стараемся затронуть ключевые понятия — те области мира облачных вычислений, в которых вы собираетесь работать, и дать о них ясное представление.

Структура книги

Книга имеет следующую структуру:

- в главах 1 и 2 рассказывается, чем хорошо «облачное» мышление, а затем предоставляется краткий экскурс в Spring Boot и Cloud Foundry;
- в главе 3 представлены способы конфигурации приложения Spring Boot; позже на этот навык мы будем опираться повсеместно;
- в главе 4 рассматриваются способы тестирования Spring-приложений, от самых простых компонентов до распределенных систем;
- в главе 5 описана незначительная реструктуризация, которую можно выполнить для перемещения приложения на такую облачную платформу, как Cloud Foundry, что позволит получить ряд ценных свойств от самой платформы;
- в главе 6 освещается вопрос создания HTTP- и RESTful-сервисов с помощью среды Spring; большей частью это будет делаться в API и в мире, ориентированном на предметную область;
- в главе 7 представлены общие способы контроля точек выдачи и поступления запросов в распределенных системах;
- в главе 8 описаны способы создания сервисов, действующих в качестве первого порта вызова для запросов, поступающих из внешнего мира;

- в главе 9 рассматриваются способы управления данными в среде Spring с помощью Spring Data, тем самым закладываются основы для мышления с прицелом на ориентацию в предметной области;
- в главе 10 изучаются методы интеграции распределенных сервисов и данных с использованием имеющейся в среде Spring архитектуры, управляемой событиями и ориентированной на рассылку сообщений;
- в главе 11 описаны способы применения возможностей масштабирования такой облачной платформы, как Cloud Foundry, для того, чтобы справиться с долговременными рабочими нагрузками;
- в главе 12 освещаются некоторые способы управления состоянием в распределенных системах;
- в главе 13 рассматриваются методы создания систем, поддерживающих отслеживаемость и оперативное реагирование;
- в главе 14 изучаются способы создания сервис-брокеров для платформ, подобных Cloud Foundry; сервис-брокеры подключают к облачной платформе такие сохраняющие состояние сервисы, как очереди сообщений, базы данных и блоки кэш-памяти;
- в главе 15 описаны творческие идеи, заложенные в непрерывную поставку. Это, может быть, и последняя глава, но для вас это всего лишь начало вашего путешествия.

Если вы похожи на нас, то не станете читать книгу от корки до корки. Если вы действительно похожи на нас, то, скорее всего, вообще не станете читать введение. Однако предположим, что вы видите эти строки. На такой случай хотим дать упреждающие советы.

- Чем бы вы ни занимались, прочтите главы 1 и 2. Они заложат основы для усвоения всего остального материала.
- В главах с 3-й по 6-ю вводятся такие понятия, которые должен знать любой проектировщик, работающий со средой Spring. Они актуальны как для прежних, так и для новых Spring-приложений. Фактически глава 5 посвящена вопросам превращения старых приложений (как ориентированных на использование среды Spring, так и не ориентированных на это) в новые приложения.
- В главах 7 и 8 вводятся понятия, применяющиеся в микросервисных системах, основанных на использовании протокола HTTP, например безопасность и маршрутизация.
- В главах с 9-й по 12-ю дается материал, помогающий лучше справляться с управлением и обработкой данных в распределенных системах.
- В главе 13 фактически изложена основная концепция, которая должна была быть освещена в этой книге значительно раньше, при рассмотрении ключевых концепций и тестирования, если не принимать во внимание тот факт, что она

зависит от технических концепций, которые ранее не вводились. Эксплуатируемые приложения должны быть отслеживаемыми. Эту главу следует прочитать как можно раньше. Для ее понимания достаточно знания основ. Или же перечитайте ее еще раз, добравшись до нее обычным порядком.

- В главе 14 рассматривается вопрос использования Spring, среды разработки облачных приложений, с целью создания компонентов для платформ, для облака. Материал, описанный в данной главе, прежде всего приобретает остроту в свете усилий открытых сервис-брокеров.
- И наконец, в главе 15 уточняются вопросы, связанные с непрерывной поставкой. Вся эта книга написана в понятиях непрерывной поставки, и последний раздел имеет такую фундаментальную значимость для всего, что мы стараемся сделать, что мы вынесли рассмотрение данного вопроса в заключение. Прочитайте эту главу одной из первых, а затем перечитайте ее в последнюю очередь.

Интернет-ресурсы

В Интернете имеется множество превосходных ресурсов, которые помогут продолжить ваши исследования и будут содействовать в работе:

- в GitHub-репозитории (https://github.com/cloud-native-java/) можно найти код для данной книги;
- сайт Spring (https://spring.io/) может послужить для вас универсальным магазином для всего, что касается Spring, включая документацию, форум технических вопросов и ответов и др.;
- сайт Cloud Foundry (https://www.cloudfoundry.org/) — центр притяжения той работы, которая проделана всеми соавторами учреждения Cloud Foundry. Здесь вы найдете видеоклипы, учебники, сводки новостей и др.

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины и слова, на которых сделан акцент.

`Моноширинный шрифт`

Применяется для листингов программ, а также внутри абзацев, чтобы обратиться к элементам программы вроде переменных, функций и типов данных. Им также выделены имена и расширения файлов.

`Полужирный моноширинный шрифт`

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

Курсивный моноширинный шрифт

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

Шрифт без засечек

Служит для указания URL, адресов электронной почты.

Этот рисунок указывает на совет или предложение.

Этот рисунок указывает на общее замечание.

Этот рисунок указывает на предупреждение.

Использование примеров кода

Загрузить дополнительные материалы (примеры кода, упражнения и т. д.) можно с сайта http://github.com/Cloud-Native-Java.

Книга нацелена на оказание помощи в решении практических задач. В целом, если пример кода предлагается вместе с книгой, то его можно использовать в ваших программах и документации. Вам не нужно обращаться к нам за разрешением, кроме тех случаев, когда вы воспроизводите весьма существенный объем кода. Например, написание программы, задействующей несколько фрагментов кода из этой книги, не требует разрешения, в отличие от продажи или распространения компакт-диска с примерами из книг издательства O'Reilly. Ответы на вопросы с цитированием материала данной книги и приведением примеров кода разрешения не требуют, чего не скажешь о вставке существенного объема примеров кода из этой книги в документацию по вашему программному продукту.

Если есть сомнения по поводу правомерности использования примеров кода в рамках вышеизложенного, вы можете свободно обратиться к нам за консультацией по адресу permissions@oreilly.com.

Благодарности

Прежде всего хочется поблагодарить наших невероятно терпеливых и участливых редакторов из O'Reilly: Нана Барбера (Nan Barber) и Брайана Фостера (Brian Foster).

Спасибо всем рецензентам и тем, кто нас вдохновлял, снабжал аналитическими выводами и идеями. Хочется выразить компании Pivotal и в целом всей ее экосистеме огромную признательность за поддержку. Особая благодарность каждому рецензенту, предоставившему технические отзывы, среди которых Брайан Дюссо (Brian Dussault), доктор Дэйв Сиер (Dave Syer), Эндрю Клей Шафер (Andrew Clay Shafer), Роб Уинч (Rob Winch), Марк Фишер (Mark Fisher), доктор Марк Поллак (Mark Pollack), Уткарш Надкарни (Utkarsh Nadkarni), Сабби Анандан (Sabby Anandan), Майкл Хангер (Michael Hunger), Бриджит Кромхаут (Bridget Kromhout), Расс Майлз (Russ Miles), Марк Хеклер (Mark Heckler), Мартен Дейнум (Marten Deinum), Натаниэль Шутта (Nathaniel Schutta), Патрик Крокер (Patrick Crocker) и многие другие. Спасибо вам!

И последними в списке, но не последними по значению хотелось бы поблагодарить Рода Джонсона и Джеймса Уоттерса. Вы проявили щедрость, не будучи ничем нам обязанными; вы дали нам все. Огромное вам спасибо за ваши предисловия, отзывы и напутствия.

Эта книга была написана с помощью инструментария Asciidoctor, разработанного под руководством Дэна Аллена (Dan Allen) (https://twitter.com/mojavelinux) и Сары Уайт (Sarah White) (http://twitter.com/carbonfray) из OpenDevise. Управление исходным кодом всех примеров выполняется в открытых хранилищах GitHub (https://github.com/). Код постоянно интегрировался в Travis CI (https://travis-ci.org/cloud-native-java). Компоненты сборок хранились и управлялись в личной учетной записи хранилища Artifactory, любезно предоставленной нам JFrog (https://jfrog.com/). Все

примеры запускались на Pivotal Web Services, где располагался экземпляр Cloud Foundry, управляемый Pivotal. Эти инструменты, без которых написание данной книги было бы сильно затруднено, относятся либо к программам с открытым кодом, либо запускаются сообществом единомышленников, не вынуждая нас ни к каким затратам. Огромное вам спасибо! Надеемся, что вы их присмотрите для своих следующих проектов и поддержите их, как они поддержали нас.

Теперь обращаемся к команде Spring (https://spring.io/team) и командам Cloud Foundry (https://www.cloudfoundry.org/): большое спасибо за весь созданный и протестированный вами код, которым нам не пришлось заниматься.

Джош Лонг

Хочу поблагодарить моего соавтора Кенни за то, что стал моим попутчиком в этом путешествии! Хочу выразить благодарность коллективу издательства O'Reilly за возможность работать с ними и за их невероятное терпение в условиях поджимающих сроков и нашего карабканья вверх в стремлении удержаться на вершине постоянно разрастающейся экосистемы Pivotal. Спасибо доктору Дэйву Сиеру (Dave Syer) (https://twitter.com/david_syer) за добрые слова о книге и за то, что он был моим вдохновителем. Меня носило по городам, часовым поясам, странам и континентам, а команды Spring и Cloud Foundry в Pivotal неизменно проявляли поддержку, независимо от дня или часа, — спасибо вам!

Кенни Бастани

В первую очередь хочу поблагодарить моего друга, наставника и коллегу Майкла Хангера (Michael Hunger), он был первым, кто привил мне страсть к написанию и обсуждению программ с открытым кодом. Хочу сказать спасибо и моему безмерно талантливому соавтору Джошу Лонгу (Josh Long), пригласившему меня поработать с ним над этой книгой, в то время как я запустил в эксплуатацию свои первые микросервисы Spring Boot более двух лет назад. Совместная работа над книгой была для меня невероятным приключением. С того времени как нашими совместными с Джошем усилиями на бумагу легли первые слова, мы увидели, что среда Spring Boot разрослась и стала одним из наиболее успешных когда-либо созданных проектов с открытым кодом. Сегодня рост ее популярности представлен почти 13 миллионами скачиваний в месяц, сочетанием новых приложений и непрерывного потока производственных развертываний. Достижением этого успеха команда Spring Engineering обязана 50 штатным инженерам. При этом скромном количестве специалистов Spring поддерживает более 100 проектов с открытым кодом, в числе которых компоненты среды, образцы приложений

и документация, и все это под крылом любезного спонсора — компании Pivotal. Все, что достигнуто нами в данной книге, не было бы возможным без невероятной самоотверженности разработчиков, которые в настоящее время ежегодно создают около трех миллионов новых приложений Spring Boot, готовых к работе в производственной среде.

С 2004 по 2017 год только в рамках проекта Spring Framework 381 Java-разработчик открытого кода внес 36 412 зафиксированных изменений, что составляет примерно 339 лет объединенных трудозатрат.

Часть I
Основы

1 Приложение, оптимизированное для работы в облачной среде

Приемы разработки программного обеспечения как в составе команд, так и в индивидуальном порядке постоянно совершенствуются. Разработка программ с открытым кодом обеспечила производство ПО чем-то похожим на Кембрийский взрыв инструментов, сред программирования, платформ и операционных систем, и все это сопровождается растущим вниманием к гибкости и автоматизации. Основная масса современного наиболее популярного инструментария с открытым кодом сфокусирована на функциональных особенностях, позволяющих командам разработчиков безостановочно поставлять программное обеспечение как никогда ранее быстрыми темпами, на любом уровне, от создания до практического применения.

История компании Amazon

В течение двух десятилетий, с начала 1990-х годов, книжный интернет-магазин Amazon.com со штаб-квартирой в Сиэтле превратился в крупнейшее в мире предприятие данного профиля. Теперь эта компания, известная в наши дни просто как *Amazon*, продает гораздо более широкий ассортимент товаров. В 2015 году Amazon превзошла по объему торговли компанию Walmart — наиболее заметного розничного торговца в США. Самая интересная часть истории беспрецедентного роста Amazon может быть сведена к одному вопросу: каким образом сайт, начинавшийся с простого книжного интернет-магазина, превратился в одного из мировых гигантов розничной торговли, не открыв при этом ни одной торговой точки?

Нетрудно было заметить, что мир торговли переместился и выстроился вокруг цифровых коммуникаций, чему поспособствовал бурный рост возможностей подключения к Интернету в любых уголках планеты. По мере уменьшения размеров персональных компьютеров и их превращения в повсеместно используемые в наши дни смартфоны, планшеты и часы мы стали свидетелями грандиозного роста доступности каналов распространения, преобразивших способы ведения торговли по всему миру.

Техническим развитием компании Amazon от весьма успешного книжного интернет-магазина до одной из наиболее влиятельных технологических компаний в мире, занимающихся розничной торговлей, управлял ее технический директор Вернер Фогельс (Werner Vogels). В июне 2006 года Фогельс дал интервью компьютерному журналу *ACM Queue* по вопросу быстрой эволюции технологий, которая привела к взлету компании Amazon. В нем директор рассказал об основном движущем факторе этого прогресса.

> В основном техническое развитие Amazon.com обуславливалось обеспечением непрерывного роста, **чтобы при исключительно высокой масштабируемости сохранялись доступность и производительность**.
>
> *Вернер Фогельс. A Conversation with Werner Vogels, ACM Queue 4, № 4 (2006): 14–22*

Фогельс продолжает утверждать, что компании Amazon для достижения высокой масштабируемости понадобился переход к другой структуре архитектуры ПО. Он напомнил, что сначала Amazon.com использовала монолитное приложение. Со временем, по мере того как над одним и тем же приложением стало трудиться все больше и больше команд, границы принадлежности кода стали размываться. «Отсутствие изолированности привело к отсутствию четко выраженной принадлежности», — сказал Фогельс.

Фогельс отметил, что совместно используемые ресурсы, такие как базы данных Vogels, затрудняют расширение всего бизнеса. Чем больше объем совместно применяемых ресурсов, будь то серверы приложений или базы данных, тем слабее контроль за работой при вводе компонентов программ в эксплуатацию.

> Вы создаете этот продукт, вам и обслуживать его работу.
>
> *Вернер Фогельс, технический директор компании Amazon*

Фогельс затронул общую тему, имеющую отношение и к приложениям, оптимизированным для работы в облачной среде: команды должны быть владельцами того,

что они создают. Далее он сказал: «Традиционная модель заключается в следующем: ПО перетаскивается через барьер, отделяющий создание от практического применения, работа над ним прекращается, после чего о нем забывают. Но это не про Amazon. *Вы создаете этот продукт, вам и обслуживать его работу*».

Слова, которые чаще всего цитировались основными докладчиками на отдельных ведущих конференциях по программному обеспечению *(«вы создаете этот продукт, вам и обслуживать его работу»)*, позже стали слоганом популярного движения, известного сегодня просто как *DevOps*.

Ряд приемов, о которых говорил Фогельс в 2006 году, дал начало многим популярным движениям в области разработки ПО, ныне процветающим. Можно установить связь таких подходов, как *DevOps* и *разработка микросервисов*, с идеями, высказанными Фогельсом более десяти лет назад. Идеи, аналогичные этим, вырабатывались в крупных интернет-компаниях, подобных Amazon, однако на создание соответствующих инструментов, позволяющих добиться их развития и становления в виде предложения конкретных услуг, ушло бы немало лет.

В 2006 году в Amazon был запущен новый продукт под названием Amazon Web Services (AWS). Идея, положенная в его основу, заключалась в предоставлении платформы, аналогичной той, что использовалась внутри Amazon, и выпуске ее в виде сервиса для открытого применения. Компания была заинтересована в развитии идей и инструментов, на которых основывалась эта платформа. Многие из выдвинутых Фогельсом идей уже были внедрены в платформе Amazon.com; выпуская платформу в качестве сервиса для открытого использования, компания стремилась выйти на новый рынок, названный *публичным облаком*.

Были озвучены идеи, положенные в основу этого облака. Виртуальные ресурсы могут предоставляться по требованию, при этом исключается необходимость беспокоиться относительно основной инфраструктуры. Разработчики могут просто арендовать виртуальную машину для размещения своих приложений, не нуждаясь в приобретении инфраструктуры или в управлении ею. Такой прием представлял собой вариант самообслуживания с низкой степенью риска, повышающий привлекательность публичного облака с технологией AWS, прокладывающей путь к внедрению.

Пройдут годы, прежде чем технология AWS превратится в полноценный набор сервисов и шаблонов для создания и запуска приложений, написанных для работы в публичном облаке. Хотя для создания новых программ к инжинирингу этих сервисов было привлечено множество разработчиков, многие компании с уже существующими приложениями по-прежнему испытывают сложности с переходом на новые технологии. Имеющиеся у них программы были созданы без расчета на переносимость. К тому же многие приложения все еще зависят от устаревших разработок, несовместимых с публичным облаком.

Чтобы самые крупные компании воспользовались публичным облаком, им нужно внести изменения в способ разработки их приложений.

Надежды, связанные с платформой

Платформа воспринимается сегодня как избитое понятие.

Когда речь заходит о платформах в компьютерном деле, чаще всего имеется в виду набор возможностей, помогающих либо создавать, либо запускать приложения. Лучше всего платформы обобщаются по требованиям, которые предъявляются к способам, используемым разработчиками при создании приложений.

Платформы позволяют автоматизировать решение задач, не играющих существенной роли в поддержке бизнес-требований, предъявляемых к приложению. Это помогает командам разработчиков действовать оперативно, позволяя поддерживать только функциональные компоненты, имеющие особую ценность для делового применения.

Та команда, которая написала сценарии оболочки либо наштамповала контейнеры или виртуальные машины для автоматизации разработки, уже создала своеобразную платформу. Вопрос заключается в следующем: что может дать эта платформа, каким ожиданиям соответствует? Какой объем работы потребуется для поддержки основных (или даже всех) требований для непрерывной поставки нового ПО?

При построении платформ создается инструмент, автоматизирующий набор повторяющихся приемов; последние складываются из набора требований, которые превращают ценные идеи в план.

- **Идеи:** каковы основные замыслы платформы и в чем их ценность?
- **Требования:** каковы требования, соблюдение которых необходимо для превращения наших идей в приемы?
- **Приемы:** как автоматически перевести требования в набор повторяющихся приемов?

В основу любой платформы закладываются простые идеи, которые в случае их реализации повышают дифференцированную деловую ценность за счет использования автоматизированного инструментария.

Возьмем, к примеру, платформу Amazon.com. Вернер Фогельс заявил: «Повышая изолированность компонентов программ друг от друга, команды будут иметь больше возможностей контроля тех функций, которые вводятся ими в производство».

Идея

Повышение изолированности компонентов программ друг от друга позволяет доводить части системы до практического применения независимо и в более сжатые сроки.

Закладывая эту идею в основу платформы, мы получаем возможность сформировать из нее набор требований. Они принимают форму взгляда на то, как основная идея будет воплощена на практике при автоматизации. Следующие утверждения

представляют собой категоричные требования к способам повышения изолированности компонентов.

Требования

- *Компоненты программ должны разрабатываться в виде независимо развертываемых сервисов.*
- *Вся бизнес-логика в сервисе инкапсулируется с теми данными, с которыми она работает.*
- *За пределами сервиса не должно быть прямого доступа к базе данных.*
- *Сервисы должны предоставлять веб-интерфейс для обращения к их бизнес-логике со стороны других сервисов.*

На основе этих требований складывается весьма категоричный взгляд на приемы усиления изолированности компонентов программы. Обещания, связанные с выполнением приведенных требований в виде автоматизированных механизмов, сулят командам предоставление более действенного контроля над функциями, поставляемыми для эксплуатации. Следующим шагом станет описание того, как эти требования могут быть сведены к набору повторяющихся приемов.

Приемы, выведенные исходя из этих требований, должны быть заявлены в виде коллекции обещаний. Указывая приемы в виде обещаний, мы задаем ожидания пользователей платформы относительно способов, которые они могут задействовать при создании и эксплуатации своих приложений.

Приемы

- *Командам предоставляется интерфейс самообслуживания, позволяющий обеспечить инфраструктуру, требуемую для функционирования приложений.*
- *Приложения помещаются в пакет, представляющий собой полный комплект, развертываются в среде с помощью интерфейса самообслуживания.*
- *Базы данных предоставляются приложениям в виде сервиса и должны вводиться в действие с использованием интерфейса самообслуживания.*
- *Приложение снабжается всем необходимым для регистрации в базе данных в виде переменных среды, но только после объявления явного отношения к базе данных в виде привязки к сервису.*
- *Каждому приложению предоставляется сервисный реестр, применяемый в качестве манифеста местоположений внешних сервисных зависимостей.*

Каждый из вышеперечисленных приемов принимает форму обещания для пользователя. Таким образом, суть идей, закладываемых в основу платформы, реализуется в виде требований, предъявляемых к приложениям.

В основе приложений, оптимизированных для работы в облачной среде, лежит набор требований, позволяющих сократить время, затрачиваемое на однообразную трудоемкую деятельность.

Когда технология AWS была впервые представлена публике, Amazon не заставляла ее пользователей придерживаться тех же требований, которые применялись в самой компании. Оставаясь верным своему названию *Amazon Web Services*, сама по себе технология AWS не является облачной *платформой*, а представляет собой коллекцию независимых инфраструктурных сервисов, из которых может быть составлена автоматическая оснастка, напоминающая платформу обещаний. Спустя годы после первого выпуска AWS компания Amazon начала предлагать коллекцию управляемых сервисов платформы с вариантами использования, начиная от Интернета вещей (IoT, Internet of Things) и заканчивая машинным самообучением.

Если какой-либо компании понадобится создать свою собственную платформу с нуля, время поставки приложений откладывается до полной сборки платформы. Компании, ставшие ранними пользователями AWS, должны были вместе собрать некую разновидность автоматизации, похожую на платформу. Каждой компании пришлось бы выработать набор обещаний, охватывающих основные идеи создания и ввода ПО в эксплуатацию.

Совсем недавно производство ПО ориентировалось на идею существования основного набора общих обещаний, исполняемых каждой облачной платформой. Эти обещания будут исследоваться в нашей книге с применением *платформы в виде сервиса* (Platform as a Service, PaaS) под названием Cloud Foundry. Основным предназначением последней является предоставление платформы, вбирающей в себя набор общих обещаний, касающихся быстрой разработки и применения приложений. Cloud Foundry дает эти обещания, одновременно сохраняя возможность переносимости между облачными инфраструктурами нескольких различных поставщиков.

Основное внимание в данной книге мы уделим вопросам создания Java-приложений, оптимизированных для работы в облачной среде. В первую очередь это будет касаться инструментов и сред, позволяющих сократить время, затрачиваемое на однообразные трудоемкие действия за счет использования преимуществ и обещаний платформы, оптимизированной для работы в облаке.

Принципы

Новые принципы создания ПО позволяют уделять больше внимания обдумыванию поведения наших приложений в ходе их применения. В совместных усилиях разработчиков и операторов делается более серьезный акцент на понимании характера поведения их приложений в процессе эксплуатации, с меньшим уровнем доверия к тому, как за счет сложности можно разрешить ситуацию при сбое.

Как и в случае с Amazon.com, архитектура ПО начала отходить от крупных монолитных приложений. Теперь основное внимание в вопросах архитектуры уделялось достижению высокого уровня масштабируемости без ущерба для производительности и доступности. Разбивая монолит на компоненты, инженерные организации

предпринимали усилия по децентрализации управления изменениями, предоставляя командам больше контроля над тем, как функции вводятся в эксплуатацию. Повышая изолированность между компонентами, команды создателей ПО начали вступать в мир разработки распределенных систем, фокусируясь на написании менее крупных, более специализированных сервисов с независимыми циклами выпуска.

В приложениях, оптимизированных для выполнения в облаке, используется набор принципов, позволяющих командам более свободно оперировать способами ввода функций в эксплуатацию. По мере роста распределенности приложений (в результате повышения степени изолированности, необходимой для предоставления большего контроля над ситуацией командам, владеющим приложением) возникает серьезная проблема, связанная с повышением вероятности сбоя при обмене данными между компонентами приложения. Неизбежным результатом превращения приложений в сложные распределенные системы становятся эксплуатационные сбои.

Архитектуры приложений, оптимизированных для работы в облачной среде, придают этим приложениям преимущества исключительно высокой масштабируемости, притом гарантируя их всеобщую доступность и высокий уровень производительности. Хотя такие компании, как Amazon, пользовались преимуществами высокой масштабируемости в облаке, широко доступного инструментария для создания приложений, оптимизированных для работы в облачной среде, еще не появлялось. В конечном итоге инструменты и платформа будут представлены в виде коллекции проектов с открытым кодом, поддерживаемых первопроходцем в мире открытых облачных вычислений — компанией Netflix.

Масштабируемость

Чтобы ускорить разработку ПО, нужно заранее продумывать возможности масштабирования на всех уровнях. В самом общем смысле *масштаб* обуславливается издержками, приносящими пользу. Уровень непредсказуемости, сокращающий этот полезный эффект, называется *риском*. В таких условиях приходится определять границы масштабирования, поскольку создание программ сопряжено с рисками. Риски, заложенные создателями ПО, не всегда известны операторам, то есть тем, кто занимается его эксплуатацией. Выдвигая разработчикам требования по скорейшему вводу функций в дело, мы добавляем риски к процессу эксплуатации этих функций, не испытывая никакого сочувствия к проблемам операторов.

В результате со стороны операторов возрастает недоверие к ПО, произведенному разработчиками. Дефицит доверия между обеими сторонами создает практику обвинений: люди предъявляют друг другу претензии вместо того, чтобы рассматривать системные проблемы, которые привели к возникновению или к ускорению появления проблем, оказывающих отрицательное влияние на ведение дел.

Для снятия напряженности, возникающей в традиционных структурах ИТ-организаций, следует переосмыслить порядок взаимодействия команд, занимающихся поставкой и эксплуатацией ПО. Общение операторов с разработчиками способно влиять на возможность масштабирования, поскольку со временем цели каждой из сторон расходятся. Чтобы добиться успеха в решении данного вопроса, необходимо перейти к более надежному способу разработки ПО, при котором основное внимание внутри процесса создания уделяется опыту команды операторов и который способствует взаимообучению и совершенствованию.

Надежность

Ожидания, возникающие во взаимоотношениях команд (относительно порядка разработки, режима эксплуатации, алгоритма взаимодействия с пользователем и т. д.), выражаются в виде соглашений. Заключаемые командами соглашения подразумевают предоставление или получение определенного уровня услуг. Наблюдая за тем, как команды предоставляют услуги друг другу в процессе создания ПО, можно прийти к более глубокому пониманию того, каким образом отсутствие надлежащего уровня общения способно повлечь возникновение рисков, приводящих к сбоям при дальнейшей совместной работе.

Соглашения об услугах между командами заключаются с целью уменьшить риск неожиданного поведения в общих функциях масштабирования, которые создают ценность для бизнеса. Данные соглашения носят конкретный характер, чтобы гарантировать соответствие поведения ожидаемой стоимости операций. Таким образом, услуги позволяют отдельным частям бизнеса довести до максимальных показателей его общую отдачу. Здесь целью для бизнеса, относящегося к производству ПО, является надежное прогнозирование создания ценности за счет затрат, то есть то, что мы называем *надежностью*.

Модель услуг для бизнеса — та же самая модель, которая используется при создании ПО. Именно так гарантируется надежность системы, неважно, к чему именно это относится: к программному продукту, производимому для автоматизации бизнес-функции, или к людям, обучаемым выполнению ручных операций.

Адаптивность

Мы начинаем понимать, что больше не существует лишь одного способа разработки и практического применения программного обеспечения. Благодаря внедрению адаптивных методологий и продвижению в направлении к бизнес-моделям ПО как услуги (или сервиса) — *Software as a Service* (SaaS) — комплект корпоративных приложений становится все более распределенным. Создание распределенных систем — весьма непростая задача. Переход к более распределенной архитектуре

приложений обуславливается необходимостью сокращать сроки поставки программ с меньшим риском возникновения сбоев.

Возможно, кто-то воскликнет: «Адаптивность? Она все еще актуальна?» (https://www.linkedin.com/pulse/agile-dead-matthew-kern). Адаптивность в нашем понимании относится как к целостному, общесистемному устремлению к незамедлительной поставке новой ценности, так и к замыслу по поводу быстрого реагирования на смену обстановки. Мы просто говорим о малом — об адаптивности, не касаясь при этом понятий управленческой практики. Существует множество путей, приводящих к эксплуатации продукта, и нас не интересует, какой именно методики управления вы будете придерживаться на своем пути. Главное, вам нужно усвоить, что адаптивность — полезное свойство, а не самоцель.

Современный бизнес, ориентированный на производство ПО, стремится реструктурировать процессы разработки, чтобы получить возможность более быстро внедрять программы и непрерывно вводить приложения в эксплуатацию. Компании хотят увеличить не только скорость разработки программ, но и количество создаваемых и сопровождаемых ими приложений, чтобы обслуживать различные бизнес-подразделения организации.

ПО все чаще становится для компаний конкурентным преимуществом. Все более совершенные инструменты позволяют бизнес-специалистам открывать новые источники дохода или оптимизировать бизнес-функции таким образом, чтобы можно было обеспечить быстрое внедрение инноваций.

И в центре всего этого движения находится *облако*. Когда о нем заходит речь, подразумевается весьма специфичный набор технологий, позволяющий разработчикам и операторам сопровождения пользоваться преимуществами веб-сервисов, существующих для предоставления виртуализированной вычислительной инфраструктуры и управления ею.

Компании начинают переходить от дата-центров к применению публичных облаков. Одной из таких является Netflix, популярная компания по предоставлению потокового мультимедиа по подписке.

История Netflix

На сегодняшний день Netflix — один из самых крупных в мире потоковых медиасервисов по требованию, эксплуатирующий свои онлайн-сервисы в облаке. Компания была основана в 1997 году в Скоттс-Валли, штат Калифорния, Ридом Хастингсом (Reed Hastings) и Марком Рэндольфом (Marc Randolph). Изначально Netflix предоставляла интернет-сервис по прокату DVD, позволявший клиентам вносить фиксированную ежемесячную плату за подписку на не-

ограниченные прокатные видео без дополнительной платы. Клиенты получали DVD по почте после выбора их изображения из списка и помещения в очередь с помощью сайта Netflix.

В 2008 году компания пережила серьезное повреждение своей базы данных, помешавшее доставке DVD клиентам. В те времена Netflix уже приступила к развертыванию сервисов потокового видео, предназначенных для обслуживания клиентов. Команда, занимавшаяся в Netflix потоковым видео, поняла, что аналогичный отказ в их сфере услуг подорвет будущее их бизнеса. В результате в компании было принято важное решение: нужно переходить к другому способу разработки и сопровождения ПО, гарантирующему постоянную доступность их сервисов клиентам.

Частью решения Netflix по предотвращению сбоев ее интернет-сервисов стал отход от вертикально масштабируемой архитектуры и единых точек отказов. Такая позиция была вызвана повреждением базы данных в результате использования вертикально масштабируемой реляционной базы данных. Компания переместила данные клиентов в распределенную базу данных NoSQL, а именно в проект Apache Cassandra с открытым кодом. Это было началом становления Netflix в качестве компании «облачного направления», запускающей все свои программные приложения в виде отказоустойчивых облачных сервисов с высокой степенью распределения. Netflix решила повысить надежность своих интернет-сервисов, добавляя избыточность к своим приложениям и базам данных в масштабируемой модели инфраструктуры.

Частично решение компании по переходу в облако потребовало переноса развертывания больших приложений на системы с высокой степенью распределенности. При этом специалисты компании столкнулись с серьезной проблемой: командам Netflix при переходе от собственного дата-центра к использованию публичного облака пришлось заниматься перепроектированием своих приложений. В 2009 году компания приступила к переходу на Amazon Web Services (AWS) и сконцентрировалась на достижении трех основных целей: масштабируемости, производительности и доступности.

В начале 2009 года стало ясно, что спрос резко возрастает. Известен такой факт: Юрий Израилевский (Yury Izrailevsky), вице-президент по проектированию облачных вычислений и платформ (Cloud and Platform Engineering) компании Netflix, на презентации в рамках конференции AWS re:Invent, состоявшейся в 2013 году, сказал, что спрос с 2009 года возрос в 100 раз. «Мы не смогли бы справиться с масштабированием наших сервисов, если бы пользовались своим внутренним решением», — сказал Израилевский.

Кроме того, он заявил, что преимущества масштабируемости в облаке на фоне его быстрого глобального расширения стали еще более очевидными. Израилевский сказал: «Чтобы сократить время ожидания для наших европейских клиентов, мы запустили второй облачный район в Ирландии. На развертывание нового

дата-центра на другой территории ушло бы много месяцев и миллионы долларов. Это были бы громадные вложения».

Как только Netflix начала размещать свои приложения на Amazon Web Services, сотрудники стали делиться впечатлениями в блоге компании. Многие поддерживали переход к новой архитектуре, сконцентрированной на горизонтальной масштабируемости на всех уровнях стека ПО.

Джон Цианкутти (John Ciancutti), занимавший тогда должность вице-президента технологий персонализации (Personalization Technologies) компании Netflix, в конце 2010 года выложил в блог сообщение, что «облачная среда идеально подошла для горизонтально масштабируемых архитектур. Нам не нужно гадать на месяцы вперед, какими будут наши потребности в оборудовании, хранилище и сетевой аппаратуре. Мы можем практически мгновенно программным способом получить доступ к большему объему этих ресурсов из общих пулов в Amazon Web Services».

Под возможностью «программного доступа» к ресурсам Цианкутти подразумевал, что разработчики и операторы сопровождения могут получить программный доступ к конкретным API управления, открываемым Amazon Web Services с целью предоставить клиентам средства управления для подготовки виртуализированной вычислительной инфраструктуры. RESTful API позволяют разработчикам создавать приложения, управляющие виртуальной инфраструктурой и подготавливающие ее для их основных приложений.

Слоеный пирог, показанный на рис. 1.1, изображает параметры облака, характеризуемые различными уровнями абстракции.

Предоставление управленческих услуг для управления виртуализированной вычислительной инфраструктурой — одна из основных концепций облачных вычислений и называется инфраструктурой как услугой (Infrastructure as a Service), чаще всего обозначаемой IaaS.

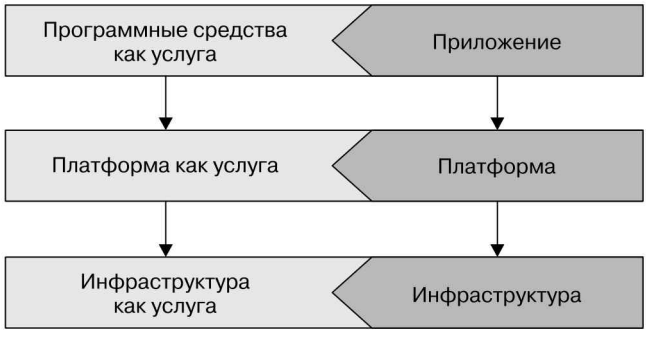


Рис. 1.1. Стек облачных вычислений

В той же публикации в блоге Цианкутти признался, что Netflix не всегда удавалось правильно спрогнозировать рост клиентуры или объемы привлечения дополнительного оборудования. Эта тема выходит на первый план для компаний, ориентированных на использование облачных технологий. В силу своих особенностей облачные технологии формируют сознание, допускающее невозможность надежных предсказаний того, где и когда понадобятся дополнительные объемы ресурсов.

На презентации, проводимой Юрием Израилевским в рамках конференции AWS re:Invent, состоявшейся в 2013 году, было сказано, что «в облаке в случае повышения объема трафика емкость хранилища можно поднять за считаные дни. Начинать можно с малых объемов, постепенно увеличивая их по мере роста трафика».

Далее он заявил: «По мере превращения нашей компании в транснациональную мы пользовались преимуществами того, что могли положиться на несколько областей присутствия Amazon Web Services по всему миру с целью дать нашим клиентам превосходные ощущения интерактивности независимо от их местонахождения».

Экономия на масштабировании, предопределившая успешное распространение AWS по всему миру, также принесла пользу и компании Netflix. По мере того как для AWS расширялись зоны доступности за счет регионов за пределами США, Netflix получала возможность распространять свои сервисы по всему миру, задействуя только те API управления, которые предоставлялись AWS.

Израилевский процитировал главный аргумент, высказываемый по поводу внедрения облачных вычислений в корпоративные информационные технологии: «Конечно, облако — это замечательно, но слишком дорого для нас». Его ответом на него стало следующее заявление: «В результате перехода Netflix на облачные технологии стоимость операций упала на 87 %. Наши затраты составляют одну восьмую от того, что нам приходилось тратить на содержание дата-центра».

Израилевский объяснил далее, почему облако дало такую экономию: «Из возможности роста, не беспокоясь о емкости буферной памяти, мы извлекаем реальную пользу. По мере роста мы можем проводить масштабирование, соответствующее нашим потребностям».

Микросервисы

В этой книге вопрос микросервисов будет рассматриваться много раз. Хотя книга не посвящена исключительно микросервисам, вы еще не раз сможете убедиться в том, что приложения, оптимизированные для работы в облачной среде, и микросервисы идут рука об руку. Один из основных замыслов, положенных в основу создания микросервисов, заключается в возможности организации как самих команд разработчиков функций, так и приложений вокруг конкретных бизнес-возможностей. Этот подход никоим образом не претендует на особую новизну. Создание системы из небольших распределенных компонентов, хорошо взаимодействующих и раздробленных таким образом, чтобы свести к минимуму все мешающее вводу

отдельных функций в эксплуатацию, было доступно уже многие десятилетия. Почему же об этом говорится сейчас? Почему такая разновидность архитектуры приобрела популярность именно в настоящее время? Может ли это иметь какое-то отношение к облаку?

Создание ПО всегда давалось с трудом. Разница между разумными действиями и простой работой зачастую есть результат того, что кто-то другой избавляет вас от ваших собственных неудачных решений. Технические решения, принятые вчера, помешают принятию правильных архитектурных решений завтра. Теперь не секрет: всем нам хочется начать все с чистого листа — *tabula rasa* — и микросервисы предоставляют способ взять неудачные вчерашние решения и разложить их на новые и, надеемся, *лучшие* варианты на завтра.

Легко понять что-нибудь небольшое, но гораздо труднее осмыслить влияние его отсутствия. Если детально исследовать качественно спроектированное монолитное приложение, то можно заметить те же стремления к достижению модульности, простоты и слабой связанности, которые наблюдаются в сегодняшних архитектурах микросервисов. Конечно, одно из главных отличий — *последовательность событий*. Нетрудно понять, как наслоения предпочтений превратили удачную конструкцию в большое и ужасное месиво. Если кто-то принял неудачное решение в отношении одного небольшого сменного блока архитектуры, то этот вариант со временем можно будет просто разбить на части. Но если тот же самый человек привлекался к работе над множеством отдельных модулей качественно спроектированного монолита, то дополнительная предыстория способна позже отрицательно повлиять на чью-либо возможность выбора желаемых решений. Поэтому мы вынуждены идти на компромисс и принимать не самые удачные решения, исходя из скудного выбора, возможность предопределения которого нам никогда не предоставлялась.

ПО, предназначенное для внесения изменений, становится подобием живого существа, которое всегда преобразуется под влиянием последовательности событий и никак не застраховано от ветров перемен в вопросах выживания. Как следствие, мы вынуждены выстраивать процесс создания с учетом будущих изменений. Мы обязаны принимать изменения, сопротивляясь при этом побуждению использовать архитектуру с заделом на будущее. В конечном счете готовность к предстоящим изменениям — генеральный план, преподносимый как адаптивная разработка. Неважно, насколько нам удастся проявить прозорливость в вопросах согласованных упреждающих усилий по созданию совершенной системы. Мы не сможем с высокой степенью надежности предсказать, как характер ее работы изменится впоследствии, так как зачастую от нас абсолютно ничего не зависит. Судьбу продукта, как правило, решает рынок. Поэтому разработку можно вести только с позиции сегодняшнего дня.

Микросервисы в настоящее время не более чем замысел. Модели и приемы создания микросервисов пребывают пока в изменчивом состоянии, продолжая коле-

баться в разные стороны, в то время как мы терпеливо дожидаемся стабильного определения.

Существует две основные силы, влияющие на скорость архитектурных изменений: микросервисы и облако. Последнее привело к резкому снижению затрат и усилий, необходимых для управления инфраструктурой. Сегодня у нас есть возможность использовать средства самообслуживания для предоставления инфраструктуры нашим приложениям *по требованию*. С этого момента началось быстрое внедрение новых, инновационных инструментов; и это постоянно побуждает нас переосмыслять и изменять наши предыдущие соглашения. То, что еще вчера было справедливо в отношении создания ПО, сегодня уже может утратить актуальность, и в большинстве случаев истина приобретает весьма туманные очертания. Теперь нужно принимать нелегкие решения на основе трудно предсказуемых предположений о том, что наши серверы являются физическими объектами. Что наши виртуальные машины обладают свойством постоянства. Что наши контейнеры не используют состояния. Наши предположения насчет уровня инфраструктуры находятся под постоянным шквалом атак от бесконечного выбора новых вариантов.

Разбиение монолита на части

Специалисты Netflix ссылаются на два основных преимущества перехода от монолита к архитектуре распределенных систем в облаке: адаптивность и надежность.

До перехода на облачную среду архитектура Netflix включала одно монолитное приложение виртуальной машины Java (Java Virtual Machine, JVM). Хотя развертывание одного большого приложения приносило массу преимуществ, основной недостаток заключался в том, что командам создателей приходилось снижать темпы работы из-за необходимости координировать вносимые ими изменения.

При создании и сопровождении ПО повышение централизованности снижает риск сбоев и повышает затраты на согласование. На координацию уходит время. Чем более централизованной является архитектура приложения, тем больше времени уходит на согласование изменений в любой его части.

Монолитные программы также страдают не слишком высокой надежностью. Когда компоненты задействуют одни и те же ресурсы на той же самой виртуальной машине, сбой в одном из них может распространиться на другие элементы, вызывая простой в работе пользователей. Риск внесения разрушительного изменения в монолит увеличивается с ростом объема усилий, необходимых командам для координации вносимых ими преобразований. Чем больше изменений, вносимых за один цикл выпуска, тем выше риск внесения разрушительного преобразования, вызывающего простой в работе пользователей. Разбивая монолит на мелкие сервисы с узкой специализацией, развертывания могут выполняться с помощью пакетов

меньшего объема с циклами выпусков, независимыми от результатов деятельности других команд.

Netflix пришлось изменить не только способ создания и сопровождения своего ПО, но и культуру его организации. Компания перешла к новой модели работы, которая называется *DevOps*. В соответствии с ней каждая команда стала группой, нацеленной на конечный результат, отходя при этом от традиционной структуры группы проектирования. В группе, нацеленной на конечный результат, команды составляли вертикальную структуру, возлагая на себя функции управления производством и сопровождением конечного продукта. У таких команд было все необходимое для создания и сопровождения их ПО.

Netflix OSS

С превращением Netflix в компанию, ориентированную на использование облачной среды, в ней также стали проявлять активность в создании программ с открытым кодом. В конце 2010 года Кевин Макэнте (Kevin McEntee), будучи тогда вице-президентом по системам и электронной торговле (Systems & Ecommerce Engineering), объявил в блоге о грядущей роли компании в разработке ПО с открытым кодом.

Макэнте заявил: «Замечательным качеством проекта с открытым кодом, решающим общую задачу, является то, что он дает импульс своему собственному развитию и долгое время поддерживается действенным циклом непрерывного совершенствования».

В последующие годы компания Netflix перевела в разряд продуктов с открытым кодом более 50 своих внутренних проектов, каждый из которых стал частью бренда *Netflix OSS*.

Руководящие сотрудники Netflix позже подтвердят желание компании сделать открытым исходный текст многих внутренних инструментов. В июле 2012 года директор по разработке облачных платформ (Director of Cloud Platform Engineering) компании Netflix Руслан Мешенберг (Ruslan Meshenberg) опубликовал в технологическом блоге компании сообщение Open Source at Netflix («Открытый код в Netflix»), где говорилось о том, что она предприняла весьма смелый шаг, переведя в разряд продуктов с открытым кодом столь большой объем своих внутренних инструментов.

Мешенберг написал в блоге, аргументируя принятый курс на открытие исходного кода, что «Netflix была одной из первых компаний, внедривших облачные технологии, переведя все свои потоковые сервисы на работу поверх AWS-инфраструктуры. Нам пришлось расплачиваться за право быть первопроходцами столкновением со множеством вопросов, острых проблем и ограничений, с которыми мы сумели справиться».

Культурная мотивация Netflix на содействие сообществу открытого кода и развитию технологической экосистемы считается тесно связанной с принципами, положенными в основу концепции микроэкономики, известной как *экономия за счет масштабов* (Economies of Scale). Мешенберг продолжил развивать свою мысль: «Мы увлеклись типовыми решениями, работающими на компонентах нашей платформы и средствах автоматизации... Мы воспользовались эффектами экономии за счет масштабов, проявившимися в результате внедрения таких же типовых решений другими пользователями AWS, и продолжим взаимодействие с сообществом по созданию экосистемы».

В начале того, что было названо *облачной эрой*, мы видели: ее первопроходцами не были такие высокотехнологичные компании, как IBM или Microsoft, это были компании, возникшие благодаря развитию Интернета. И Netflix, и Amazon запускались в конце 1990-х годов как интернет-компании. Обе начали с предложения интернет-сервисов, нацеленных конкурировать с противниками, *занимающимися традиционными формами ведения бизнеса*.

Со временем и Netflix, и Amazon превзошли по оценке рыночной стоимости своих конкурентов, *придерживавшихся традиционных форм ведения бизнеса*. При выходе на рынок облачных вычислений Amazon превратила свой коллективный опыт и внутренние инструменты в набор сервисов. Затем, с опорой на сервисы Amazon, то же самое сделала Netflix. Наряду с этим она перевела в разряд средств с открытым кодом все свои наработки и инструменты. Это превратило Netflix в компанию, ориентированную на использование облачных технологий, основанных на сервисах виртуализированной инфраструктуры, предоставляемой средой AWS компании Amazon. Именно так экономия за счет масштабов привела к революции в индустрии облачных вычислений.

В начале 2015 года в отчетах о доходе за первый квартал компания Netflix отрапортовала о размере своей рыночной стоимости 32,9 млрд долларов. В результате этой новой оценки размер рыночной стоимости Netflix впервые превзошел аналогичный показатель сети CBS.

Облачная Java-платформа

В результате перехода в разряд компаний, ориентированных на применение облачных технологий, Netflix предоставила индустрии разработки ПО свой богатый опыт. В этой книге уроки Netflix и проектов с открытым кодом будут применяться в качестве примеров в двух основных темах:

- создание устойчивых распределенных систем благодаря Spring и Netflix OSS;
- использование непрерывной поставки для обеспечения работы приложений, ориентированных на применение облачной среды, с помощью платформы Cloud Foundry.

Первым пунктом в нашем путешествии станет усвоение понятий и концепций, которые будут использоваться во всей книге для описания хода создания и практического применения облачных приложений.

Двенадцать факторов

Методология 12 факторов представляет собой популярный набор принципов создания приложений, составленный создателями облачной платформы Heroku. *Twelve-Factor App* — сайт, изначально созданный Адамом Виггинсом (Adam Wiggins), соучредителем Heroku, в виде манифеста с описанием SaaS-приложений, разработанных с целью получить преимущества от использования приемов, свойственных современным облачным платформам.

На сайте https://12factor.net/ методология начинается с описания набора ключевых основополагающих идей по созданию приложений.

Ранее в данной главе шла речь об обещаниях, которые платформа дает пользователям, создающим приложения. В табл. 1.1 представлен набор идей, точно определяющих ценные предложения по созданию приложений и следующих методологии 12 факторов. Идеи разбиваются еще и на набор требований — на 12 отдельных факторов, выделяющих суть этих ключевых идей в коллекцию советов по способам создания приложений.

Таблица 1.1. Ключевые идеи 12-факторных приложений

Использование **декларативных** форматов для автоматизации установок и настроек в целях сведения к минимуму времени и усилий присоединяющихся к проекту новых разработчиков
Заключение четких соглашений с базовой операционной системой, предусматривающих **максимальную переносимость** между средами выполнения
Обеспечение должного **развертывания** на современных **облачных платформах**, избавляющего от необходимости использовать серверы и системное администрирование
Сведение к минимуму отступлений развертывания от эксплуатации с целью предоставить возможность **непрерывного развертывания**, позволяющего достичь максимальной адаптации к новым условиям
Предоставление возможности **масштабирования** без существенных изменений архитектуры, а также практики применения оснастки или развертывания

Все 12 факторов, перечисленных в табл. 1.2, описывают требования, способствующие созданию приложений, использующих идеи, изложенные в табл. 1.1. Двенадцать факторов — базовый набор требований, которые могут применяться в целях создания приложений, оптимизированных для работы в облачной среде. Поскольку факторы охватывают широкий круг вопросов, касающихся самых

распространенных приемов во всех современных облачных платформах, создание 12-факторных приложений есть общая отправная точка в разработке облачных приложений.

Таблица 1.2. Приемы создания 12-факторных приложений

Кодовая база	Использование одной кодовой базы, отслеживаемой в системе контроля версий при множестве развертываний
Зависимости	Явное объявление и изолирование зависимостей
Конфигурация	Сохранение конфигурации в среде
Вспомогательные сервисы	Отношение к вспомогательным сервисам как к подключенным ресурсам
Сборка, выпуск, практическое применение	Четкое разделение стадий сборки и практического применения
Процессы	Выполнение приложения в виде одного или нескольких процессов, не использующих состояния
Привязка портов	Экспорт сервисов через привязку портов
Многопоточное выполнение	Горизонтальное масштабирование с помощью модели процесса
Утилизируемость	Максимальная надежность за счет быстрого запуска и корректного завершения работы
Функциональная совместимость разработки и практического применения	Максимально возможная схожесть разработки, доводки и практического применения
Ведение регистрационных записей	Отношение к регистрационным записям как к потокам событий
Процессы администрирования	Запуск задач администрирования и управления в виде разовых процессов

Кроме информации на сайте, посвященном 12-факторным приложениям, подробно описывающей каждый из них, дальнейшему раскрытию каждого требования были посвящены целые книги. Методология 12 факторов теперь используется в отдельных прикладных средах, чтобы помочь разработчикам выполнить требования некоторых или даже абсолютно всех 12 факторов.

В данной книге методология 12 факторов будет применяться для описания конкретных особенностей Spring-проектов, реализованных с целью следовать данному стилю разработки приложений. Так что сейчас важно предоставить краткое изложение каждого из этих факторов.

Кодовая база

Одна кодовая база, отслеживаемая в системе контроля версий при множестве развертываний

Репозиториям исходного кода приложения надлежит содержать одно приложение с манифестом зависимостей этого приложения. Приложению не следует требовать перекомпиляции или пакетирования для различных сред. Как показано на рис. 1.2, все уникальное для каждой среды должно содержаться вне кода.

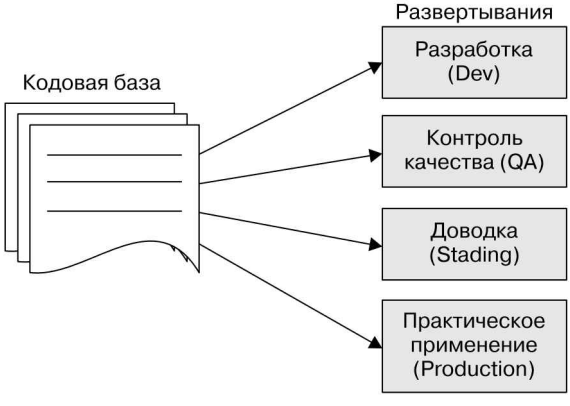


Рис. 1.2. Исходный код создается один раз и развертывается для каждой среды

Зависимости

Явное объявление и изолирование зависимостей

Зависимости приложения необходимо объявить явным образом, и все без исключения зависимости должны быть доступны из репозитория компонентов, который может загружаться с помощью такого диспетчера зависимостей, как Apache Maven.

Двенадцатифакторные приложения никогда не полагаются на существование подразумеваемых общесистемных пакетов, требуемых в качестве зависимостей для запуска приложений. Все зависимости приложения объявляются в явном виде в файле манифеста, в котором приводится четкое описание всех подробностей каждой ссылки.

Конфигурация

Сохранение конфигурации в среде

Код приложения нужно полностью отделить от конфигурации. Последняя должна определяться средой.

Такие настройки приложения, как строки подключения, учетные данные или имена хостов зависимых веб-сервисов, должны храниться в виде переменных среды, чтобы их можно было легко изменить без развертывания конфигурационных файлов.

Как показано на рис. 1.3, любое отклонение в вашем приложении от среды к среде рассматривается в качестве конфигурации среды и должно быть сохранено в среде, а не с приложением.

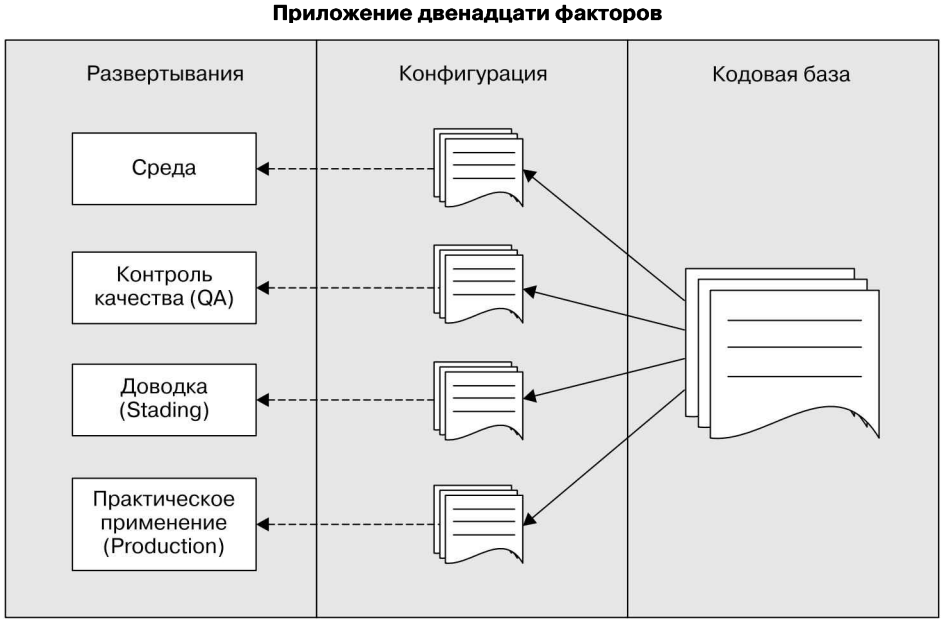


Рис. 1.3. Выделение конфигурации, специфичной для среды, в саму среду

Вспомогательные сервисы

Отношение к вспомогательным сервисам как к подключенным ресурсам

К вспомогательному относится любой сервис, используемый 12-факторным приложением в качестве части его обычной работы. Примерами таких сервисов могут послужить базы данных, веб-сервисы RESTful, управляемые через API, SMTP- или FTP-сервер.

Вспомогательные сервисы считаются *ресурсами* приложения. Эти ресурсы подключены к приложению в течение всего периода его практического применения. Развертывание 12-факторного приложения должно быть рассчитано на замену встроенной базы SQL в среде тестирования внешней базой данных MySQL, размещенной в среде доводки, без внесения изменений в код приложения.

Сборка, выпуск, практическое применение

Четкое разделение стадий сборки и практического применения

В 12-факторном приложении имеется четко выраженное разделение стадий сборки, выпуска и практического применения.

- *Стадия сборки*. Происходит либо компиляция исходного кода приложения, либо его сборка в пакет. Созданный пакет называется сборкой.
- *Стадия выпуска*. Сборка объединяется со своей конфигурацией. Затем выпуск, созданный для развертывания, готов для работы в среде выполнения. Каждый выпуск должен иметь уникальный идентификатор, использующий либо семантическое обозначение версии, либо метку времени. Каждый выпуск нужно добавить к каталогу, который может использоваться инструментами управления выпусками для отката на предыдущий выпуск.
- *Стадия практического применения*. На данной стадии, зачастую называемой *временем выполнения* (runtime), приложение в виде выбранного выпуска запускается в среде выполнения.

Разбиение этой стадии на отдельные процессы позволяет устранить возможность изменения кода приложения во время выполнения. Единственным способом изменить код остается запуск стадии сборки для создания нового выпуска или запуск отката для развертывания предыдущего выпуска.

Процессы

Выполнение приложения в виде одного или нескольких процессов, не использующих состояния

Двенадцатифакторные приложения создаются без расчета на использование состояния и с прицелом на архитектуру *без совместно используемых ресурсов*. Этим приложениям присуща лишь возможная зависимость от вспомогательного сервиса. Примерами таких сервисов, предоставляющих средства сохранения данных, могут послужить базы данных или хранилища объектов. Все ресурсы подключаются к приложению во время выполнения в качестве вспомогательного сервиса. Лакмусовой бумажкой, показывающей, применяет ли приложение состояние, является возможность демонтажа и повторного создания среды выполнения без какой-либо потери данных.

Двенадцатифакторные приложения не сохраняют состояние на локальной файловой системе в среде выполнения.

Привязка портов

Экспорт сервисов через привязку портов

Двенадцатифакторные приложения полностью автономны, то есть им в ходе выполнения для внедрения в среду выполнения с целью создания интернет-сервиса

веб-сервер не требуется. Каждое приложение откроет к себе доступ через порт HTTP, который привязан к приложению в среде выполнения. В ходе развертывания уровень маршрутизации будет обрабатывать поступающие из открытого имени хоста входящие запросы, направляя их к среде выполнения приложения, в привязки HTTP-порта.

Джошу Лонгу, одному из соавторов этой книги, в сообществе Java приписывается изречение: «Make JAR not WAR» («Создавайте не WAR, а JAR»). Джош использует данное выражение, чтобы объяснить, как в сборочном JAR-файле самые новые Spring-приложения могут встраивать такой сервер Java-приложения, как Tomcat.

Многопоточное выполнение

Горизонтальное масштабирование с помощью модели процесса

Приложения должны быть пригодны к масштабированию процессов или потоков для параллельного выполнения работы по требованию. JVM-приложения могут автоматически справляться с оперативным распараллеливанием с помощью нескольких потоков.

Приложениям следует распределять работу параллельно, в зависимости от выполняемых действий. Сегодня большинство прикладных сред JVM имеют соответствующую встроенную возможность. Некоторые сценарии, требующие заданий по обработке данных, которые выполняются в виде продолжительных задач, должны задействовать управляющие программы, способные осуществлять асинхронную диспетчеризацию параллельного функционирования по доступному пулу потоков.

Двенадцатифакторные приложения должны также иметь возможность горизонтального масштабирования и обработки запросов по сбалансированности нагрузки между несколькими идентичными выполняемыми экземплярами приложения. Отказ от использования состояний в конструкции приложений позволяет справляться с более тяжелыми нагрузками с помощью горизонтального масштабирования приложений по нескольким узлам.

Утилизируемость

Максимальная надежность за счет быстрого запуска и корректного завершения работы

Процессы 12-факторных приложений спроектированы с прицелом на утилизируемость. Приложение в ходе выполнения процесса может быть остановлено в любое время и при этом достойно справиться с утилизацией процессов.

Процессы приложения должны максимально сократить время запуска. Приложениям следует запускаться за несколько секунд и приступать к обработке входящих

запросов. Быстрые запуски сокращают время масштабирования экземпляров приложений в ответ на увеличение нагрузки.

Если процессы приложения запускаются долго, то могут ограничить доступность при большом объеме трафика, который способен перегрузить все доступные нормально действующие экземпляры приложения. При уменьшении времени запуска приложений до нескольких секунд его вновь вводимые в общую работу экземпляры получают возможность быстрее реагировать на непредсказуемые резкие скачки трафика без снижения доступности или производительности.

Функциональная совместимость разработки и практического применения

Максимально возможная схожесть разработки, доводки и эксплуатации

Двенадцатифакторные приложения не должны допускать расхождений между средами разработки и применения. Существует три типа расхождений, которые следует иметь в виду.

- *Расхождение по времени.* Разработчики должны рассчитывать на то, что изменения, внесенные в процессе создания, будут быстро введены в эксплуатацию.
- *Расхождения в персонале.* Разработчики, вносящие изменение в код, привлекаются непосредственно к его внедрению в производство и тщательно отслеживают поведение приложения после внесения изменений.
- *Расхождение в инструментарии.* В каждой среде должно быть зеркальное отображение технологии и режимов работы среды, чтобы ограничить возможность неожиданного поведения из-за незначительных несоответствий.

Ведение регистрационных записей

Отношение к регистрационным записям как к потокам событий

Двенадцатифакторные приложения ведут регистрационные записи в виде упорядоченного потока событий в стандартное устройство вывода `stout`. Приложениям не следует пытаться управлять хранилищем собственных регистрационных файлов. Вместо этого сбором и архивированием регистрационных записей для приложения должна заниматься *среда выполнения*.

Процессы администрирования

Запуск задач администрирования и управления в виде разовых процессов

Иногда у разработчиков приложения возникает потребность запустить разовые административные задачи. К таковым могут относиться миграции баз данных или запуск одноразовых сценариев, зарегистрированных в репозитории исходного кода

приложения. Они считаются процессами администрирования. Подобные процессы для поддержания согласованности между средами должны запускаться в среде выполнения с помощью сценариев, зарегистрированных в репозитории.

Резюме

В этой главе рассматривались мотивы, побудившие организации принять конкретные требования и смену архитектуры. Мы постарались ознакомить вас с некоторыми ранее появившимися толковыми идеями, в частности с манифестом 12-факторных приложений.

2 Bootcamp: введение в Spring Boot и Cloud Foundry

В данной главе будет рассмотрен вопрос создания приложений, ориентированных на работу в облаке с помощью Spring Boot и Cloud Foundry.

Что такое Spring Boot

Когда приложение называют *ориентированным на работу в облачной среде*, это значит, что оно написано для успешного выполнения в рабочей среде на основе облака. Среда Spring Boot предоставляет способ создания готовых к использованию Spring-приложений с минимальным временем настройки. Основные цели создания проекта Spring Boot были сконцентрированы на идее, согласно которой пользователи должны иметь возможность быстро освоиться в среде Spring.

В проекте Spring Boot также прослеживается особый взгляд на платформу Spring и библиотеки сторонних разработчиков. Особый взгляд означает, что в Spring Boot излагается набор знакомых абстракций, общий для всех Spring-проектов. Этот особый взгляд предоставляет подводки к тому, что необходимо всем проектам, притом ничем не мешая создателю. Таким образом, Spring Boot упрощает замену компонентов при изменении проектных требований. Он основывается на экосистеме Spring и библиотеках сторонних разработчиков, включая мнения и формирование соглашений, призванных упростить реализацию приложений, готовых к эксплуатации.

Начало работы с проектом Spring Initializr

Spring Initializr представляет собой проект с открытым кодом (https://github.com/spring-io/initializr) и инструмент в экосистеме Spring, помогающий быстро создавать новые приложения Spring Boot. Pivotal запускает экземпляр Spring Initializr, размещенный на сервисах Pivotal Web Services (http://start.spring.io/). Он генерирует

проекты Maven и Gradle с любыми указанными зависимостями, схематический образ Java-класса, служащего точкой входа, и схематический образ блочного теста.

В мире монолитных приложений такая цена способна быть непомерно высокой, но стоимость инициализации можно легко амортизировать на протяжении всей жизни проекта. При переходе на архитектуру, ориентированную на использование облачной среды, понадобится создание все большего и большего количества приложений. Из-за этого разногласия в создании новых приложений в вашей архитектуре должны быть сведены к минимуму. Spring Initializr помогает снизить первоначальную стоимость. Это и веб-приложение, пригодное к применению из вашего браузера, и REST API, который будет создавать для вас новые проекты.

Например, этим инструментом можно воспользоваться для создания нового проекта Spring Boot, запустив команду `curl`:

```
curl http://start.spring.io
```

Результат будет выглядеть примерно так же, как на рис. 2.1.

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

:: Spring Initializr ::  https://start.spring.io

This service generates quickstart projects that can be easily customized.
Possible customizations include a project's dependencies, Java version, and
build system or build structure. See below for further details.

The services uses a HAL based hypermedia format to expose a set of resources
to interact with. If you access this root resource requesting application/json
as media type the response will contain the following links:
+-----------------+-----------------------------------------+
| Rel             | Description                             |
+-----------------+-----------------------------------------+
| gradle-build    | Generate a Gradle build file            |
| gradle-project  | Generate a Gradle based project archive |
| maven-build     | Generate a Maven pom.xml                |
| maven-project * | Generate a Maven based project archive  |
+-----------------+-----------------------------------------+


The URI templates take a set of parameters to customize the result of a request
to the linked resource.
+----------------+-----------------------------------------+-----------------------------+
| Parameter      | Description                             | Default value               |
+----------------+-----------------------------------------+-----------------------------+
| applicationName | application name                        | DemoApplication             |
| artifactId     | project coordinates (infer archive name) | demo                        |
| baseDir        | base directory to create in the archive | no base dir                 |
| bootVersion    | spring boot version                     | 1.4.2.RELEASE               |
| dependencies   | dependency identifiers (comma-separated) | none                        |
| description    | project description                     | Demo project for Spring Boot |
| groupId        | project coordinates                     | com.example                 |
| javaVersion    | language level                          | 1.8                         |
| language       | programming language                    | java                        |
| name           | project name (infer application name)   | demo                        |
| packageName    | root package                            | com.example                 |
| packaging      | project packaging                       | jar                         |
| type           | project type                            | maven-project               |
| version        | project version                         | 0.0.1-SNAPSHOT              |
+----------------+-----------------------------------------+-----------------------------+
```

Рис. 2.1. Взаимодействие с Spring Initializr через REST API

В качестве альтернативы Spring Initializr можно применить из браузера (http://start.spring.io/), как показано на рис. 2.2.

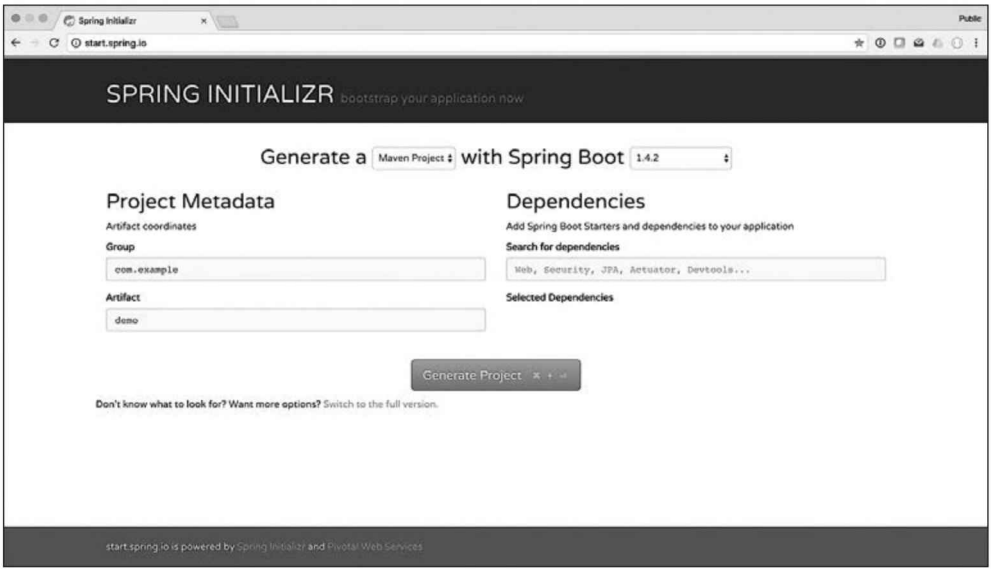


Рис. 2.2. Сайт Spring Initializr

А теперь предположим, что нам нужно создать простой веб-сервис RESTful, который обращается к базе данных SQL (H2). Понадобится привлечь необходимые библиотеки из экосистемы Spring, включая Spring MVC, Spring Data JPA и Spring Data REST (табл. 2.1).

Таблица 2.1. Некоторые примеры Spring Boot Starters для типичного приложения Spring Boot

Проект Spring	Проекты Starter	Maven-идентификатор компонента
Spring Data JPA	JPA	spring-boot-starter-data-jpa
Spring Data REST	REST Repositories	spring-boot-starter-data-rest
Spring Framework (MVC)	Web	spring-boot-starter-web
Spring Security	Security	spring-boot-starter-security
H2 Embedded SQL DB	H2	h2

Чтобы найти эти ингредиенты, можно приступить к набору имени *зависимостей* в поле поиска или щелкнуть на пункте Switch to the full version (Переключиться на полную версию), а затем вручную установить флажки для желаемых зависимостей. Большинство из них называется зависимостями *начального проекта* (Starter Project dependencies).

Начальная зависимость *starter* — тип особой дополнительной библиотеки, которая автоматически вставит набор базовых функций, составленный из других проектов экосистемы Spring, в ваше новое приложение Spring Boot.

При создании приложения Spring Boot с самого начала пришлось бы перемещаться по сети промежуточных зависимостей. Вследствие этого можно столкнуться с проблемой, если у приложения есть зависимости от сторонних библиотек, имеющих конфликтующие версии. Spring Boot обрабатывает конфликтующие промежуточные зависимости, поэтому нужно только указать одну версию родительского проекта Spring Boot. Это затем гарантирует, что любая начальная зависимость Spring Boot от пути к классам вашего приложения будет использовать совместимые версии.

Выполним небольшое упражнение. Обратимся к Spring Initializr и включим начальные зависимости для `Web`, `H2`, `REST Repositories` и `JPA`. Все остальное оставим в качестве установок по умолчанию. Нажмем кнопку Generate Project (Создать проект), и начнется загрузка архива `demo.zip`. Распакуем архив и получим исходный код схематического образа проекта, готовый к импортированию в IDE-среду по вашему выбору (пример 2.1).

Пример 2.1. Содержимое сгенерированного архива приложения Spring Boot после распаковки

```
.
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           └── DemoServiceApplication.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── java
            └── com
                └── example
                    └── DemoServiceApplicationTests.java
```

В примере 2.1 показана структура каталогов сгенерированного приложения. Выступая в качестве части содержимого сгенерированного проекта, инструмент Spring Initializr предоставляет сценарий-упаковщик, либо Gradle wrapper (`gradlew`), либо Maven wrapper (из проекта Maven wrapper) (`mvnw`). Этот упаковщик можно применить для сборки и запуска проекта. При первом запуске упаковщик загружает сконфигурированную версию инструмента сборки. Версии Maven или Gradle контролируются. Это значит следующее: все последующие пользователи

будут иметь воспроизводимую сборку. Риск того, что кто-то попытается провести сборку вашего кода с несовместимой версией Maven или Gradle, отсутствует. Это также существенно упрощает процесс непрерывного развертывания: сборка, используемая для разработки, будет точно такой же, как и сборка, применяемая в среде непрерывной интеграции.

Следующая команда запустит новую установку Maven-проекта с загрузкой и кэшированием зависимостей, указанных в `pom.xml`, и установкой компонента сборки `.jar` в локальное Maven-хранилище (обычно это `$HOME/.m2/repository/*`):

```
$ ./mvnw clean install
```

Чтобы запустить приложение Spring Boot из командной строки, воспользуйтесь предоставляемым дополнительным модулем Spring Boot Maven, автоматически сконфигурированным в сгенерированном файле `pom.xml`:

```
$ ./mvnw spring-boot:run
```

Веб-приложение должно быть успешно запущено и доступно по адресу http://localhost:8080. Но беспокоиться не стоит, ничего особо интересного *пока* не произошло.

Откройте файл проекта `pom.xml` в привычном для вас текстовом редакторе: `emacs`, `vi`, TextMate, Sublime, Atom, Notepad.exe и т. д., подойдет любой из них (пример 2.2).

Пример 2.2. Раздел зависимостей из файла проекта демонстрационного сервиса pom.xml

```
<dependencies>

    ❶
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    ❷
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-rest</artifactId>
    </dependency>

    ❸
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    ❹
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
```

```
            <scope>runtime</scope>
        </dependency>

        ❺
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
</dependencies>
```

❶ Компонент `spring-boot-starter-data-jpa` обеспечивает существующие Java-объекты всем необходимым, используя спецификацию объектно-реляционного отображения — Java ORM (Object Relational Mapping) и JPA (Java Persistence API), и обладает высокой продуктивностью на выходе. Этот компонент включает типы JPA-спецификации, основное подключение на Java к базам данных SQL — Java database connectivity (JDBC) и поддержку JPA для Spring, Hibernate в качестве реализации, Spring Data JPA и Spring Data REST.

❷ Компонент `spring-boot-starter-data-rest` упрощает экспорт работающих с гипермедиа REST-сервисов из определения хранилища Spring Data.

❸ Компонент `spring-boot-starter-web` обеспечивает всем необходимым для сборки REST-приложений со Spring. Он вводит поддержку JSON- и XML-маршализации, выгрузки файлов, встроенный веб-контейнер (по умолчанию используется самая последняя версия Apache Tomcat), поддержку проверочных операций, Servlet API и др. Это дублирующая зависимость, поскольку `spring-boot-starter-data-rest` автоматически введет ее для нас. Здесь она обозначена для прояснения ситуации.

❹ Компонент `h2` является встроенной базой данных SQL, хранящей данные в памяти. Если в путях классов среда Spring Boot обнаружит встроенную базу данных, подобную H2, Derby или HSQL, и увидит, что вы не располагаете где-либо по-иному сконфигурированным источником данных `javax.sql.DataSource`, то сконфигурирует его за вас. Встроенный источник данных `DataSource` заработает при запуске приложения и сам себя уничтожит (вместе со всем своим содержимым!), когда приложение прекратит функционировать.

❺ Компонент `spring-boot-starter-test` вводит все исходные типы, требуемые для написания эффективных имитаторов и комплексных тестов, включая среду тестирования Spring MVC. Предполагается, что поддержка тестирования необходима, и эта зависимость добавляется по умолчанию.

В родительской сборке указываются все версии этих зависимостей. В типовом проекте Spring Boot явно задается только версия для самой среды Spring Boot. Как только становится доступна новая версия последней, укажите в вашей сборке на нее, и с ней будут обновлены все соответствующие библиотеки и включения.

Затем откройте в привычном для вас текстовом редакторе класс точки входа приложения, `demo/src/main/java/com/example/DemoApplication.java`, и вместо него вставьте код примера 2.3.

Пример 2.3. Простейшее приложение Spring Boot с JPA-классом Cat

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

❶
@SpringBootApplication
public class DemoApplication {

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

❸
@Entity
class Cat {

 @Id
 @GeneratedValue
 private Long id;

 private String name;

 Cat() {
 }

 public Cat(String name) {
   this.name = name;
 }

 @Override
 public String toString() {
   return "Cat{" + "id=" + id + ", name='" + name + '\'' + '}';
 }

 public Long getId() {
   return id;
 }

 public String getName() {
   return name;
 }
}

❹
@RepositoryRestResource
```

```
interface CatRepository extends JpaRepository<Cat, Long> {
}
```

1. Выполнение аннотации класса в качестве приложения Spring Boot.
2. Запуск приложения Spring Boot.
3. Обычный JPA-элемент для моделирования элемента `Cat`.
4. Хранилище Spring Data JPA (в котором обрабатываются все общие операции создания, чтения, обновления и удаления), экспортированное в качестве REST API.

Данный код *должен* функционировать, но убедиться в этом у нас не получится, пока мы его не протестируем! При наличии тестов можно установить контрольное рабочее состояние для разрабатываемого ПО, а затем усовершенствовать его на основе исходного уровня качества. Так что откройте класс тестирования, `demo/src/test/java/com/example/DemoApplicationTests.java`, и замените его кодом из примера 2.4.

Пример 2.4. Комплексный тест для нашего приложения

```
package com.example;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;

//@formatter:off
import org.springframework.boot.test.autoconfigure.web
  .servlet.AutoConfigureMockMvc;
//@formatter:on
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import java.util.stream.Stream;

import static org.junit.Assert.assertTrue;
//@formatter:off
import static org.springframework.test.web.servlet
  .request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet
  .result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet
  .result.MockMvcResultMatchers.status;
//@formatter:on

❶
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureMockMvc
```

```
public class DemoApplicationTests {

 ❷
 @Autowired
 private MockMvc mvc;

 ❸
 @Autowired
 private CatRepository catRepository;

 ❹
 @Before
 public void before() throws Exception {
  Stream.of("Felix", "Garfield", "Whiskers").forEach(
   n -> catRepository.save(new Cat(n)));
 }

 ❺
 @Test
 public void catsReflectedInRead() throws Exception {
  MediaType halJson = MediaType
   .parseMediaType("application/hal+json;charset=UTF-8");
  this.mvc
   .perform(get("/cats"))
   .andExpect(status().isOk())
   .andExpect(content().contentType(halJson))
   .andExpect(
    mvcResult -> {
     String contentAsString = mvcResult.getResponse().getContentAsString();
     assertTrue(contentAsString.split("totalElements")[1].split(":")[1].trim()
      .split(",")[0].equals("3"));
    });
  }
}
```

❶ Это блочный тест, использующий исполнитель тестов среды Spring. Кроме того, мы настроили его на успешное взаимодействие с инструментарием тестирования Spring Boot, справляющимся с имитационным веб-приложением.

❷ Вставка клиента `MockMvc` теста Spring MVC, из которого выполняются вызовы к конечным точкам REST.

❸ Здесь можно сослаться на любые другие управляемые компоненты в контексте нашего Spring-приложения, включая `CatRepository`.

❹ Установка в базу данных информационного образца.

❺ Вызов конечной точки HTTP GET для ресурса `/cats`.

Более подробно вопросы тестирования будут рассмотрены в главе 4.

Запустите приложение и поработайте с базой данных через HAL-закодированный гипермедиа-REST API по адресу http://locahost:8080/cats. Если запустить тест, то он должен быть пройден!

Прежде чем приступить к созданию приложений, ориентированных на функционирование в облаке с помощью Spring Boot, нужно настроить среду разработки.

Начало работы со Spring Tool Suite

До сих пор все наши действия выполнялись с использованием текстового редактора, но большинство инженеров в наше время применяют IDE-среду. В этой области есть ряд вполне подходящих средств, и Spring Boot хорошо работает с любым из них. Если у вас еще нет Java IDE, то обратите внимание на Spring Tool Suite (STS) (http://spring.io/tools). Данная среда является IDE, основанной на Eclipse, и в ней собран общий инструментарий экосистемы, позволяющий не тратить время на его загрузку с сайта Eclipse (http://www.eclipse.org/). STS имеется в свободном доступе в соответствии с условиями Eclipse Public License.

STS позволяет получить при работе с Spring Initializr такие же ощущения, как и при обычном взаимодействии с IDE-средой. Фактически функциональные возможности, имеющиеся в Spring Tool Suite, IntelliJ Ultimate edition, NetBeans и в самом веб-приложении Spring Initializr, делегируются REST API, имеющемуся в Spring Initializr, поэтому вы получаете одинаковый результат, независимо от того, откуда начнете.

Для написания приложений Spring или Spring Boot *не нужно* использовать некую конкретную IDE-среду. Авторы без каких-либо проблем создавали Spring-приложения в emacs, обычной среде Eclipse, Apache Netbeans (как с поддержкой, так и без поддержки замечательного дополнительного модуля Spring Boot, в создании которого поучаствовали специалисты такой серьезной компании, как Oracle), и IntelliJ IDEA Community edition и Ultimate edition. Для Spring Boot 1.x, в частности, требуется Java 6 и выше, а также поддержка редактирования обычных файлов `.properties`, как и поддержка работы со сборками на основе Maven или Gradle. Для этого подойдут любые IDE-среды начиная с 2010 года и выше.

При использовании Eclipse у Spring Tool Suite появляется множество приятных особенностей, которые придают дополнительные удобства работе с проектами на основе Spring Boot.

- В STS IDE можно получить доступ ко всем справочникам по Spring.
- IDE-среда позволяет создавать новые проекты с помощью Spring Initializr.
- При попытке обращения к типу, которого нет в путях к классам (classpath), но который может быть обнаружен в `starter`-зависимостях Spring Boot, STS автоматически добавит этот тип.
- Boot Dashboard позволяет беспрепятственно редактировать локальные приложения Spring Boot, сохраняя их синхронизацию с развертыванием Cloud

Foundry. Дальнейшую отладку и перезагрузку развернутых приложений Cloud Foundry можно проводить, не выходя из IDE.

- STS облегчает редактирование файлов Spring Boot с расширениями `.properties` или `.yml`, предлагая, кроме всего прочего, функцию автозавершения.
- Spring Tool Suite представляет собой стабильное, интегрированное издание Eclipse, выпущенное вскоре после выхода основного издания Eclipse.

Установка Spring Tool Suite (STS)

Загрузите и установите Spring Tool Suite (STS) (http://spring.io/):

- перейдите на https://spring.io/tools/sts;
- выберите пункт Download STS (Скачать STS);
- загрузите, распакуйте и запустите STS.

После выполнения последнего пункта вам будет предложено выбрать расположение рабочей области. Выберите желаемый вариант и нажмите кнопку OK. Если планируется применение той же области при каждом запуске STS, то щелкните на пункте Use this as the default and do not ask again (Использовать по умолчанию и больше не спрашивать). После определения расположения рабочей области и нажатия кнопки OK среда STS IDE будет загружена в первый раз (рис. 2.3).

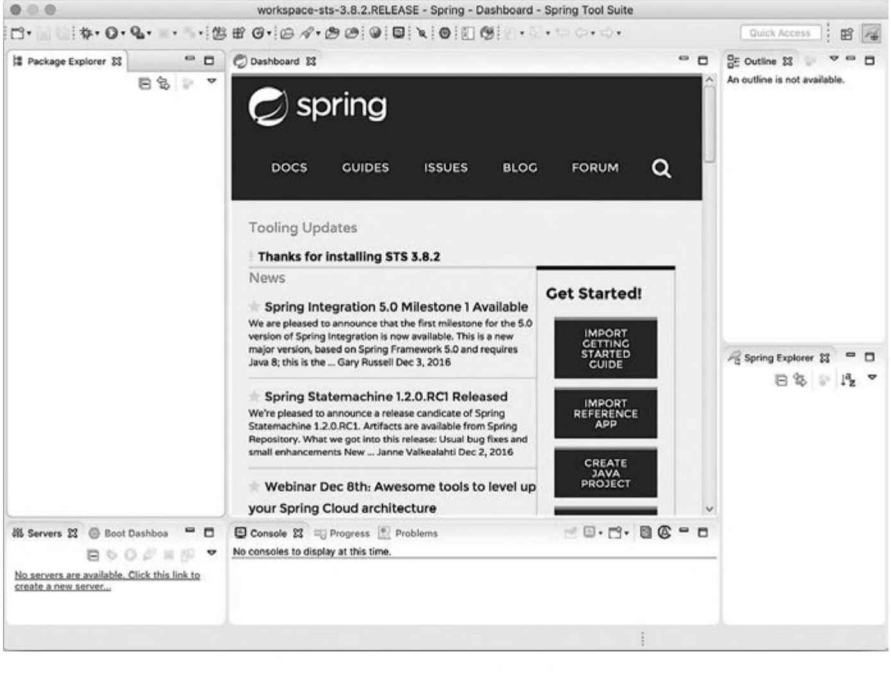


Рис. 2.3. Панель управления STS

Есть также пакеты для многочисленных операционных систем. Например, если применяется Homebrew Cask на OS X или macOS Sierra, то можно воспользоваться разделом репозитория Pivotal tap (https://github.com/pivotal/homebrew-tap), а затем запустить команду brew cask install sts.

Создание нового проекта с помощью Spring Initializr

Только что созданный пример можно импортировать из Spring Initializr напрямую, для чего нужно перейти по пунктам File ▸ Import ▸ Maven (Файл ▸ Импорт ▸ Maven), а затем в корневом каталоге имеющегося проекта указать на необходимость импортирования файла `pom.xml`. Однако вместо этого для создания нашего первого приложения Spring Boot задействуем STS. Мы собираемся с помощью проекта Spring Boot Starter разработать простой веб-сервис Hello World. Чтобы создать новое приложение Spring Boot, используя проект Spring Boot Starter, выберите из меню, как показано на рис. 2.4, пункты File ▸ New ▸ Spring Starter Project (Файл ▸ Новый ▸ Spring Starter Project).

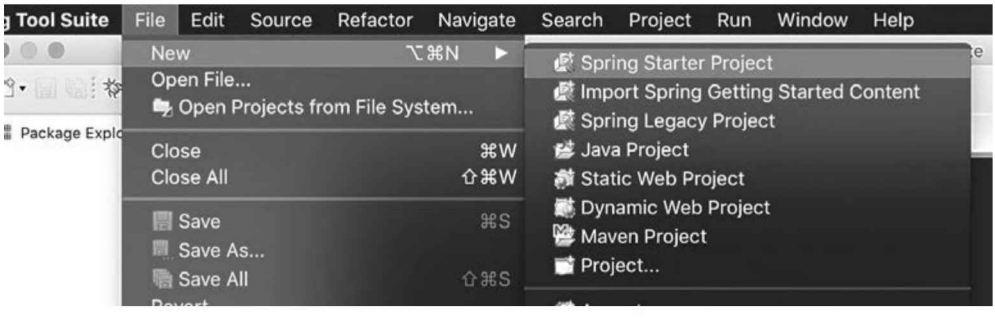


Рис. 2.4. Создание нового проекта Spring Boot Starter

После выбора пункта, позволяющего создать новый проект Spring Boot Starter, появится диалоговое окно, предназначенное для настройки вашего нового приложения Spring Boot (рис. 2.5).

Можно, конечно, выбрать свои варианты, но, чтобы не усложнять нашу легкую прогулку, воспользуемся исходными значениями и щелкнем на кнопке Next (Далее). После этого будет показан набор проектов Spring Boot Starter, из которого можно выбрать некие элементы для вашего нового приложения Spring Boot. Для нашего первого приложения мы собираемся выбрать Web (рис. 2.6.).

Нажмите кнопку Finish (Завершить). После этого ваше приложение Spring Boot будет создано и импортировано для вас в рабочее пространство IDE-среды, став видимым в проводнике пакетов (Package Explorer).

Раскройте, если это еще не сделано, узел `demo [boot]` в Package Explorer, и просмотрите содержимое пакета, как показано на рис. 2.7.

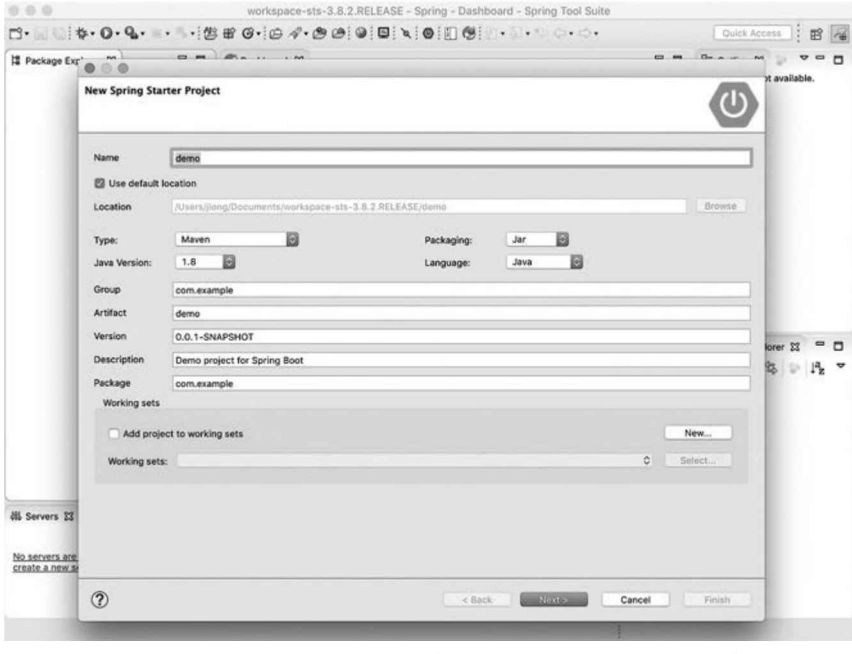


Рис. 2.5. Настройка вашего нового проекта Spring Boot Starter

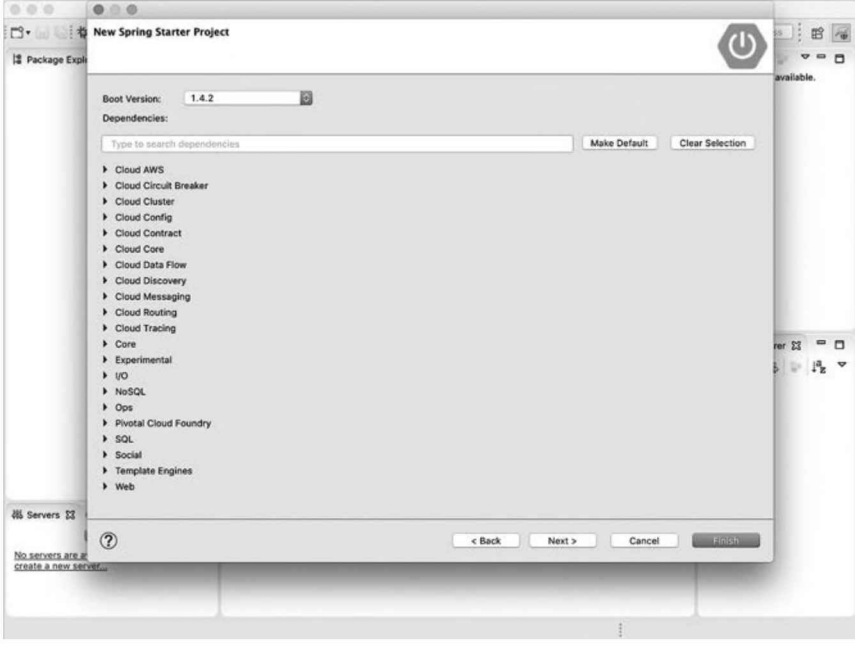


Рис. 2.6. Выберите тип вашего проекта Spring Boot Start

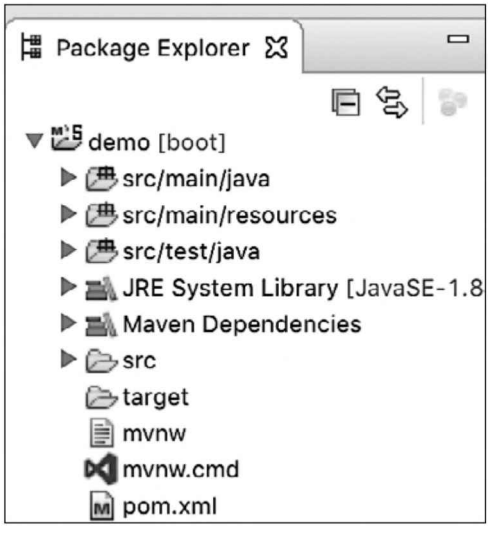


Рис. 2.7. Раскройте в проводнике пакетов проект demo

Из раскрытых файлов проекта перейдите к файлу `src/main/java/com/example/DemoApplication.java`. И, не останавливаясь на этом, запустите приложение, воспользовавшись пунктами меню Run ▸ Run (Пуск ▸ Пуск) (рис. 2.8).

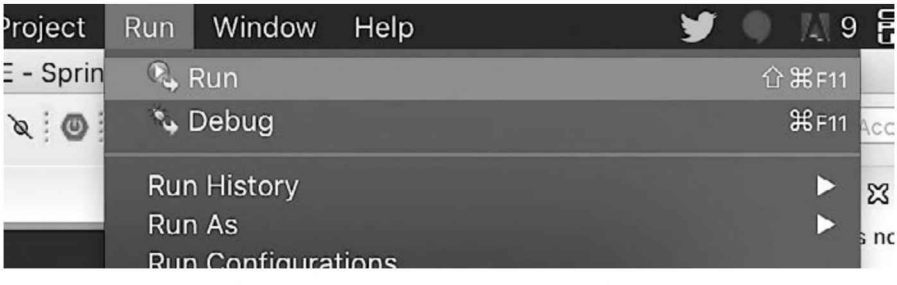


Рис. 2.8. Запустите приложение Spring Boot

После выбора из меню пункта Run (Пуск) появится диалоговое окно Run As (Запустить как). Выберите пункт Spring Boot App (Приложение Spring Boot), после чего нажмите кнопку OK (рис. 2.9).

Теперь ваше приложение Spring Boot запустится (рис. 2.10). Если посмотреть на консоль STS, то там после запуска будет значок Spring Boot, составленный из ASCII-символов, и версия Spring Boot. Там же можно увидеть вывод регистрационных записей данного приложения. Вы увидите, что был запущен встроенный сервер Tomcat и этот запуск произошел с привязкой к порту `8080`, используемому по умолчанию. К вашему веб-сервису Spring Boot можно обратиться с адреса http://localhost:8080.

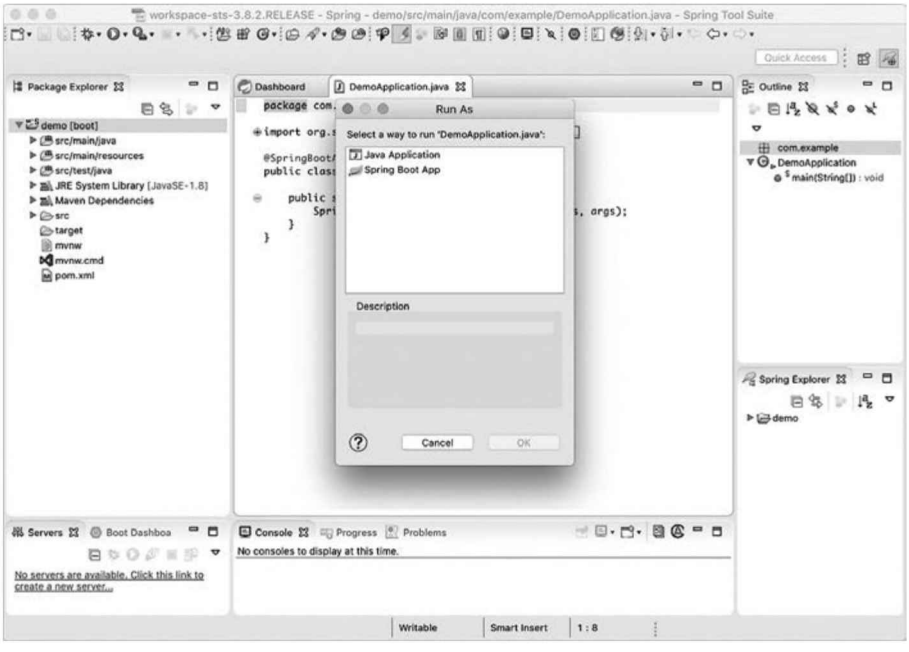


Рис. 2.9. Выберите Spring Boot App и запустите приложение

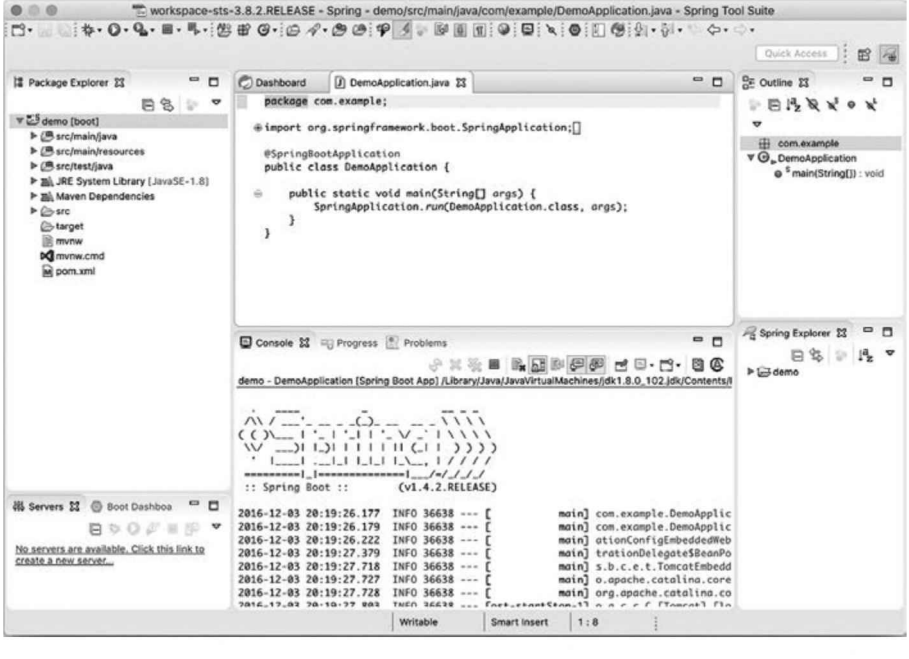


Рис. 2.10. Посмотрите на вывод регистрационных записей Spring Boot в консоли STS

Вот и все! Вы только что создали свое первое приложение Spring Boot, воспользовавшись Spring Tool Suite.

Руководства по Spring

Руководства по Spring (https://spring.io/guides) представляют собой набор небольших, конкретизированных введений во всевозможные темы, относящиеся к проектам Spring. Существует множество руководств, большинство из которых написано специалистами из команды Spring, а к отдельным из них приложили руку партнеры по экосистеме. В каждом из руководств по Spring используются одинаковые подходы с целью предоставить всеобъемлющее и в то же время практичное справочное руководство, которое можно было бы изучить за 15–30 минут. Эти руководства — одни из наиболее полезных ресурсов для тех, кто приступает к работе со Spring Boot.

Как показано на рис. 2.11, на сайте Spring Guides предоставляется целая коллекция постоянно поддерживаемых примеров, нацеленных на конкретные варианты использования.

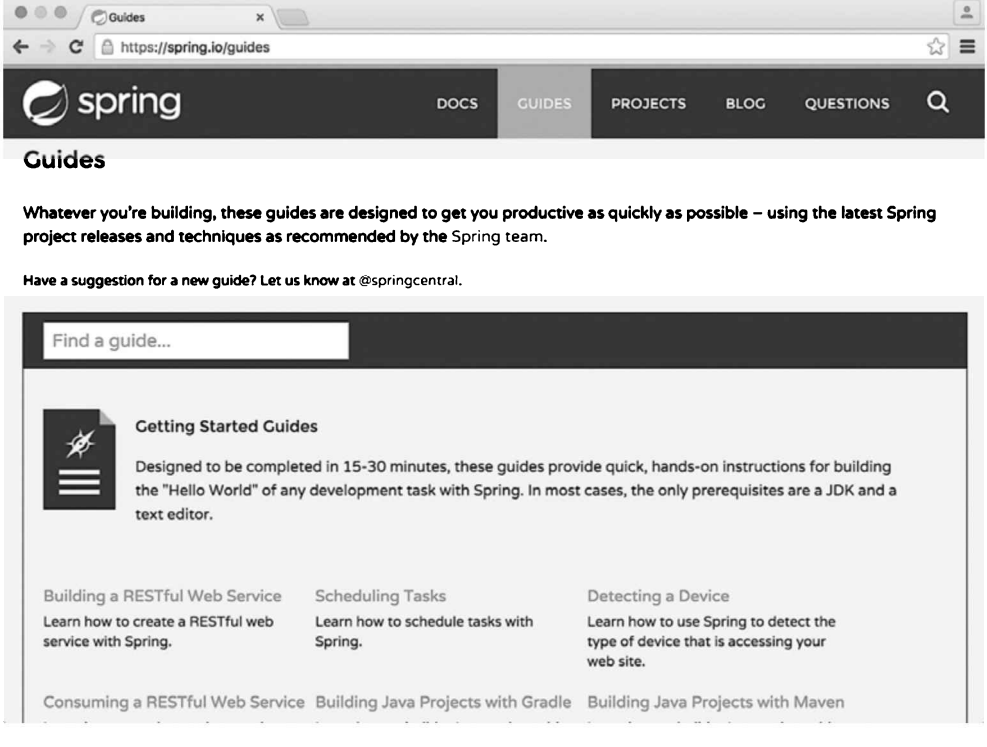


Рис. 2.11. Сайт Spring Guides

На рис. 2.12 был введен поисковый запрос `spring boot`, сужающий список руководств, имеющих отношение к Spring Boot.

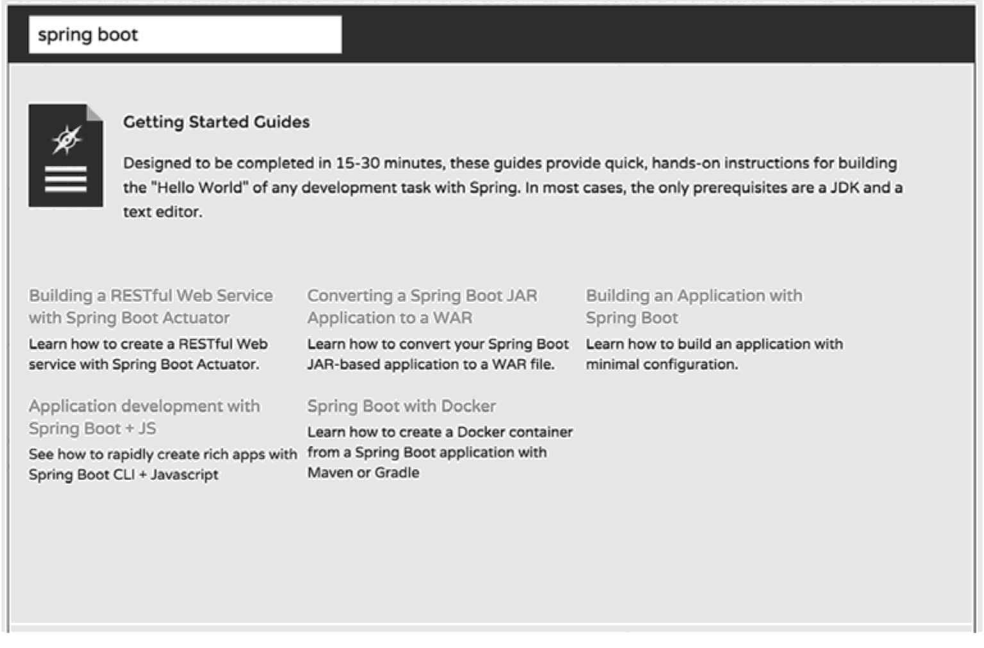


Рис. 2.12. Просмотр руководств по Spring

В качестве части каждого руководства по Spring будут встречаться уже знакомые разделы:

- Getting started (С чего начать);
- Table of contents (Содержимое);
- What you'll build (Что будет создаваться);
- What you'll need (Что понадобится);
- How to complete this guide (Порядок изучения руководства);
- Walkthrough (Пошаговая инструкция);
- Summary (Резюме);
- Get the code (Получение кода);
- Link to GitHub repository (Ссылка на репозиторий GitHub).

А теперь выберем одно из основных руководств по Spring Boot под названием Building an Application with Spring Boot (Создание приложения с помощью Spring Boot) (https://spring.io/guides/gs/spring-boot/).

На рис. 2.13 показана основная структура руководства по Spring. Каждое руководство структурировано узнаваемым образом, чтобы помочь усвоить содержимое наиболее рациональным способом. В каждом руководстве имеется ссылка на репозиторий GitHub, включающий в себя три папки: `complete`, `initial` и `test`. В папке `complete` хранится конечный вариант работоспособного примера, по которому можно проверить результат усилий. В папке `initial` содержится схематическая модель, почти пустая файловая система, пригодная для обхода рутинного кода и концентрации внимания на том, что является уникальным для данного руководства. В папке `test` имеется все необходимое для подтверждения работоспособности всего проекта.

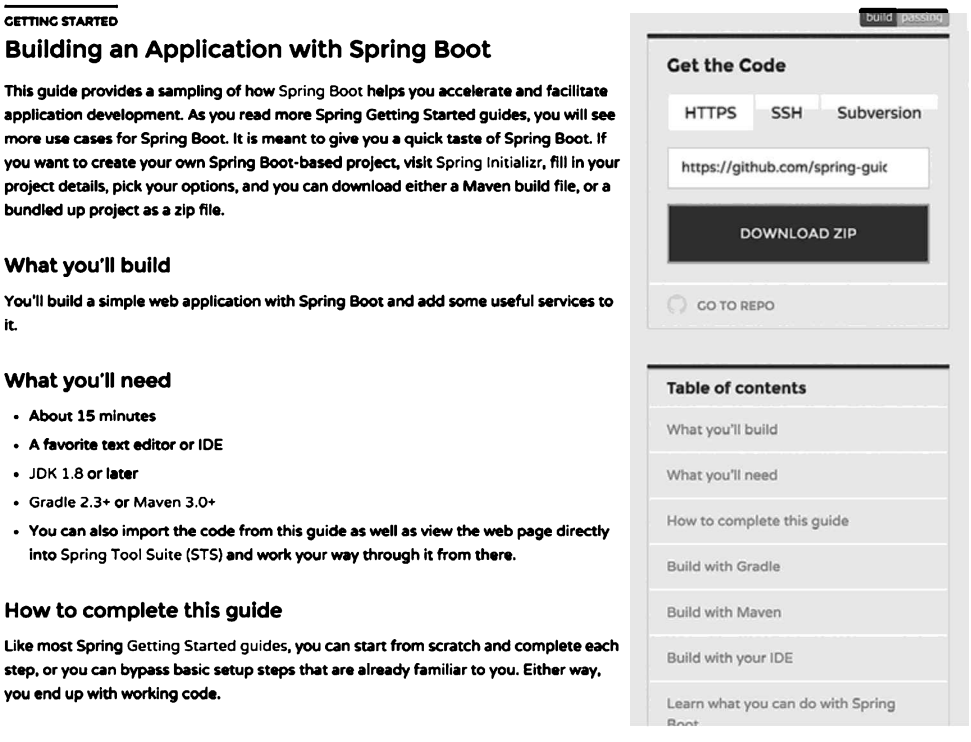


Рис. 2.13. Создание приложения с помощью руководства по Spring Boot

В книге будут встречаться ситуации или сценарии, которые могут потребовать использования расширенного справочного руководства, чтобы помочь вам лучше понять материал. В таких случаях рекомендуется обращаться к руководствам по Spring и найти то, которое будет отвечать вашим потребностям наилучшим образом.

Следование руководствам в STS. При использовании Spring Tool Suite можно следовать рекомендациям руководств прямо из IDE-среды. Для этого нужно пройти по пунктам меню File ▸ New ▸ Import Spring Getting Started Content (Файл ▸ Новый ▸ Импортировать содержимое Spring Getting Started) (рис. 2.14).

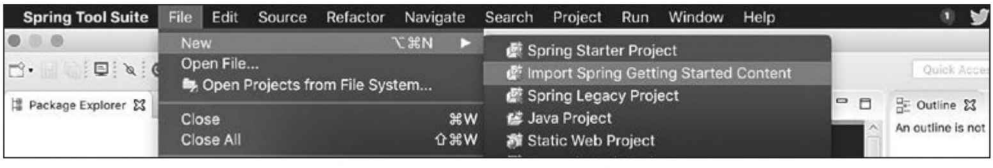


Рис. 2.14. Импортирование содержимого Spring Getting Started

В вашем распоряжении окажется точно такой же каталог руководств, что и на сайте spring.io/guides (рис. 2.15).

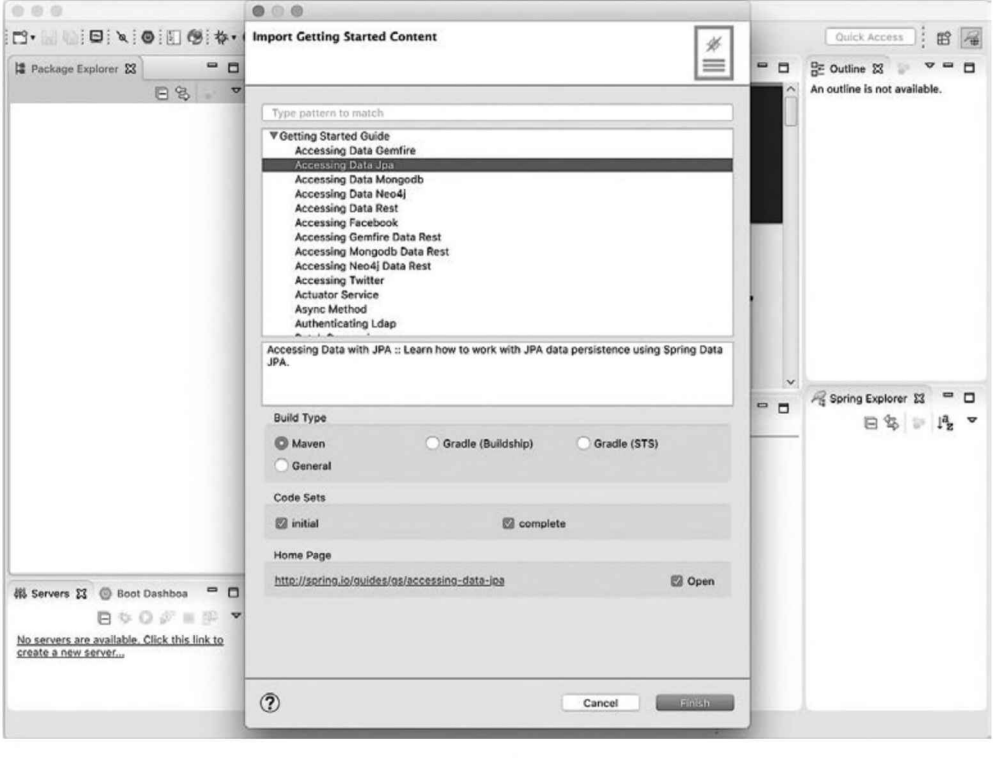


Рис. 2.15. Выбор руководства из диалогового окна Import Getting Started Content (Импортировать содержимое Getting Started)

Выберите руководство, и оно отобразится в вашей IDE-среде, как и соответствующий код (рис. 2.16).

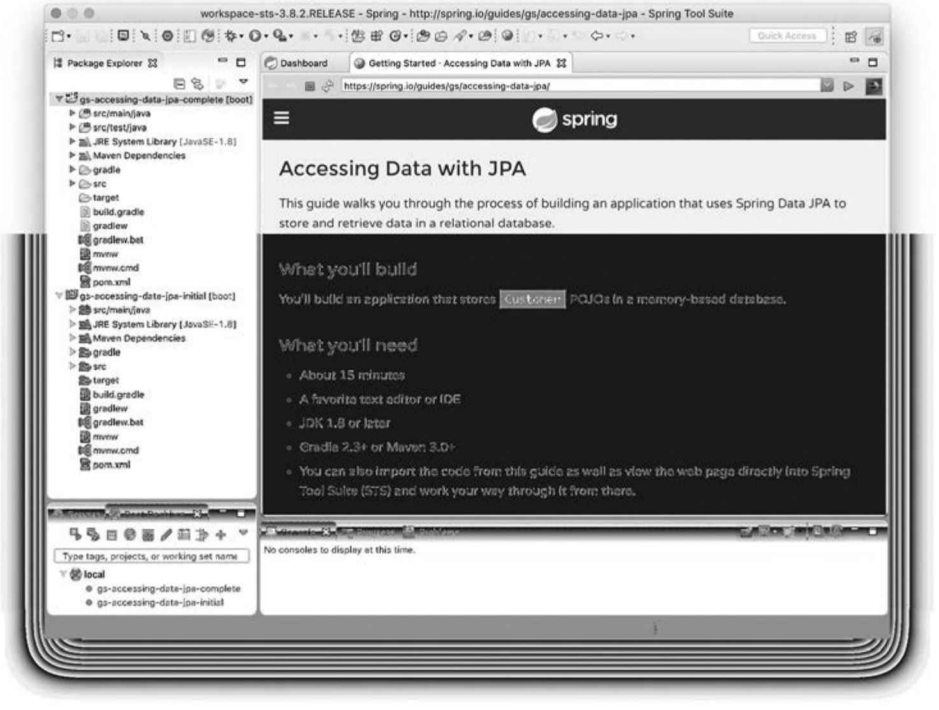


Рис. 2.16. IDE-среда загружает руководство и соответствующие ему Git-проекты

Конфигурация

По сути, приложение Spring — коллекция объектов. Spring управляет ими и их взаимоотношениями в ваших интересах, предоставляя по мере необходимости сервисы этим объектам. Среда Spring сможет поддерживать ваши объекты в виде библиотечных компонентов (*bean*-компонентов), если будет осведомлена о них. В среде Spring предоставляется несколько различных (дополнительных) способов описания ваших компонентов.

Предположим, у нас имеется типичный многоуровневый сервис с объектом сервиса, который, в свою очередь, обращается к `javax.sql.DataSource`, чтобы обменяться данными (в этом конкретном случае) со встроенной базой данных, то есть с H2. Нам необходимо определить сервис. Ему понадобится источник данных (datasource). Мы можем создать его экземпляр в требующем его месте, но затем придется продублировать эту же логику там, где нужно будет повторно воспользоваться источником данных в других объектах. Получение и инициализация ресурса происходят на месте вызова, следовательно, мы не можем где-либо применить эту же логику еще раз. Пример 2.5 показывает, что встроенный экземпляр `DataSource` позже способен стать источником проблемы.

Пример 2.5. Приложение, в котором экземпляр DataSource создается в классе, что фактически не позволяет рассчитывать на совместное использование определения

```
package com.example.raai;

import com.example.Customer;
import org.springframework.core.io.ClassPathResource;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.jdbc.datasource.init.DataSourceInitializer;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;
import org.springframework.util.Assert;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

public class CustomerService {

 private final DataSource dataSource = new EmbeddedDatabaseBuilder()
  .setName("customers").setType(EmbeddedDatabaseType.H2).build();

 public static void main(String argsp[]) throws Throwable {
  CustomerService customerService = new CustomerService();

  ❶
  DataSource dataSource = customerService.dataSource;
  DataSourceInitializer init = new DataSourceInitializer();
  init.setDataSource(dataSource);
  ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
  populator.setScripts(new ClassPathResource("schema.sql"),
   new ClassPathResource("data.sql"));
  init.setDatabasePopulator(populator);
  init.afterPropertiesSet();

  ❷
  int size = customerService.findAll().size();
  Assert.isTrue(size == 2);

 }

 public Collection<Customer> findAll() {
  List<Customer> customerList = new ArrayList<>();
  try {
   try (Connection c = dataSource.getConnection()) {
    Statement statement = c.createStatement();
    try (ResultSet rs = statement.executeQuery("select * from CUSTOMERS")) {
```

```
    while (rs.next()) {
     customerList.add(new Customer(rs.getLong("ID"), rs.getString("EMAIL")));
    }
   }
  }
 }
 catch (SQLException e) {
  throw new RuntimeException(e);
 }
return customerList;
}
}
```

❶ Трудно выставить источник данных (datasource), поскольку единственная ссылка на него скрыта в закрытом неизменяемом поле (private final field) в самом классе `CustomerService`. Единственный способ получения доступа к этой переменной заключается в использовании имеющегося в Java *«дружественного»* доступа (friend access), где экземпляры заданного объекта способны видеть другие экземпляры, а также закрытые переменные.

❷ Здесь применяются типы из самой среды Spring, а не из JUnit, поскольку библиотека JUnit для «отработки» компонента использует типы, относящиеся к области видимости `test`. Класс `Assert` среды Spring поддерживает проектирование по контракту, а не блочное тестирование!

Поскольку мы не в состоянии подключить имитатор источника данных (mock datasource) и выставить источник данных каким-либо иным образом, мы вынуждены встроить *тестовый* код в сам компонент. Провести тестирование надлежащим образом *будет нелегко*, следовательно, в пути к классам для нашего основного кода понадобятся `test`-зависимости на период проверки.

Мы задействуем встроенный источник данных, который будет одинаковым как для разработки, так и для эксплуатации. В реальном примере конфигурация особенностей, относящихся к среде окружения, таких как имена пользователей и хостов, будет параметризирована; в противном случае при любой попытке выполнения кода будет задействоваться источник данных, предназначенный для практического использования!

Рациональнее было бы централизовать определение bean-компонентов, вынеся его за пределы тех мест вызова, где оно применяется. Каким образом код нашего компонента получит доступ к этой централизованной ссылке? Мы можем сохранить ее в статических переменных, но как тогда проводить тестирование, при условии, что статические ссылки будут разбросаны по всему коду? Как *имитировать* ссылку? Ссылки можно сохранить в каком-либо совместно используемом контексте, подобном интерфейсу доступа к сервисам имен и каталогов — JNDI (Java Naming and Directory Interface), но тогда мы столкнемся с похожей проблемой: возникнут сложности с тестированием такой схемы без имитации всего интерфейса JNDI!

Вместо того чтобы закрывать логику инициализации ресурсов и их получения во всех потребителях этих ресурсов, можно создать объекты и установить их подключение в одном месте: в *отдельном классе*. Данный принцип называется *инверсией управления* — Inversion of Control (IoC).

Подключение объектов отделяется от самих компонентов. Благодаря этому разделению появляется возможность создавать код компонентов, имеющий зависимость от базовых типов и интерфейсов и не содержащий привязки к конкретной реализации. Такая операция называется *внедрением зависимостей*. Компонент, не знающий о том, как и где создана конкретная зависимость, проигнорирует тот факт, что в ходе блочного тестирования она будет представлена поддельным (*имитационным*) объектом.

Переместим подключение объектов — конфигурацию — в отдельный класс, то есть в класс *конфигурации*. А теперь посмотрим в примере 2.6, как это поддерживается имеющейся в Spring конфигурацией Java.

А как насчет XML? Среда Spring дебютировала с поддержкой конфигурации на основе XML. Такая конфигурация предлагает множество преимуществ, сходных с конфигурацией Java: она является централизованным компонентом, отделенным от подключаемых компонентов. Она по-прежнему поддерживается, но не очень подходит приложениям Spring Boot, зависящим от конфигурации Java. В этой книге конфигурация на основе XML использоваться не будет.

Пример 2.6. Класс конфигурации, позволяющий удалить определения bean-компонентов из мест их вызова

```
package com.example.javaconfig;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

import javax.sql.DataSource;

❶
@Configuration
public class ApplicationConfiguration {

 ❷
 @Bean(destroyMethod = "shutdown")
 DataSource dataSource() {
  return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2)
   .setName("customers").build();
 }

 ❸
 @Bean
```

```
 CustomerService customerService(DataSource dataSource) {
  return new CustomerService(dataSource);
 }
}
```

❶ Данный класс относится к Spring-классу `@Configuration`, который сообщает среде Spring, что она может ожидать обнаружения определений объектов и порядка их взаимного подключения в этом классе.

❷ Определение `DataSource` будет извлечено и помещено в определение bean-компонента. Это позволит увидеть его и работать с ним любому другому элементу Spring, и это единственный экземпляр `DataSource`.

Если от `DataSource` *зависят* десять компонентов Spring, то у них у *всех* будет исходный доступ к одному и тому же экземпляру в памяти. Это связано с понятием области видимости среды Spring. По умолчанию bean-компонент Spring имеет *одноэлементную область видимости*.

❸ Регистрация экземпляра `CustomerService` со Spring и принуждение Spring к соответствию требованиям `DataSource` с помощью сквозного анализа других зарегистрированных библиотечных модулей в контексте приложения и нахождения тех, чей тип соответствует параметру поставщика bean-компонента.

Вернитесь к `CustomerService` и удалите явно указанную логику создания `DataSource` (пример 2.7).

Пример 2.7. Очищенный код CustomerService, из которого убрана логика инициализации и получения ресурсов

```
package com.example.javaconfig;

import com.example.Customer;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

public class CustomerService {

 private final DataSource dataSource;

 ❶
 public CustomerService(DataSource dataSource) {
  this.dataSource = dataSource;
 }

 public Collection<Customer> findAll() {
```

```
  List<Customer> customerList = new ArrayList<>();
  try {
   try (Connection c = dataSource.getConnection()) {
    Statement statement = c.createStatement();
    try (ResultSet rs = statement.executeQuery("select * from CUSTOMERS")) {
     while (rs.next()) {
      customerList.add(new Customer(rs.getLong("ID"), rs.getString("EMAIL")));
     }
    }
   }
  }
  catch (SQLException e) {
   throw new RuntimeException(e);
  }
  return customerList;
 }
}
```

❶ Определение типа `CustomerService` существенно упростилось, поскольку теперь он зависит от `DataSource`. Мы сняли с себя часть ответственности, возможно, чтобы больше думать об области видимости этого типа: о взаимодействии с `dataSource`, а не об определении самого `dataSource`.

Конфигурация задается явно, но имеет слегка избыточный характер. Если среде Spring позволить, то она способна создать ряд конструкций! В конце концов, зачем нам брать на себя все трудности? Можно же воспользоваться стереотипными аннотациями Spring, чтобы пометить собственные компоненты и позволить Spring создать их экземпляры на основе соглашения.

Вернемся к классу `ApplicationConfiguration` и позволим Spring обнаружить наши стереотипные компоненты, используя *сканирование компонентов* (пример 2.8). Теперь уже не нужно составлять явные описания относительно конструкции компонента `CustomerService`, поэтому его определение также подвергнется удалению. Тип `CustomerService` будет точно таким же, как и раньше, за исключением того, что к нему применится аннотация `@Component`.

Пример 2.8. Выделение конфигурации DataSource в отдельный конфигурационный класс

```
package com.example.componentscan;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

import javax.sql.DataSource;

@Configuration
@ComponentScan
```

```
❶
public class ApplicationConfiguration {

 @Bean(destroyMethod = "shutdown")
 DataSource dataSource() {
  return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2)
   .setName("customers").build();
 }

}
```

❶ Эта аннотация указывает среде Spring на необходимость обнаружения других bean-компонентов в контексте приложения путем сканирования текущего пакета (или того, что находится ниже) и поиска всех объектов, аннотированных с помощью *стереотипных* аннотаций наподобие `@Component`. Эта аннотация и другие, которые сами аннотируются благодаря `@Component`, действуют в качестве своеобразного маркера для Spring, сходного с тегами. Spring различает их по компонентам и создает новый экземпляр объекта, к которому они применяются. По умолчанию они вызывают конструктор без аргументов или же конструктор с параметрами, если все параметры удовлетворяют ссылкам на другие объекты в контексте приложения. Spring предоставляет множество сервисов в виде *присоединенных* аннотаций, ожидаемых в классе `@Configuration`.

В коде нашего примера источник данных (datasource) используется напрямую, и для получения простого результата мы вынуждены набирать массу низкоуровневого шаблонного JDBC-кода. Внедрение зависимостей представляется весьма эффективным инструментом, но это наименее интересный аспект среды Spring. Воспользуемся одной из наиболее привлекательных особенностей Spring — портируемыми абстракциями сервисов — для упрощения взаимодействия с источником данных. Мы заменим наш соответствующий руководству и пространный JDBC-код, применив вместо него шаблон `JdbcTemplate` среды Spring (пример 2.9).

Пример 2.9. Использование JdbcTemplate вместо низкоуровневых вызовов JDBC API

```
package com.example.psa;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

import javax.sql.DataSource;

@Configuration
@ComponentScan
public class ApplicationConfiguration {

 @Bean(destroyMethod = "shutdown")
```

```
 DataSource dataSource() {
  return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2)
   .setName("customers").build();
 }

 ❶
 @Bean
 JdbcTemplate jdbcTemplate(DataSource dataSource) {
  return new JdbcTemplate(dataSource);
 }
}
```

❶ Шаблон `JdbcTemplate` — одна из реализаций Spring-экосистемы шаблонных образцов. Он предоставляет удобные и полезные методы, делающие общение с JDBC простым и обыденным. Он управляет инициализацией и получением ресурсов, уничтожением, обработкой исключений и др., позволяя сконцентрироваться на самой сути решаемой задачи.

Благодаря `JdbcTemplate` пересмотренный код `CustomerService` становится *намного* чище (пример 2.10).

Пример 2.10. Значительно упрощенный код CustomerService

```
package com.example.psa;

import com.example.Customer;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Component;

import java.util.Collection;

@Component
public class CustomerService {

 private final JdbcTemplate jdbcTemplate;

 public CustomerService(JdbcTemplate jdbcTemplate) {
  this.jdbcTemplate = jdbcTemplate;
 }

 public Collection<Customer> findAll() {
  ❶
  RowMapper<Customer> rowMapper = (rs, i) -> new Customer(rs.getLong("ID"),
   rs.getString("EMAIL"));
  ❷
  return this.jdbcTemplate.query("select * from CUSTOMERS ", rowMapper);
 }
}
```

❶ Существует множество перезагружаемых вариантов метода `query`, одним из которых ожидается реализация `RowMapper`. Это объект обратного вызова, который

среда Spring будет вызывать в отношении каждого возвращаемого результата, позволяя отображать объекты, поступившие из базы данных, на объект области вашей системы. Интерфейс `RowMapper` также отлично подходит для лямбда-выражений Java 8!

❷ Метод `query` — обычный метод, умещающийся в одной строке. Так уже гораздо лучше!

Поскольку управление подключением происходит централизованно, то конфигурация bean-компонентов позволяет выполнить *подстановку* (или *внедрение*) реализаций с различными специализациями или свойствами. При желании можно изменить все внедренные реализации для поддержки сквозной функциональности, не меняя получателей bean-компонентов. Предположим, что нужно регистрировать время, затрачиваемое на вызов всех методов. Можно создать класс, являющийся подклассом существующего `CustomerService`, а затем в переопределении метода вставить функцию регистрации до и после вызова реализации в родительском классе. Данная функция является сквозной функциональностью, но для внедрения ее в поведение иерархии объектов придется переопределить все методы.

В идеале нам не придется проходить столь много стадий, чтобы вставить обычные сквозные функциональные возможности поверх таких объектов. Языки, подобные Java, в которых поддерживается только единичное наследование, не предоставляют четкого способа рассмотрения данного варианта использования для любого произвольного объекта. В среде Spring поддерживается альтернативный вариант: *аспектно-ориентированное программирование* — aspect-oriented programming (AOP). Это более обширная тема, нежели Spring, но последняя предоставляет весьма доступный поднабор AOP для Spring-объектов. Имеющаяся в Spring поддержка AOP сосредотачивается вокруг понятия аспекта, кодирующего сквозное поведение. *Элемент среза* (pointcut) дает описание шаблона, соответствие которому должно соблюдаться при использовании аспекта. Шаблон в элементе среза — часть полноценного языка срезов, поддерживаемого средой Spring. Язык срезов позволяет описывать вызовы методов для объектов в приложении Spring. Предположим, что нужно создать аспект, соответствующий всем вызовам методов в нашем примере `CustomerService` сегодня и завтра, и вставить регистрирование для фиксации отметок времени.

Для активации имеющейся в Spring AOP-функциональности добавьте `@EnableAspectJAutoProxy` к классу `ApplicationConfiguration` `@Configuration`. После чего останется только выделить сквозную функциональность в отдельный тип, в `@Aspect`-аннотированный объект (пример 2.11).

Пример 2.11. Выделение сквозной функциональности в аспект Spring AOP

```
package com.example.aop;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
```

```
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

import java.time.LocalDateTime;

@Component
@Aspect
❶
public class LoggingAroundAspect {
 private Log log = LogFactory.getLog(getClass());

 ❷
 @Around("execution(* com.example.aop.CustomerService.*(..))")
 public Object log(ProceedingJoinPoint joinPoint) throws Throwable {
  LocalDateTime start = LocalDateTime.now();

  Throwable toThrow = null;
  Object returnValue = null;

  ❸
  try {
   returnValue = joinPoint.proceed();
  }
  catch (Throwable t) {
   toThrow = t;
  }
  LocalDateTime stop = LocalDateTime.now();

  log.info("starting @ " + start.toString());
  log.info("finishing @ " + stop.toString() + " with duration "
   + stop.minusNanos(start.getNano()).getNano());

  ❹
  if (null != toThrow)
   throw toThrow;

  ❺
  return returnValue;
 }
}
```

❶ Пометка этого bean-компонента в качестве аспекта.

❷ Объявление о предоставлении данному методу возможности выполниться в режиме охвата, то есть до и после выполнения любого метода, соответствующего выражению среза в аннотации `@Around`. Существует множество других аннотаций, но на данный момент именно эта придает нашему коду высокую эффективность. Кроме того, можно исследовать имеющуюся в Spring поддержку AspectJ.

❸ При вызове метода, соответствующего срезу, сначала вызывается наш аспект и передается `ProceedingJoinPoint`, являющемуся дескриптором вызова запущенного метода. Можно выбрать опрос выполнения метода, его продолжение,

его пропуск и т. д. Этот аспект выполняет регистрационную запись до и после того, как продолжит вызов метода.

4. При выдаче исключения оно перехватывается и чуть позже выдается повторно.
5. Если записывается возвращаемое значение, то оно также и возвращается (при условии, что не было выдано исключение).

В случае необходимости AOP можно использовать напрямую, но многие из действительно значимых сквозных функций, с большей долей вероятности встречающихся при создании обычного приложения, уже извлечены в самой среде Spring. Образцом способно послужить декларативное управление транзакциями. В нашем примере имеется один метод, предназначенный только для чтения. Если бы нужно было ввести еще один обслуживающий бизнес метод, который при одном вызове сервиса вносил бы изменения в базу данных несколько раз, то пришлось бы обеспечить внесение данных изменений в едином *рабочем блоке*. При этом либо каждое взаимодействие с ресурсом, имеющим состояние (с источником данных), завершалось бы успехом, либо ни одно из взаимодействий не признавалось бы успешным.

Нам бы не хотелось оставлять систему в противоречивом состоянии. Это идеальный пример сквозной функциональности: AOP послужит для начала блока транзакций перед каждым вызовом метода в бизнес-сервисе и для фиксации (или отката) данной транзакции по завершении вызова. *Можно* сделать именно так, но, к счастью, имеющаяся в Spring поддержка декларативных транзакций уже делает это за нас; не придется создавать низкоуровневый код, выстраиваемый вокруг AOP, чтобы получать работоспособный сеанс транзакций. Добавьте `@EnableTransactionManagement` к классу конфигурации, а затем опишите границы транзакций в бизнес-сервисе, применив аннотацию `@Transactional`.

У нас имеется сервисное звено, по логике, следующим шагом может стать создание веб-приложения. Создать конечную точку REST поможет Spring MVC. Для этого понадобится задать конфигурацию самой среды Spring MVC, развернуть ее на сервлет-совместимом сервере приложений, а затем сконфигурировать взаимодействие сервера приложений с Servlet API. Прежде чем сделать следующий шаг и реализовать весьма скромное рабочее веб-приложение и конечную точку REST, может понадобиться выполнить довольно много действий!

Здесь мы использовали стандарт JDBC, но могли выбрать и применение уровня ORM. Но тогда все еще больше усложнилось бы. По отдельности, при пошаговой разработке, все это не слишком сложно, но по совокупности когнитивная нагрузка может стать чрезмерной.

И здесь на сцену выходит среда Spring Boot и ее автоматическое конфигурирование.

В первом примере данной главы создавалось приложение с интерфейсом, JPA-элемент, несколько аннотаций, класс точки входа `public static void main` и... все! Среда Spring Boot все запустила, и вуаля! Это был действующий REST API, запускаемый на http://localhost:8080/cats. Приложение поддерживало работу с JPA-элементами

с помощью хранилища на основе Spring Data JPA. Делалось многое, для чего, казалось бы, не было указано никакого явного кода, — *просто волшебство*!

Если вы уже знаете многое о среде Spring, то, без сомнения, сможете распознать, что, кроме всего прочего, мы задействовали среду Spring и имеющуюся в ней надежную поддержку JPA. Spring Data JPA используется для конфигурирования декларативного хранилища, основанного на интерфейсе. Spring MVC и Spring Data REST применяются для обслуживания REST API на основе HTTP, но даже при этом вас может заинтересовать, откуда берется сам веб-сервер. Каждому из упомянутых модулей требуется *некая* конфигурация. Обычно не слишком объемная, но, конечно же, более пространная, чем та, которую мы здесь создавали! По крайней мере им требуется аннотация для *выбора* определенного поведения по умолчанию.

Исторически сложилось так, что для среды Spring была выбрана конфигурация. В ее *плоскости* лежит возможность уточнить поведение приложения. Приоритет в Spring Boot — предоставление разумного поведения по умолчанию и поддержка несложных переопределений. Это полномасштабное использование соглашения по конфигурации. По сути, автоматическое конфигурирование, имеющееся в Spring Boot, есть конфигурация того же порядка, что и конфигурация, создаваемая вручную, с применением тех же аннотаций и тех же bean-компонентов, которые вы можете зарегистрировать, а также с разумными настройками по умолчанию.

В Spring поддерживается понятие *загрузчика сервисов* для поддержки регистрации пользовательских вкладов в приложение без изменения самой среды Spring. Загрузчик сервисов — отображение типов на имена классов конфигурации Java, которые затем анализируются и позже становятся доступными для приложения Spring. Регистрация пользовательских вкладов выполняется в файле `META-INF/spring.factories`. Среда Spring Boot заглядывает в файл `spring.factories`, чтобы, кроме всего прочего, найти все классы ниже записи `org.springframework.boot.autoconfigure.EnableAutoConfiguration`. В файле `spring-boot-autoconfigure.jar`, поставляемом вместе со средой Spring Boot, имеются *десятки* различных классов конфигураций, и Spring Boot будет пытаться проанализировать все эти классы! Данная среда попробует как минимум проанализировать все имеющиеся классы, но *безуспешно* вследствие использования различных условий, если хотите, защитников, помещенных в классы конфигурации и в находящиеся в них определения `@Bean`. Этот инструмент условной регистрации компонентов Spring происходит из самой среды Spring, на основе которой построена Spring Boot. Данные условия весьма разнообразны: они тестируют доступность bean-компонентов заданного типа, наличие свойств среды, доступность конкретных типов в путях к классам и др.

Рассмотрим пример `CustomerService`. Нам нужно создать версию приложения Spring Boot, использующую встроенную базу данных и шаблон фреймворка Spring `JdbcTemplate`, а затем поддержку создания веб-приложения. Все необходимое будет сделано средой Spring Boot. Вернемся к приложению `ApplicationConfiguration` (пример 2.12) и превратим его в соответствующее приложение Spring Boot.

Пример 2.12. Класс ApplicationConfiguration.java

```
package com.example.boot;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ApplicationConfiguration {
 ❶
}
```

❶ А конфигурации-то и нет! Spring Boot внесет все то, что было внесено нами вручную, а также многое другое.

Применим контроллер на основе среды Spring MVC, чтобы показать конечную точку REST для ответа на HTTP GET-запросы на `/customers` (пример 2.13).

Пример 2.13. Класс CustomerRestController.java

```
package com.example.boot;

import com.example.Customer;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collection;

❶
@RestController
public class CustomerRestController {

 private final CustomerService customerService;

 public CustomerRestController(CustomerService customerService) {
  this.customerService = customerService;
 }

 ❷
 @GetMapping("/customers")
 public Collection<Customer> readAll() {
  return this.customerService.findAll();
 }
}
```

❶ `@RestController` — еще одна стереотипная аннотация, подобная `@Component`. Она сообщает среде Spring, что этот компонент предназначен для работы в качестве контроллера REST.

❷ Можно воспользоваться аннотациями Spring MVC, приводящими к отображению на область того, что мы пытаемся сделать: в данном случае приведением HTTP GET-запросов к конкретной конечной точке с помощью `@GetMapping`.

Точку входа можно сделать в классе `ApplicationConfiguration` (пример 2.14), затем запустить приложение и перейти в браузере на http://localhost:8080/customers. Разобраться в происходящем в бизнес-логике будет гораздо проще, и мы добились большего с меньшими усилиями!

Пример 2.14. Создание конечной точки REST с помощью Spring MVC

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

Мы получили больше пользы при меньших затратах. Нам известно, что где-то нечто другое создает точно такую же конфигурацию, которую мы сделали ранее. Где создается шаблон `JdbcTemplate`? В классе автоматической конфигурации по имени `JdbcTemplateAutoConfiguration`, чье определение показано в примере 2.15 (в слегка упрощенном виде).

Пример 2.15. JdbcTemplateAutoConfiguration

```
@Configuration     ❶
@ConditionalOnClass({ DataSource.class, JdbcTemplate.class })     ❷
@ConditionalOnSingleCandidate(DataSource.class)     ❸
@AutoConfigureAfter(DataSourceAutoConfiguration.class)     ❹
public class JdbcTemplateAutoConfiguration {

        private final DataSource dataSource;

        public JdbcTemplateAutoConfiguration(DataSource dataSource) {
                this.dataSource = dataSource;
        }

        @Bean
        @Primary
        @ConditionalOnMissingBean(JdbcOperations.class)  ❺
        public JdbcTemplate jdbcTemplate() {
                return new JdbcTemplate(this.dataSource);
        }

        @Bean
        @Primary
        @ConditionalOnMissingBean(NamedParameterJdbcOperations.class)  ❻
        public NamedParameterJdbcTemplate namedParameterJdbcTemplate() {
                return new NamedParameterJdbcTemplate(this.dataSource);
        }
}
```

❶ Это обычный класс `@Configuration`.

❷ Класс конфигурации должен быть проанализирован *только* в том случае, если тип `DataSource.class` и `JdbcTemplate.class` находятся где-то в пути к классам; в противном случае, без сомнения, будет выдана ошибка наподобие `ClassNotFoundException`.

❸ Этот класс конфигурации понадобится, только если где-то в контексте приложения будет внесен bean-компонент `DataSource`.

❹ Мы знаем, что если `DataSourceAutoConfiguration` разрешить, то будет внесен источник данных H2, так что данная аннотация гарантирует введение этой конфигурации *после* запуска `DataSourceAutoConfiguration`. В случае ввода базы данных будет проанализирован текущий класс конфигурации.

❺ Нам нужно внести `JdbcTemplate`, но только при условии, что пользователи (то есть мы с вами) еще не определили bean-компонент того же типа в наших собственных классах конфигурации.

❻ Нам нужно внести `NamedParameterJdbcTemplate`, но только при условии, что такого шаблона еще нет.

Автоматическая конфигурация Spring Boot значительно уменьшает объем фактического кода и когнитивную нагрузку, связанную с этим кодом. Она освобождает нас, позволяя сосредоточиться на сути бизнес-логики, перекладывая всю рутину на среду. В случае необходимости контролировать какой-либо аспект набора компонентов мы имеем все возможности внесения bean-компонентов конкретных типов, и они будут подключены к среде. Spring Boot — реализация принципа открыто-замкнутой среды: она открыта для расширения, но закрыта для изменения.

Переопределение части машины *не требует* перекомпиляции Spring или Spring Boot. Можно понять, что ваше приложение Spring Boot демонстрирует странное поведение, которое вам хочется переопределить или перенастроить. Важно знать, как это сделать. Укажите при запуске приложения ключ `--Ddebug=true`, и Spring Boot выдаст распечатку отчета об отладке — Debug Report, показывая все вычисленные условия и то, где у них были положительные или отрицательные соответствия. Пользуясь отчетом, нетрудно будет проанализировать происходящее в соответствующем классе автоматической конфигурации, чтобы прояснить его поведение.

Платформа Cloud Foundry

Spring Boot позволяет сконцентрироваться на сути самого приложения, но что хорошего мы получим от новоприобретенной производительности, если при этом не сможем добиться ввода приложения в производство? Эксплуатация программы — весьма непростая задача, но от ее решения никуда не деться. Одна из целей непрерывного развертывания нашего приложения в среде, имитирующей рабочую, заключается в проведении проверочных объединенных тестов и приемочных испытаний на работоспособность в условиях его развертывания в среде реальной эксплуатации. Очень важно получать изделие как можно раньше и чаще, поскольку существует только одно место, где заказчик, то есть сторона, наиболее заинтересованная в конечном результате деятельности команды, может убедиться в его работоспособности.

Если переход к эксплуатации станет затруднительным, то команда разработчиков неизбежно станет опасаться данного процесса и обретет неуверенность в своих действиях, увеличивая разрыв между созданием и производством. При этом увеличивается отставание в работе, результаты которой не переведены в плоскость реального использования, что означает возрастание риска каждой подвижки, поскольку в каждом выпуске становится больше бизнес-составляющей. Чтобы снизить риск ввода в эксплуатацию, нужно сократить объем действий и увеличить частоту развертывания. Если в таком применении выявится ошибка, то доработка и ее развертывание должны быть как можно менее затратными.

Замысел заключается в автоматизации всего, что может быть автоматизировано в цепочке получения желаемого результата, от управления программным продуктом до его запуска в производство, что не добавляет в процесс практической ценности. Развертывание не относится к видоизменениям в бизнесе, оно не добавляет результата к процессу. Оно должно быть полностью автоматизировано. Повышение производительности достигается за счет автоматизации.

Cloud Foundry — облачная платформа, предназначенная для оказания помощи. Это платформа в виде сервиса (Platform as a Service, PaaS). В Cloud Foundry основное внимание уделяется не жестким дискам, не оперативной памяти, не центральному процессору, не установке Linux и не доработкам, закрывающим бреши безопасности, предлагаемым в инфраструктуре в виде сервиса — Infrastructure as a Service (IaaS), и не контейнерам, которые можно получить в контейнерах в виде сервиса, а приложениям и их сервисам. Специалисты сосредотачиваются на приложениях и их сервисах и ни на чем другом. В данной книге Cloud Foundry будет использоваться повсеместно, так что сейчас обратимся к развертыванию поэтапно разрабатываемого приложения Spring Boot, рассматривая наряду с этим поддержку конфигурации в среде Spring.

Существует несколько реализаций Cloud Foundry, каждая из которых основана на открытом коде Cloud Foundry. Все приложения, рассматриваемые в данной книге, будут запускаться и развертываться в Pivotal Web Services (http://run.pivotal.io/). Эта платформа предлагает поднабор функций, предоставляемых компанией Pivotal в устанавливаемой на площадке заказчика (on-premise) системе Pivotal Cloud Foundry (https://pivotal.io/platform). Она базируется на Amazon Web Services, в регионе AWS East region. PWS поможет сделать первые шаги в работе с Cloud Foundry, поскольку является вполне возможным и допустимым вариантом, обслуживающим как небольшие, так и крупные проекты и сопровождаемым командой поддержки компании Pivotal.

Перейдите на главную страницу. Там будут поля входа в систему по существующей учетной записи или регистрации новой учетной записи (рис. 2.17).

После входа в систему появляется личный кабинет, и можно получать информацию о развернутых приложениях (рис. 2.18).

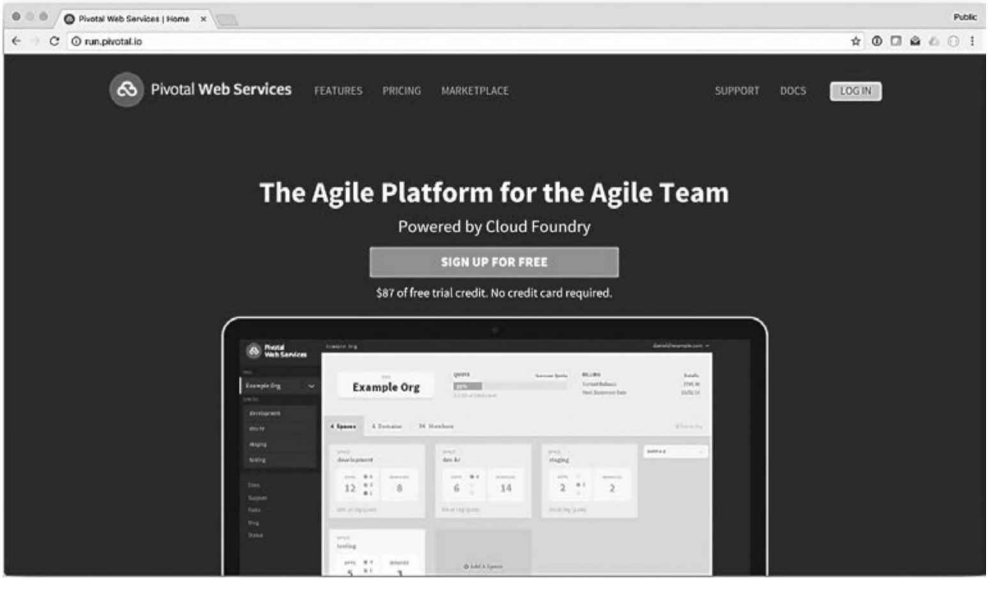


Рис. 2.17. Главная страница PWS

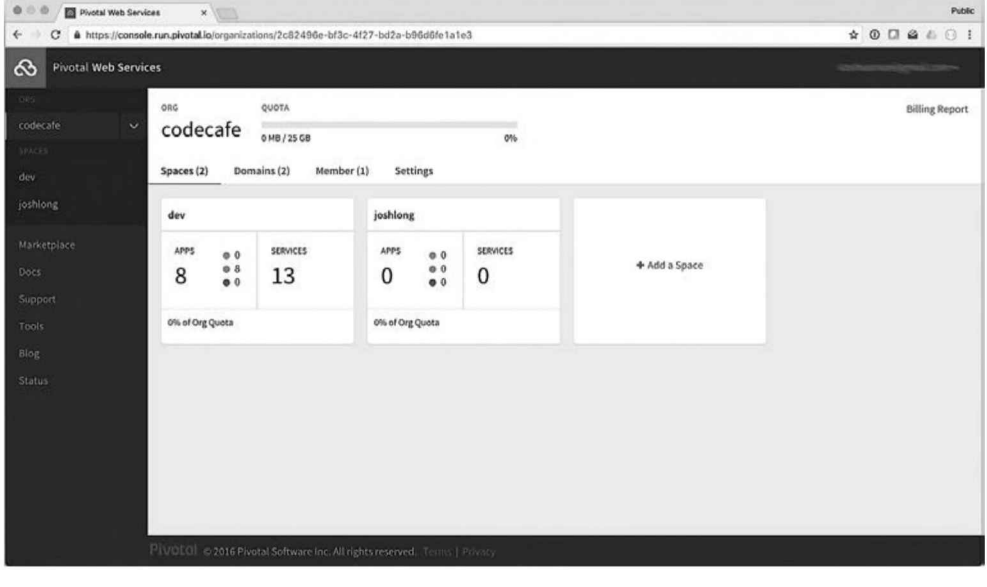


Рис. 2.18. Консоль PWS

Получив возможность работать в личном кабинете, следует убедиться в наличии интерфейса командной строки — cf command-line interface (CLI) (https://docs.cloudfoundry.org/cf-cli/install-go-cli.html). Если его у вас еще не было, нужно зарегистрироваться.

Задайте соответствующий экземпляр Cloud Foundry, применив команду `cf target api.run.pivotal.io`, и затем зарегистрируйтесь, воспользовавшись командой `cf login` (пример 2.16).

Пример 2.16. Аутентификация и определение организации и пространства Cloud Foundry

```
→ cf login

API endpoint: https://api.run.pivotal.io

Email> email@gmail.com

Password>
Authenticating...
OK

Select an org (or press enter to skip):
1. marketing
2. sales
3. back-office

Org> 1
Targeted org back-office

Targeted space development

API endpoint:  https://api.run.pivotal.io (API version: 2.65.0)
User:          email@gmail.com
Org:           back-office          ❶
Space:         development          ❷
```

❶ В Cloud Foundry может быть множество организаций, к которым данный пользователь имеет доступ.

❷ В рамках данной организации может быть несколько сред (например, разработки — `development`, доводки — `staging` и объединения — `integration`).

Теперь, имея действительный личный кабинет и войдя в него, можно развернуть приложение в `target/configuration.jar`. Необходимо предоставить базу данных, используя команду `cf create-service` (пример 2.17).

Пример 2.17. Создание базы данных MySQL, ее привязка к нашему приложению и после этого запуск приложения

```
cf create-service p-mysql 100mb bootcamp-customers-mysql          ❶
cf push -p target/configuration.jar bootcamp-customers \
 --random-route --no-start                                        ❷

cf bind-service bootcamp-customers bootcamp-customers-mysql       ❸

cf start bootcamp-customers                                       ❹
```

❶ Сначала нужно предоставить базу данных MySQL из сервиса MySQL (который называется `p-mysql`) по плану `100mb`. Здесь ей присваивается логическое

имя `bootcamp-customers-mysql`. Для перечисления других сервисов в каталоге сервисов экземпляра Cloud Foundry следует воспользоваться командой `cf marketplace`.

❷ После этого в Cloud Foundry помещается компонент приложения `target/configuration.jar`. Мы назначим приложение и произвольный маршрут — random route (URL, в данном случае под доменом PWS `cfapps.io`), но пока запускать его не собираемся: ему по-прежнему нужна база данных!

❸ База данных есть, но к ней нельзя обратиться, пока она не *привязана* к приложению. По сути, привязка заключается в выставлении в приложении переменных среды с соответствующей информацией в них о подключении.

❹ После привязки приложения к базе данных его наконец-то можно запустить!

Добавляя приложение к Cloud Foundry, вы предоставляете его как приложение в двоичном коде в виде файла с расширением `.jar`, а не виртуальной машины или Linux-контейнера (хотя последний ему *можно* предоставить). При получении файла `.jar` платформа Cloud Foundry пытается определить природу приложения. К чему оно относится, к приложению Java? К приложению Ruby? К .NET-приложению? И в конечном итоге платформа останавливается на приложении Java. И файл `.jar` передается пакету Java *buildpack*. Этот пакет — каталог, наполненный сценариями, вызываемыми в соответствии с известным функциональным циклом. Можно указать URL и переопределить таким образом исходный пакет buildpack, или же можно позволить запуститься именно этому исходному пакету. Существуют buildpack-пакеты для всех разновидностей языков и платформ, включая Java, .NET, Node.js, Go, Python, Ruby и др. Java buildpack определит, что приложение представляет собой выполняемый метод `main(String [] args)` и поэтому является самодостаточным. Будет задействована самая последняя версия OpenJDK и указано, что наше приложение должно запускаться. Вся эта конфигурация запакована в Linux-контейнер, который затем диспетчер Cloud Foundry развернет в кластере. В одной среде Cloud Foundry может выполнять сотни тысяч контейнеров.

Приложение моментально будет развернуто, и на консоль выведется сообщение с URL, где было открыто приложение. Наши поздравления! Теперь приложение развернуто в экземпляре Cloud Foundry.

В приложении имеется один bean-компонент источника данных, в соответствии с чем Cloud Foundry автоматически заново отобразит его на единственный привязанный сервис MySQL, заменив исходное встроенное определение источника данных H2 определением, указывающим на источник данных MySQL.

Существует множество аспектов развернутого приложения, которым можно дать описание при каждом развертывании. В первом запуске для конфигурации приложения мы воспользовались различными «заклинаниями» `cf`, но вскоре такие ухищрения могут превратиться в весьма утомительное занятие. Вместо этого оформим конфигурацию приложения в файле-манифесте Cloud Foundry, обычно называемом `manifest.yml`.

Файл `manifest.yml` для приложения, который будет работать при условии наличия источника данных MySQL, получившего указанное ранее имя, получит следующий вид (пример 2.18).

Пример 2.18. Манифест Cloud Foundry

```
---
applications:
- name: bootcamp-customers                                          ❶
  buildpack: https://github.com/cloudfoundry/java-buildpack.git     ❷
  instances: 1
  random-route: true
  path: target/spring-configuration.jar                             ❸
  services:
    - bootcamp-customers-mysql                                      ❹
  env:
    DEBUG: "true"
    SPRING_PROFILES_ACTIVE: cloud                                   ❺
```

❶ Мы предоставили логическое имя приложения...

❷ ...задали пакет buildpack...

❸ ...указали, какой двоичный код используется...

❹ ...определили зависимость от предоставляемых Cloud Foundry сервисов...

❺ ...и указали переменные среды для переопределения свойств, на которые будет реагировать среда Spring Boot. Запись `--Ddebug=true` (или `DEBUG: true`) переписывает условия в автоматической конфигурации, а запись `--Dspring.profiles.active=cloud` указывает, какие профили или логические группировки с произвольными именами должны быть активизированы в приложении Spring. Эта конфигурация предписывает запуск всех bean-компонентов Spring без профилей, а также имеющих облачный профиль.

Теперь вместо написания всех этих `cf`-заклинаний нужно просто запустить команду `cf push -f manifest.yml`. Вскоре приложение будет запущено и готово к проверке.

Итак, мы увидели, что использование Cloud Foundry направлено на повышение производительности за счет автоматизации: платформа берет на себя возможный максимум трудоемких задач, позволяя сконцентрироваться на по-настоящему важной бизнес-логике. Мы работали в рамках консервативного подхода, предлагаемого Cloud Foundry: если приложению что-нибудь требуется, то можно объявить соответствующее количество `cf`-заклинаний или записей в `manifest.yml`, и платформа поддержит все нужды. Поэтому вызывает удивление тот факт, что, несмотря на незначительный вклад в платформу, сама Cloud Foundry программируется довольно просто. Она предоставляет довольно богатый API, поддерживающий практически все, что вам хочется сделать. Предпринимая дальнейшие шаги, команды Spring и Cloud Foundry разработали Java-клиент Cloud Foundry, поддерживающий все основные компоненты в реализации Cloud Foundry. Java-клиент Cloud Foundry также поддерживает высокоуровневые, более детализированные бизнес-операции,

соответствующие тому, что вы могли бы сделать с помощью интерфейса командной строки (CLI).

Java-клиент Cloud Foundry построен на основе проекта Pivotal Reactor 3.0. Reactor, в свою очередь, лежит в основе среды выполнения reactive web runtime в Spring 5. Java-клиент Cloud Foundry работает весьма *быстро* и построен на принципах reactive, поэтому практически не подвержен блокировкам.

В свою очередь, проект Reactor является реализацией инициативы Reactive Streams. Эта инициатива, сообщается на сайте http://www.reactive-streams.org/, «представляет собой инициативу предоставления стандарта для асинхронного потока, обрабатываемого без блокирующего противодействия». Проект предоставляет язык и API для описания потенциально неограниченного потока *живых* данных, поступающих в асинхронном режиме. Reactor API упрощает написание кода, пользующегося преимуществами распараллеливания, избавляя от необходимости писать код для реализации параллельных вычислений. Ставится цель не допустить агрессивного потребления ресурсов и эффективным образом изолировать блокирующие части облачной рабочей нагрузки. Инициатива Reactive Streams определяет *контроль обратного потока* (backpressure); подписчик может сигнализировать издателю, что не хочет больше получать уведомления. По сути, он сопротивляется производителю, регулируя потребление до тех пор, пока не сможет себе его позволить.

Суть этого API — `org.reactivestreams.Publisher`, способный впоследствии производить от нуля до нескольких значений. Подписчик (`Subscriber`) подписывается на уведомления о новых значениях от издателя (`Publisher`): `Mono` и `Flux`. `Mono<T>` — издатель `Publisher<T>`, производящий одно значение. `Flux<T>` — издатель `Publisher<T>`, производящий от нуля до нескольких значений.

Элементы множества:

- `Synchronous`;
- `Asynchronous`;
- `One`;
- `T`;
- `Future<T>`;
- `Many`;
- `Collection<T>`;
- `org.reactivestreams.Publisher<T>`.

Мы не будем слишком глубоко вникать в реактивные потоки (reactive streams) или в проект Reactor, но учтем, что он закладывает основы удивительной эффективности Java API Cloud Foundry. Данный интерфейс поддается распараллеливанию обработки, которой очень трудно достичь с помощью команды `cf` интерфейса командной строки (CLI). Мы развернули то же самое приложение, используя CLI-команду `cf`, и файл `manifest.yml`; посмотрим на это в Java-коде. Особенно удобно применять Java-клиент Cloud Foundry в комплексных тестах. Чтобы задействовать Java-клиент

Cloud Foundry, нужно сконфигурировать ряд объектов, требуемых для безопасного объединения с различными подсистемами в Cloud Foundry, включая подсистему обобщающей регистрации (log-aggregation subsystem), REST API Cloud Foundry и подсистему аутентификации Cloud Foundry. Мы покажем здесь конфигурацию упомянутых компонентов (пример 2.19), но вполне вероятно, что вам в ближайших выпусках Spring Boot уже не понадобится конфигурировать эти функциональные свойства.

Пример 2.19. Конфигурирование Java-клиента Cloud Foundry

```
package com.example;

import org.cloudfoundry.client.CloudFoundryClient;
import org.cloudfoundry.operations.DefaultCloudFoundryOperations;
import org.cloudfoundry.reactor.ConnectionContext;
import org.cloudfoundry.reactor.DefaultConnectionContext;
import org.cloudfoundry.reactor.TokenProvider;
import org.cloudfoundry.reactor.client.ReactorCloudFoundryClient;
import org.cloudfoundry.reactor.doppler.ReactorDopplerClient;
import org.cloudfoundry.reactor.tokenprovider.PasswordGrantTokenProvider;
import org.cloudfoundry.reactor.uaa.ReactorUaaClient;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class CloudFoundryClientExample {

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

 ❶
 @Bean
 ReactorCloudFoundryClient cloudFoundryClient(
  ConnectionContext connectionContext, TokenProvider tokenProvider) {
  return ReactorCloudFoundryClient.builder()
   .connectionContext(connectionContext).tokenProvider(tokenProvider).build();
 }

 ❷
 @Bean
 ReactorDopplerClient dopplerClient(ConnectionContext connectionContext,
 TokenProvider tokenProvider) {
  return ReactorDopplerClient.builder().connectionContext(connectionContext)
   .tokenProvider(tokenProvider).build();
  }

 ❸
 @Bean
 ReactorUaaClient uaaClient(ConnectionContext connectionContext,
  TokenProvider tokenProvider) {
```

```
  return ReactorUaaClient.builder().connectionContext(connectionContext)
   .tokenProvider(tokenProvider).build();
  }

❹
@Bean
DefaultCloudFoundryOperations cloudFoundryOperations(
 CloudFoundryClient cloudFoundryClient, ReactorDopplerClient dopplerClient,
 ReactorUaaClient uaaClient, @Value("${cf.org}") String organization,
 @Value("${cf.space}") String space) {
 return DefaultCloudFoundryOperations.builder()
  .cloudFoundryClient(cloudFoundryClient).dopplerClient(dopplerClient)
  .uaaClient(uaaClient).organization(organization).space(space).build();
}

❺
@Bean
DefaultConnectionContext connectionContext(@Value("${cf.api}") String
    apiHost) {
 if (apiHost.contains("://")) {
  apiHost = apiHost.split("://")[1];
 }
 return DefaultConnectionContext.builder().apiHost(apiHost).build();
}

❻
@Bean
PasswordGrantTokenProvider tokenProvider(@Value("${cf.user}") String
    username,
 @Value("${cf.password}") String password) {
 return PasswordGrantTokenProvider.builder().password(password)
  .username(username).build();
 }
}
```

❶ `ReactorCloudFoundryClient` — клиент для Cloud Foundry REST API.

❷ `ReactorDopplerClient` — клиент для Doppler, подсистемы Cloud Foundry, основанной на веб-сокетах и предназначенной для обобщающей регистрации.

❸ `ReactorUaaClient` — клиент для UAA, подсистемы авторизации и аутентификации в Cloud Foundry.

❹ Экземпляр `DefaultCloudFoundryOperations` предоставляет укрупненные операции, составляющие клиентов низкоуровневой подсистемы. Все начинается отсюда.

❺ `ConnectionContext` описывает экземпляр Cloud Foundry, на который нужно нацелиться.

❻ `PasswordGrantTokenProvider` описывает аутентификацию.

Имея все это в своем распоряжении, довольно легко получить подтверждение работоспособности. Код в простом примере 2.20 перечисляет все развернутые приложения в конкретном пространстве и организации.

Пример 2.20. Перечисление экземпляров приложений

```
package com.example;

import org.cloudfoundry.operations.CloudFoundryOperations;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
class ApplicationListingCommandLineRunner implements CommandLineRunner {

 private final CloudFoundryOperations cf; ❶

 ApplicationListingCommandLineRunner(CloudFoundryOperations cf) {
  this.cf = cf;
 }

 @Override
 public void run(String... args) throws Exception {
  cf.applications().list().subscribe(System.out::println); ❷
 }
}
```

❶ Внедрение сконфигурированного экземпляра `CloudFoundryOperations`...

❷ ...и использование его для перечисления всех развернутых приложений в этом конкретном пространстве и организации Cloud Foundry.

Чтобы рассмотреть образец, более приближенный к реальным условиям, развернем наше bootcamp-customers приложение, воспользовавшись Java-клиентом. Речь пойдет о простом комплексном тесте, предоставляющем сервис MySQL, который помещает приложение в среду Cloud Foundry (но не запускает его), выполняет привязку переменных среды, привязывает сервис MySQL и затем *наконец-то* запускает приложение. Сначала посмотрим на схематический код, в котором идентифицируется развертываемый файл с расширением `.jar` и имя приложения, а также имя сервиса (пример 2.21). Мы будем делегировать два компонента, `ApplicationDeployer` и `ServicesDeployer`.

Пример 2.21. Схематический код комплексного теста, предоставляющий приложение

```
package bootcamp;

import org.cloudfoundry.operations.CloudFoundryOperations;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.PropertySource;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.io.File;
```

```
import java.time.Duration;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.CyclicBarrier;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = SpringConfigurationIT.Config.class)
public class SpringConfigurationIT {

 @Autowired
 private ApplicationDeployer applicationDeployer;

 @Autowired
 private ServicesDeployer servicesDeployer;

 @Test
 public void deploy() throws Throwable {

  File projectFolder = new File(new File("."), "../spring-configuration");
  File jar = new File(projectFolder, "target/spring-configuration.jar");

  String applicationName = "bootcamp-customers";
  String mysqlSvc = "bootcamp-customers-mysql";
  Map<String, String> env = new HashMap<>();
  env.put("SPRING_PROFILES_ACTIVE", "cloud");

  Duration timeout = Duration.ofMinutes(5);
  servicesDeployer.deployService(applicationName, mysqlSvc, "p-mysql", "100mb")

   ❶
   .then(
    applicationDeployer.deployApplication(jar, applicationName, env, timeout,
     mysqlSvc))    ❷
    .block();    ❸
 }

 @SpringBootApplication
 public static class Config {

  @Bean
  ApplicationDeployer applications(CloudFoundryOperations cf) {
   return new ApplicationDeployer(cf);
  }

  @Bean
  ServicesDeployer services(CloudFoundryOperations cf) {
   return new ServicesDeployer(cf);
  }
 }

}
```

❶ Сначала развертывание поддерживающего сервиса (экземпляра MySQL)...

❷ ...затем развертывание приложения с гарантированной паузой пять минут...

❸ ...а затем блокирование для продолжения теста.

В данном примере составляются два экземпляра `Publisher`, один из которых описывает обработку, необходимую для предоставления сервиса, а другой — обработку, требуемую для предоставления приложения. Финальный вызов в цепочке, `.block()`, запускает обработку; это конечный метод, активирующий весь поток выполнения кода.

`ServicesDeployer` получает требуемые параметры и предоставляет экземпляр MySQL (пример 2.22). Он также удаляет привязку экземпляров и сами экземпляры, если все это уже существует.

Пример 2.22. ServicesDeployer

```
package bootcamp;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.cloudfoundry.operations.CloudFoundryOperations;
import org.cloudfoundry.operations.services.CreateServiceInstanceRequest;
import org.cloudfoundry.operations.services.DeleteServiceInstanceRequest;
import org.cloudfoundry.operations.services.ServiceInstanceSummary;
import org.cloudfoundry.operations.services.UnbindServiceInstanceRequest;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.function.Function;

class ServicesDeployer {

 private final Log log = LogFactory.getLog(getClass());

 private final CloudFoundryOperations cf;

 ServicesDeployer(CloudFoundryOperations cf) {
  this.cf = cf;
 }

 Mono<Void> deployService(String applicationName, String svcInstanceName,
  String svcTypeName, String planName) {

  return cf.services().listInstances().cache()                    ❶
   .filter(si1 -> si1.getName().equalsIgnoreCase(svcInstanceName))  ❷
   .transform(unbindAndDelete(applicationName, svcInstanceName))    ❸
   .thenEmpty(createService(svcInstanceName, svcTypeName, planName)); ❹
 }

 private Function<Flux<ServiceInstanceSummary>, Publisher<Void>>
```

```
unbindAndDelete(
  String applicationName, String svcInstanceName) {
  return siFlux -> Flux.concat(
   unbind(applicationName, svcInstanceName, siFlux),
   delete(svcInstanceName, siFlux));
 }

 private Flux<Void> unbind(String applicationName, String svcInstanceName,
  Flux<ServiceInstanceSummary> siFlux) {
  return siFlux.filter(si -> si.getApplications().contains(applicationName))
   .flatMap(
    si -> cf.services().unbind(
     UnbindServiceInstanceRequest.builder().applicationName(applicationName)
      .serviceInstanceName(svcInstanceName).build()));
 }

 private Flux<Void> delete(String svcInstanceName,
  Flux<ServiceInstanceSummary> siFlux) {
  return siFlux.flatMap(si -> cf.services().deleteInstance(
   DeleteServiceInstanceRequest.builder().name(svcInstanceName).build()));
 }

 private Mono<Void> createService(String svcInstanceName, String svcTypeName,
  String planName) {
  return cf.services().createInstance(
   CreateServiceInstanceRequest.builder().serviceName(svcTypeName)
    .planName(planName).serviceInstanceName(svcInstanceName).build());
 }
}
```

❶ Перечисление всех экземпляров приложения и их кэширование, чтобы последующие подписчики не выполняли повторное вычисление REST-вызова.

❷ Фильтрация всех экземпляров сервиса с оставлением только одного, соответствующего конкретному имени.

❸ Последующая отмена привязки (если она была) и удаление сервиса.

❹ И наконец, создание сервиса заново.

Затем `ApplicationDeployer` предоставляет само приложение (пример 2.23). Оно привязывается к сервису, который только что был подготовлен с помощью `ServicesDeployer`.

Пример 2.23. ApplicationDeployer

```
package bootcamp;

import org.cloudfoundry.operations.CloudFoundryOperations;
import org.cloudfoundry.operations.applications.PushApplicationRequest;
import org.cloudfoundry.operations.applications.
    SetEnvironmentVariableApplicationRequest;
import org.cloudfoundry.operations.applications.StartApplicationRequest;
import org.cloudfoundry.operations.services.BindServiceInstanceRequest;
```

```
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.io.File;
import java.time.Duration;
import java.util.HashMap;
import java.util.Map;

class ApplicationDeployer {

 private final CloudFoundryOperations cf;

 ApplicationDeployer(CloudFoundryOperations cf) {
  this.cf = cf;
 }

 Mono<Void> deployApplication(File jar, String applicationName,
  Map<String, String> envOg, Duration timeout, String... svcs) {
  return cf.applications().push(pushApp(jar, applicationName))  ❶
   .then(bindServices(applicationName, svcs))  ❷
   .then(setEnvironmentVariables(applicationName, new HashMap<>(envOg)))  ❸
   .then(startApplication(applicationName, timeout));  ❹

 }

 private PushApplicationRequest pushApp(File jar, String applicationName) {
  return PushApplicationRequest.builder().name(applicationName).noStart(true)
   .randomRoute(true)
   .buildpack("https://github.com/cloudfoundry/java-buildpack.git")
   .application(jar.toPath()).instances(1).build();
 }

 private Mono<Void> bindServices(String applicationName, String[] svcs) {
  return Flux
   .just(svcs)
   .flatMap(
    svc -> {
     BindServiceInstanceRequest request = BindServiceInstanceRequest.builder()
      .applicationName(applicationName).serviceInstanceName(svc).build();
     return cf.services().bind(request);
    }).then();
 }

 private Mono<Void> startApplication(String applicationName, Duration timeout)
{
  return cf.applications().start(
  StartApplicationRequest.builder().name(applicationName)
     .stagingTimeout(timeout).startupTimeout(timeout).build());
 }

 private Mono<Void> setEnvironmentVariables(String applicationName,
  Map<String, String> env) {
```

```
  return Flux
    .fromIterable(env.entrySet())
    .flatMap(
    kv -> cf.applications().setEnvironmentVariable(
     SetEnvironmentVariableApplicationRequest.builder().name(applicationName)
      .variableName(kv.getKey()).variableValue(kv.getValue()).build())).then();
 }
}
```

❶ Сначала добавляется двоичный файл приложения (путь к нему), указываются пакет buildpack, память и свойства развертывания наподобие счетчика экземпляров. Важно отметить, что пока приложение не запускается. Это будет сделано *после* привязки переменных среды и сервисов. Данный шаг выполняется в автоматическом режиме, когда используются манифесты Cloud Foundry.

❷ Привязка экземпляров сервиса к вашему приложению.

❸ Установка переменных среды для приложения.

❹ И наконец, запуск приложения.

На запуск всего набора уходит немного времени, и его тратится еще меньше, если указать, какие именно ответвления выполняемого кода должны запускаться в разных потоках. Java-клиент Cloud Foundry — довольно эффективный способ описания сложных систем, которому, наряду с некоторыми другими, мы, авторы, отдаем предпочтение. Он также очень удобен, когда развертывание требует применения не только обычного набора сценариев оболочки.

Резюме

В этой главе мы вкратце рассмотрели вопросы начала работы со Spring Boot и поддерживающий инструментарий (например, набор Spring Tool Suite), создание конфигурации Java и затем способы перемещения приложения в облачную среду. Мы автоматизировали развертывание кода в среде его промышленной эксплуатации и увидели, что будет совсем не сложно освоить процесс непрерывного развертывания. В следующих главах все вопросы, касающиеся Spring Boot и Cloud Foundry, мы рассмотрим более обстоятельно.

3 Стиль конфигурации двенадцатифакторных приложений

В этой главе будет рассмотрен порядок реализации конфигурации приложения.

Путаница, связанная с понятием «конфигурация»

Определим ряд словарных терминов. Когда речь заходит о *конфигурации* в Spring, *чаще всего* имеется в виду ввод в среду Spring различных реализаций контекста приложения — `ApplicationContext` (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/ApplicationContext.html), что помогает контейнеру понять, как связать bean-компоненты. Такую конфигурацию можно представить в виде файла в формате XML, который должен быть подан в `ClassPathXmlApplicationContext` (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html), или классов Java, аннотированных способом, позволяющим быть предоставленными объекту `AnnotationConfigApplicationContext` (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/AnnotationConfigApplicationContext.html). И конечно же, при изучении последнего варианта мы станем ссылаться на *конфигурацию Java*.

Но в этой главе мы собираемся рассмотреть конфигурацию в том виде, в котором она определена в манифесте 12-факторного приложения (https://12factor.net/config). В данном случае она касается буквальных значений, способных изменяться от одной среды окружения к другой: речь идет о паролях, портах и именах хостов или же о флагах свойств. Конфигурация игнорирует встроенные в код магические константы. В манифест включен отличный критерий правильности настройки конфигурации: может ли кодовая база приложения быть открытым источником в любой момент без раскрытия и компрометации важных учетных данных? Эта разновидность конфигурации относится исключительно к тем значениям, которые изменяются от одной среды окружения к другой, и не относится, например, к подключению bean-компонентов Spring или конфигурации маршрутов Ruby.

Поддержка во фреймворке Spring

В Spring стиль конфигурации, соответствующий 12 факторам, поддерживается с появления класса `PropertyPlaceholderConfigurer` (`https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/config/PropertyPlaceholderConfigurer.html`). Как только определяется его экземпляр, он заменяет литералы в XML-конфигурации значениями, извлеченными из файла с расширением `.properties`. В среде Spring класс `PropertyPlaceholderConfigurer` (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/PropertyPlaceholderConfigurer.html) предлагается с 2003 года. В Spring 2.5 появилась поддержка пространства имен XML, а вместе с тем и поддержка в данном пространстве подстановки свойств. Это позволяет проводить подстановку в XML-конфигурации литеральных значений определений bean-компонентов значениями, назначенными ключам во внешнем файле свойств (в данном случае в файле `simple.properties`, который может фигурировать в пути к классам или быть внешним по отношению к приложению).

Конфигурация в стиле 12 факторов нацелена на устранение ненадежности имеющихся *магических строк*, то есть значений наподобие адресов баз данных и учетных записей для подключения к ним, портов и т. д., жестко заданных в скомпилированном приложении. Если конфигурация вынесена за пределы приложения, то ее можно заменить, не прибегая для этого к новой сборке кода.

Класс PropertyPlaceholderConfigurer

Посмотрим образец использования класса `PropertyPlaceholderConfigurer`, XML-определений bean-компонентов Spring и вынесенного за пределы приложения файла с расширением `.properties`. Нам нужно просто вывести значение, имеющееся в данном файле свойств. Это поможет сделать код, показанный в примере 3.1.

Пример 3.1. Файл свойств: some.properties

```
configuration.projectName=Spring Framework
```

Это принадлежащий Spring класс `ClassPathXmlApplicationContext`, таким образом, мы используем пространство имен XML из контекста Spring и указываем на наш файл `some.properties`. Затем в определениях bean-компонентов задействуем литералы в форме `${configuration.projectName}`, и Spring в ходе выполнения заменит их значениями из нашего файла свойств (пример 3.2).

Пример 3.2. XML-файл Spring-конфигурации

```
<context:property-placeholder location="classpath:some.properties"/>  ❶

<bean class="classic.Application">
    <property name="configurationProjectName"
    value="${configuration.projectName}"/>
</bean>
```

❶ `classpath:` местоположение, ссылающееся на файл в текущем откомпилированном блоке кода (`.jar`, `.war` и т. д.). Spring поддерживает множество альтернативных вариантов, включая `file:` и `url:`, позволяющих файлу существовать обособленно от блока кода.

И наконец, рассмотрим, как выглядит класс Java, благодаря которому возможно свести все это воедино (пример 3.3).

Пример 3.3. Класс Java, который должен быть сконфигурирован со значением свойства

```
package classic;

import org.apache.commons.logging.LogFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {

 public static void main(String[] args) {
  new ClassPathXmlApplicationContext("classic.xml");
 }

 public void setConfigurationProjectName(String pn) {
  LogFactory.getLog(getClass()).info("the configuration project name is " + pn);
 }
}
```

В первом примере используется XML-формат конфигурации bean-компонентов Spring. В Spring 3.0 и 3.1 ситуация для разработчиков, применяющих конфигурацию Java, значительно улучшилась. В этих выпусках были введены аннотация `@Value` и абстракция `Environment`.

Абстракция Environment и @Value

Абстракция `Environment` (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/core/env/Environment.html) представляет в ходе выполнения кода его косвенное отношение к той среде окружения, в которой он запущен, и позволяет приложению ставить вопрос («Какой разделитель строк `line.separator` на данной платформе?») о свойствах среды. Абстракция действует в качестве отображения из ключей и значений. Сконфигурировав в `Environment` источник свойств `PropertySource`, можно настроить то место, откуда эти значения будут считываться. По умолчанию Spring загружает системные ключи и значения среды, такие как `line.separator`. Системе Spring можно предписать загрузку ключей конфигурации из файла в том же порядке, который мог бы использоваться в ранних выпусках решения Spring по подстановке свойств с помощью аннотации `@PropertySource`.

Аннотация `@Value` предоставляет способ внедрения значений среды окружения в конструкторы, сеттеры, поля и т. д. Эти значения могут быть вычислены с помощью языка выражений Spring Expression Language или синтаксиса подстановки свойств при условии регистрации `PropertySourcesPlaceholderConfigurer` (https://

docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/PropertySourcesPlaceholderConfigurer.html), как сделано в примере 3.4.

Пример 3.4. Регистрация PropertySourcesPlaceholderConfigurer

```
package env;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.
AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.
PropertySourcesPlaceholderConfigurer;
import org.springframework.core.env.Environment;

import javax.annotation.PostConstruct;

❶
@Configuration
@PropertySource("some.properties")
public class Application {

 private final Log log = LogFactory.getLog(getClass());

 public static void main(String[] args) throws Throwable {
  new AnnotationConfigApplicationContext(Application.class);
 }

 ❷
 @Bean
 static PropertySourcesPlaceholderConfigurer pspc() {
  return new PropertySourcesPlaceholderConfigurer();
 }

 ❸
 @Value("${configuration.projectName}")
 private String fieldValue;

 ❹
 @Autowired
 Application(@Value("${configuration.projectName}") String pn) {
  log.info("Application constructor: " + pn);
 }

 ❺
 @Value("${configuration.projectName}")
 void setProjectName(String projectName) {
```

```
  log.info("setProjectName: " + projectName);
 }

 ❻
 @Autowired
 void setEnvironment(Environment env) {
  log.info("setEnvironment: " + env.getProperty("configuration.projectName"));
 }

 ❼
 @Bean
 InitializingBean both(Environment env,
  @Value("${configuration.projectName}") String projectName) {
  return () -> {
   log.info("@Bean with both dependencies (projectName): " + projectName);
   log.info("@Bean with both dependencies (env): "
    + env.getProperty("configuration.projectName"));
  };
 }

 @PostConstruct
 void afterPropertiesSet() throws Throwable {
  log.info("fieldValue: " + this.fieldValue);
 }
}
```

❶ Аннотация `@PropertySource` — сокращение наподобие `property-placeholder`, настраивающее `PropertySource` из файла с расширением `.properties`.

❷ `PropertySourcesPlaceholderConfigurer` нужно зарегистрировать в качестве статического bean-компонента, поскольку он является реализацией `BeanFactoryPostProcessor` и должен вызываться на ранней стадии жизненного цикла инициализации в Spring bean-компонентов. При использовании в Spring XML-конфигурации bean-компонентов этот нюанс не просматривается.

❸ Можно отдекорировать поля аннотацией `@Value` (но не делайте этого, иначе код не пройдет тестирование!)...

❹ ...или аннотацией `@Value` можно отдекорировать параметры конструктора...

❺ ...или воспользоваться методами установки...

❻ ...или внедрить объект Spring `Environment` и выполнить разрешение ключа вручную.

❼ Параметры с аннотацией `@Value` можно использовать также в поставщике аргументов методов `@Bean` в Java-конфигурации Spring.

В этом примере значения загружаются из файла `simple.properties`, а затем в нем имеется значение `configuration.projectName`, предоставляемое различными способами.

Профили

Кроме всего прочего, абстракция `Environment` вводит *профили* (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/Profile.html). Это позволяет приписывать метки (профили) в целях группировки bean-компонентов. Профили следует использовать для описания bean-компонентов и bean-графов, изменяющихся от среды к среде. Одновременно могут активироваться сразу несколько профилей. Bean-компоненты, не имеющие назначенных им профилей, активируются всегда. Bean-компоненты, имеющие профиль `default`, активируются только в том случае, если у них нет других активных профилей. Атрибут `profile` можно указать в определении bean-компонента в XML либо в классах тегов, классах конфигурации, отдельно взятых bean-компонентах или в методах `@Bean`-поставщика с помощью `@Profile`.

Профили позволяют описывать наборы bean-компонентов, которые должны быть созданы в одной среде несколько иначе, чем в другой. В локальном разработочном `dev`-профиле можно, к примеру, воспользоваться встроенным источником данных H2 `javax.sql.DataSource`, а затем, когда активен `prod`-профиль, переключиться на источник данных `javax.sql.DataSource` для PostgreSQL, получаемый с помощью JNDI-поиска или путем чтения свойств из переменной среды в Cloud Foundry (https://cloudfoundry.org/). В обоих случаях ваш код будет работоспособен: вы получаете `javax.sql.DataSource`, но решение о том, *какой* конкретный экземпляр задействовать, принимается с помощью активации одного профиля или нескольких (пример 3.5).

Пример 3.5. Демонстрация того, что классы @Configuration могут загружать различные файлы конфигурации и предоставлять различные bean-компоненты на основе активного профиля

```
package profiles;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.*;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;
import org.springframework.core.env.Environment;
import org.springframework.util.StringUtils;

@Configuration
public class Application {

 private Log log = LogFactory.getLog(getClass());

 @Bean
```

```
 static PropertySourcesPlaceholderConfigurer pspc() {
  return new PropertySourcesPlaceholderConfigurer();
 }

 ❶
 @Configuration
 @Profile("prod")
 @PropertySource("some-prod.properties")
 public static class ProdConfiguration {

  @Bean
  InitializingBean init() {
   return () -> LogFactory.getLog(getClass()).info("prod InitializingBean");
  }
 }

 @Configuration
 @Profile({ "default", "dev" })
 ❷
 @PropertySource("some.properties")
 public static class DefaultConfiguration {

  @Bean
  InitializingBean init() {
   return () -> LogFactory.getLog(getClass()).info("default InitializingBean");
  }
 }

 ❸
 @Bean
 InitializingBean which(Environment e,
                        @Value("${configuration.projectName}") String
projectName) {
  return () -> {
   log.info("activeProfiles: '"
           + StringUtils.arrayToCommaDelimitedString(e.getActiveProfiles()) +
             "'");
   log.info("configuration.projectName: " + projectName);
  };
 }

 public static void main(String[] args) {
  AnnotationConfigApplicationContext ac = new
  AnnotationConfigApplicationContext();
  ac.getEnvironment().setActiveProfiles("dev");  ❹
  ac.register(Application.class);
  ac.refresh();
 }
}
```

1. Этот класс конфигурации и все имеющиеся в нем определения `@Bean` будут вычислены только в случае активности `prod`-профиля.
2. Данный класс конфигурации и все имеющиеся в нем определения `@Bean` будут вычислены, только если активен `dev`-профиль *или* не активен **ни один** профиль, включая `dev`.
3. Этот компонент `InitializingBean` просто записывает текущий активный профиль и вводит значение, которое в конечном итоге было внесено в файл свойств.
4. Активировать профиль (или профили) программным способом довольно просто.

Spring откликается еще на несколько других методов активации профилей, использующих токен `spring_profiles_active` или `spring.profiles.active`. Профиль можно установить с помощью переменной среды (например, `SPRING_PROFILES_ACTIVE`), JVM-свойства (`-Dspring.profiles.active=..`), параметра инициализации сервлет-приложения или программным способом.

Конфигурация Bootiful

Spring Boot (http://spring.io/projects/spring-boot) существенно улучшает ситуацию. Среда изначально автоматически загрузит свойства из иерархии заранее известных мест. Аргументы командной строки переопределяют значения свойств, полученных из JNDI, которые переопределяют свойства, полученные из `System.getProperties()`, и т. д.

- Аргументы командной строки.
- Атрибуты JNDI из `java:comp/env`.
- Свойства `System.getProperties()`.
- Переменные среды операционной системы.
- Внешние файлы свойств в файловой системе: `(config/)?application.(yml.properties)`.
- Внутренние файлы свойств в архиве `(config/)?application.(yml.properties)`.
- Аннотация `@PropertySource` в классах конфигурации.
- Исходные свойства из `SpringApplication.getDefaultProperties()`.

В случае активности профиля (https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-profiles.html) будут автоматически считаны данные из файлов конфигурации, основанных на имени профиля, например из такого файла, как `src/main/resources/application-foo.properties`, где `foo` — текущий профиль.

Если библиотека SnakeYAML (https://bitbucket.org/asomov/snakeyaml) упомянута в путях к классам (classpath), то будут также автоматически загружены YAML-файлы, следуя в основном тому же соглашению.

На странице спецификации YAML (http://yaml.org/) указано, что «YAML является удобным для человеческого восприятия стандартом сериализации данных для всех языков программирования». YAML — иерархическое представление значений. В обычных файлах с расширением .properties иерархия обозначается с помощью точки («.»), а в YAML-файлах используется символ новой строки и дополнительный уровень отступа. Было бы неплохо воспользоваться этими файлами, чтобы избежать необходимости указывать общие корни при наличии сильно разветвленных деревьев конфигурации.

Содержимое файла с расширением `.yml` показано в примере 3.6.

Пример 3.6. Файл свойств application.yml. Данные изложены в иерархическом порядке

```
configuration:
  projectName : Spring Boot
management:
  security:
    enabled: false
```

Кроме того, среда Spring Boot существенно упрощает получение правильного результата в общих случаях. Она превращает аргументы `-D` в переменные процесса и среды `java`, доступные в качестве свойств. Она даже проводит их нормализацию, при которой переменная среды `$CONFIGURATION_PROJECTNAME` (КОНФИГУРАЦИЯ_ИМЯПРОЕКТА) или аргумент `-D` в форме `-Dconfiguration.projectName` (конфигурация.имя_проекта) становятся доступными с помощью ключа `configuration.projectName` (конфигурация.имя_проекта) точно так же, как ранее был доступен токен `spring_profiles_active`.

Значения конфигурации являются строками и при их достаточном количестве могут стать неудобочитаемыми при попытке убедиться, что такие ключи не стали сами по себе магическими строками в коде. В Spring Boot вводится тип компонента `@ConfigurationProperties`. При аннотировании POJO — Plain Old Java Object (обычный старый объект Java) — с помощью `@ConfigurationProperties` и указании префикса среда Spring предпримет попытку отобразить все свойства, начинающиеся с этого префикса, на POJO-свойства. В показанном ниже примере значение для `configuration.projectName` будет отображено на экземпляр POJO, который весь код затем может внедрить и разыменовать для типобезопасного чтения значений. Как следствие, у вас будет только отображение из `(String)` ключа в одном месте (пример 3.7).

Пример 3.7. Автоматическое разрешение свойств из src/main/resources/application.yml

```
package boot;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.
EnableConfigurationProperties;
import org.springframework.stereotype.Component;

❶
@EnableConfigurationProperties
@SpringBootApplication
public class Application {

 private final Log log = LogFactory.getLog(getClass());

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

 @Autowired
 public Application(ConfigurationProjectProperties cp) {
  log.info("configurationProjectProperties.projectName = "
   + cp.getProjectName());
 }
}

❷
@Component
@ConfigurationProperties("configuration")
class ConfigurationProjectProperties {

 private String projectName;  ❸
 public String getProjectName() {
  return projectName;
 }

 public void setProjectName(String projectName) {
  this.projectName = projectName;
 }
}
```

❶ Аннотация `@EnableConfigurationProperties` предписывает среде Spring отображать свойства на POJO-объекты, аннотированные с помощью `@ConfigurationProperties`.

❷ Аннотация `@ConfigurationProperties` показывает среде Spring, что этот bean-компонент должен использоваться как корневой для всех свойств, начинающихся с `configuration.`, с последующими токенами, отображаемыми на свойства объекта.

❸ Поле `projectName` будет в конечном итоге иметь значение, присвоенное ключу свойства `configuration.projectName`.

Spring Boot активно применяет механизм `@ConfigurationProperties`, чтобы дать пользователям возможность переопределять элементарные составляющие системы. Можно заметить, что ключи свойств позволяют вносить изменения, например, путем добавления зависимости `org.springframework.boot:spring-boot-starter-actuator` в веб-приложение на основе Spring Boot с последующим посещением страницы http://127.0.0.1:8080/configprops.

Конечные точки актуатора более подробно будут рассмотрены в главе 13. Они заперты и требуют по умолчанию имени пользователя и пароля. Меры безопасности можно отключить (но только чтобы взглянуть на эти точки), указав management.security.enabled=false в файле application.properties (или в application.yml).

Вы получите список поддерживаемых свойств конфигурации на основе типов, представленных в путях к классам (classpath) во время выполнения. По мере наращивания количества типов Spring Boot будут показываться дополнительные свойства. В этой конечной точке также станут отображаться свойства, экспортированные вашими POJO-объектами, имеющими аннотацию `@ConfigurationProperties`.

Централизованная регистрируемая конфигурация с использованием сервера конфигурации Spring Cloud

Пока все хорошо, однако нужно, чтобы дела шли еще успешнее. Мы по-прежнему не ответили на вопросы, касающиеся общих случаев применения:

- после изменений, внесенных в конфигурацию приложения, потребуется перезапуск;
- отсутствует прослеживаемость: как определить изменения, введенные в эксплуатацию, и при необходимости выполнить их откат?
- конфигурация децентрализована; не сразу видно, куда следует вносить изменения, чтобы изменить тот или иной аспект;
- отсутствует установочная поддержка для кодирования и декодирования в целях безопасности.

Сервер конфигурации Spring Cloud

Проблему централизации конфигурации можно решить, сохранив конфигурацию в одном каталоге и указав всем приложениям на него. Можно также установить управление версиями этого каталога, используя Git или Subversion. Тогда будет получена поддержка, необходимая для проверки и регистрирования. Но последние

два требования по-прежнему не будут выполнены, поэтому нужно нечто более изощренное. Обратимся к серверу конфигурации Spring Cloud (http://cloud.spring.io/spring-cloud-config/). Платформа Spring Cloud предлагает сервер конфигурации и клиента для этого сервера.

Сервер Spring Cloud Config представляет собой REST API, к которому будут подключаться наши клиенты, чтобы забирать свою конфигурацию. Сервер также управляет хранилищем конфигураций с управлением версиями. Он посредник между нашими клиентами и хранилищем конфигурации и таким образом находится в выгодной позиции, позволяющей внедрять средства обеспечения безопасности подключений со стороны клиентов к сервису и подключений со стороны сервиса к хранилищу конфигураций с управлением версиями. Клиент Spring Cloud Config предоставляет клиентским приложениям новую область видимости, `refresh`, дающую возможность конфигурировать компоненты Spring заново, не прибегая к перезапуску приложения.

Технологии, подобные серверу Spring Cloud Config, играют важную роль, но влекут дополнительные рабочие издержки. В идеале эта обязанность должна быть переложена на платформу и автоматизирована. При использовании Cloud Foundry в каталоге сервисов можно найти сервис Config Server, действия которого основаны на применении сервера Spring Cloud Config.

Рассмотрим простой пример. Сначала настроим сервер Spring Cloud Config. К одному такому сервису могут иметь доступ сразу несколько приложений Spring Boot. Вам нужно где-то и как-то заставить его функционировать. Затем останется только оповестить все наши сервисы о том, где найти сервис конфигурации. Он работает как некий посредник для ключей конфигурации и значений, которые он считывает из Git-хранилища по сети или с диска. Добавьте к сборке вашего приложения Spring Boot строку `org.springframework.cloud : spring-cloud-config-server`, чтобы ввести сервер Spring Cloud Config (пример 3.8).

Пример 3.8. Чтобы встроить в сборку сервер конфигурации, воспользуйтесь аннотацией @EnableConfigServer

```
package demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

❶
@SpringBootApplication
@EnableConfigServer
public class Application {

 public static void main(String[] args) {
```

```
  SpringApplication.run(Application.class, args);
 }
}
```

❶ Использование аннотации `@EnableConfigServer` приводит к установке сервера Spring Cloud Config.

В примере 3.9 показана конфигурация для сервиса конфигурации.

Пример 3.9. Конфигурация сервера конфигурации src/main/resources/application.yml

```
server.port=8888
spring.cloud.config.server.git.uri=\
  https://github.com/cloud-native-java/config-server-configuration-repository ❶
```

❶ Указание на работающее Git-хранилище, имеющее локальный характер либо доступное по сети (например, на GitHub (https://github.com/)) и используемое сервером Spring Cloud Config.

Здесь сервису конфигурации Spring Cloud предписывается выполнить поиск файлов конфигурации для отдельно взятых клиентов в Git-хранилище на GitHub. Мы указали на это хранилище, но подошла бы ссылка на любой действующий Git URI. Разумеется, он даже не обязан относится к Git-системе, можно воспользоваться Subversion или даже неуправляемыми каталогами (хотя мы настоятельно не рекомендуем делать это). В данном случае URI хранилища жестко задан, но нет ничего, что помешает получить его из аргумента `-D`, аргумента `--` или из переменной среды окружения.

Клиенты Spring Cloud Config

Для эффективной работы в экосистеме Spring Cloud приложения Spring Boot должны предоставлять имя. Присвойте `spring.application.name` уникальное, имеющее определенный смысл и хорошо запоминающееся имя, такое как `customer-service` (клиент-сервис). Сервис на основе Spring Cloud ищет файл по имени `src/main/resources/bootstrap.(properties,yml)`, в котором он, как вы уже, наверное, догадались, рассчитывает найти загрузку сервиса. Там Spring Cloud станет искать имя сервиса (`spring.application.name`) и местонахождение сервера Spring Cloud Config, откуда он должен получить свою конфигурацию. Данный файл (`bootstrap.properties`) загружается раньше других файлов свойств (включая `application.yml` или `application.properties`). Здесь есть вполне определенный смысл: этот файл сообщает Spring, где искать остальную конфигурацию приложения. Если имеются и `application.properties`, и `bootstrap.properties`, то последний будет загружаться раньше.

Сервер Config Server управляет каталогом, содержащим файлы с расширениями `.properties` или `.yml`. Платформа Spring Cloud обслуживает конфигурирование подключенных клиентов, находя соответствие `spring.application.name` клиента файлу конфигурации в каталоге. Таким образом, клиент конфигурации, идентифи-

цировавший себя как некий сервис (foo), увидит конфигурацию в `fooservice.yml` или в `foo-service.properties`.

Запустите Config Server и убедитесь в работе вашего сервиса конфигурации; укажите в браузере адрес http://localhost:8888/SERVICE/master, где SERVICE — имя, указанное в качестве `spring.application.name`. В нашем Git-хранилище имеется отдельный файл `configuration-client.properties`. Используйте в качестве значения для SERVICE строку `configuration-client`. Код JSON, созданный для этого файла конфигурации, показан на рис. 3.1.

Рис. 3.1. Вывод сервера Spring Cloud Config подтверждает, что ему видна конфигурация в нашем Git-хранилище

В примере 3.10 показан файл конфигурации клиента `bootstrap.yml`.

Пример 3.10. Вариант файла bootstrap.yml

```
spring:
  application:
    name: configuration-client
  cloud:
   config:
     uri: ${vcap.services.configuration-service.credentials.uri:http://
          localhost: 8888}
```

Сервер Spring Cloud Config возвращает соответствующую клиенту конфигурацию путем поиска соответствия `spring.application.name`, но может возвратить глобальную конфигурацию, которая видна *любому* клиенту, из файла `application.properties` или `application.yml` в своем хранилище.

Сервис конфигурации возвращает JSON-код, содержащий все значения конфигурации в файле `application.(properties,yml)`, наряду с любой соответствующей сервису конфигурацией в файле `configuration-client.(yml,properties)`. Он также загрузит любую конфигурацию для заданного сервиса и конкретный профиль, например `configuration-client-dev.properties`, где `configuration-client` — имя клиента, а `dev` — имя профиля, под которым был запущен клиент.

Значения в сервере Spring Cloud Config с точки зрения любого приложения Spring Boot — просто еще один источник свойств `PropertySource`, который разрешается всеми обычными способами: через элементы класса, имеющие аннотацию `@Value`, а также из абстракции `Environment`.

Безопасность

Если в вашем Git-хранилище предприняты меры безопасности, например с помощью HTTP BASIC, то определите свойства `spring.cloud.config.server.git.username` и `spring.cloud.config.server.git.password` для сервера Spring Cloud Config, чтобы разрешить доступ к безопасным Git-хранилищам.

Безопасность самого сервера Spring Cloud Configuration можно обеспечить с помощью аутентификации HTTP BASIC. Проще всего включить `org.springframework.boot : spring-boot-starter-security`, а затем определить свойства `security.user.name` и `security.user.password`. Стартер `spring-boot-starter-security` предоставляет среду Spring Security, поэтому вы можете подключить конкретную реализацию Spring Security `UserDetailsService`, чтобы переопределить порядок обработки аутентификации.

Клиенты Spring Cloud Config могут закодировать имя пользователя и пароль в значении `spring.cloud.config.uri`, например https://user:secret@host.com.

Обновляемая конфигурация

Централизованная конфигурация высокоэффективна, но изменения, вносимые в нее, не сразу видны bean-компонентам, которые от нее зависят. Решение предлагается *обновлением* области видимости, имеющимся в Spring Cloud (и удобной аннотацией `@RefreshScope`). Взглянем на код `ProjectNameRestController`, показанный в примере 3.11.

Пример 3.11. Клиентское приложение Config Server

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

❶
@RestController
@RefreshScope
```

```
class ProjectNameRestController {

 private final String projectName;

 @Autowired
 public ProjectNameRestController(
  @Value("${configuration.projectName}") String pn) {  ❷
  this.projectName = pn;
 }

 @RequestMapping("/project-name")
 String projectName() {
  return this.projectName;
 }
}
```

❶ Аннотация `@RefreshScope` позволяет этому bean-компоненту стать обновляемым.

❷ Spring получает значения конфигурации из Config Server, которое является еще одним источником свойств `PropertySource` в `Environment`.

`ProjectNameRestController` имеет аннотацию `@RefreshScope` (http://cloud.spring.io/spring-cloud-config/spring-cloud-config.html#_refresh_scope), позволяющую получать область видимости, предоставляемую Spring Cloud, что дает любому bean-компоненту возможность на месте создавать самого себя заново (и перечитывать значения конфигурации из сервиса конфигурации). В данном случае повторно создастся `ProjectNameRestController`, его обратные вызовы жизненного цикла будут проигнорированы, а внедрения `@Value` и `@Autowired` — выполнены повторно, как только будет выдано событие *обновления*.

По сути, все bean-компоненты, подвергаемые *обновлению* области видимости, обновят сами себя при получении Spring-события `ApplicationContext`, имеющего тип `RefreshScopeRefreshedEvent`. Исходная реализация повторно создает любые bean-компоненты с аннотацией `@RefreshScope`, сбрасывая весь bean-компонент и создавая его заново. Нет причин, не позволяющих вашим компонентам тоже откликаться на это событие, если у вас есть обновляемое состояние, которое иначе не привязано к внешним значениям в сервере Spring Cloud Config. В примере 3.12 показан простой компонент, всего лишь увеличивающий значение счетчика при каждом событии обновления.

Пример 3.12. Клиентское приложение Config Server

```
package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.cloud.context.scope.refresh.RefreshScopeRefreshedEvent;
```

```
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

import java.util.concurrent.atomic.AtomicLong;

@Component
public class RefreshCounter {

 private final Log log = LogFactory.getLog(getClass());

 private final AtomicLong counter = new AtomicLong(0);  ❶

 ❷
 @EventListener
 public void refresh(RefreshScopeRefreshedEvent e) {
   this.log.info("The refresh count is now at: "
    + this.counter.incrementAndGet());
 }
}
```

❶ В компоненте содержится счетчик, работающий в атомарном режиме...

❷ ...который обновляет сам себя при каждом обнаружении события `RefreshScopeRefreshedEvent`.

Существует несколько способов его запуска. Обновление можно запускать путем отправки пустого POST-запроса на http://127.0.0.1:8080/refresh — конечную точку актуатора Spring Boot, выставляемого автоматически. Как это сделать с помощью команды `curl`, показано в примере 3.13.

Пример 3.13. Выдача события обновления с помощью клиента Spring Cloud Config в конечной точке актуатора /refresh в отношении одного экземпляра

```
curl -d{} http://127.0.0.1:8080/refresh
```

Кроме этого, можно, как показано на рис. 3.2, воспользоваться автоматически сконфигурированной конечной точкой Spring Boot Actuator JMX.

Чтобы увидеть это в действии, внесите изменения в файл конфигурации в Git и как минимум выполните в отношении данного файла команду `git commit`. Затем вызовите конечную точку REST или JMX *не* на сервере конфигурации, а на том узле, где нужно увидеть изменения в конфигурации.

Обе эти конечные точки актуатора Spring Boot работают на основе установки `ApplicationContext` через `ApplicationContext`. Если у вас имеется десять запущенных приложений и нужно увидеть обновленную конфигурацию их всех, то следует вызвать актуатор `Actuator` в отношении каждого экземпляра. Масштабируемость здесь явно хромает!

Канал Spring Cloud Bus (http://cloud.spring.io/spring-cloud-bus/) поддерживает обновление сразу нескольких экземпляров `ApplicationContext` (в частности, множества узлов). Он получает ссылки на все сервисы через работающий канал Spring Cloud

Stream. Последний поддерживает различные технологии сообщений через свою абстракцию привязки, реализация которой предоставляется для RabbitMQ, Apache Kafka, Reactor Project и многих других проектов. Дополнительные сведения о Spring Cloud Stream будут предоставлены в одноименном разделе главы 10.

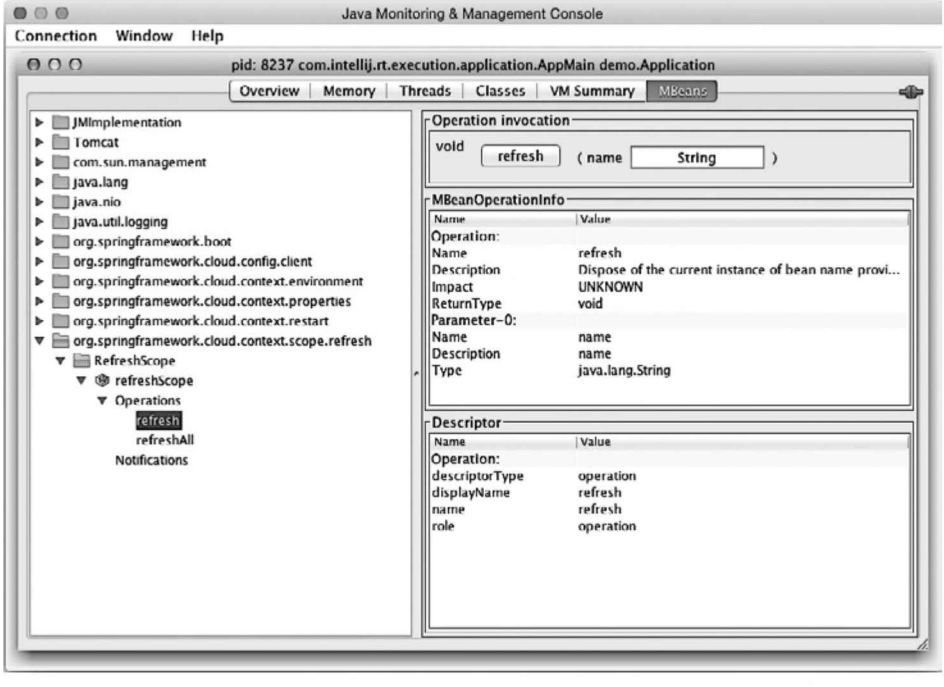


Рис. 3.2. Использование консоли jconsole для активации конечной точки актуатора обновления refresh (или refreshAll)

Эта технология обладает высокой эффективностью. Можно предписать одному (или же тысячам!) микросервису провести самообновление, отправив одно сообщение по каналу. Тем самым будут исключены простои и обеспечен более высокий уровень комфорта для пользователей, чем при необходимости систематически перезапускать отдельные сервисы или узлы. Добавьте зависимость Spring Cloud Bus для RabbitMQ (`org.springframe work.cloud : spring-cloud-starter-bus-amqp`) к путям к классам (classpath).

Изначально автоматическая настройка Spring Boot для RabbitMQ предпримет попытку подключиться к локальному экземпляру RabbitMQ. Можно сконфигурировать конкретный хост и порт, воспользовавшись для этого свойствами, предоставляемыми `Environment` среды Spring. Было бы удобно хранить все упомянутые конфигурации в одном месте, скажем, на сервере Spring Cloud Config. Тогда все сервисы, подключенные к сервису конфигурации, будут также общаться с нужным экземпляром RabbitMQ (пример 3.14).

Пример 3.14. Определение RabbitMQ ConnectionFactory

```
spring:
  rabbitmq:
    host: my-rmq-host
    port: 5672
    username: user
    password: secret
```

Эти значения конфигурации становятся подпиткой для автоматической конфигурации Spring Boot AMQP и выливаются в экземпляр `ConnectionFactory`. Клиент Spring Cloud Bus отслеживает сообщения и, получив их, выдает событие обновления. При наличии в контексте приложения Spring нескольких экземпляров `ConnectionFactory` можно *уточнить*, к какому конкретно экземпляру применяется обновление, задействовав аннотацию `@BusConnectionFactory`. Классифицировать любой другой экземпляр в качестве используемого для обычных действий, не связанных с обработкой канала, можно с помощью имеющейся в Spring классификационной аннотации `@Primary`.

В Spring Cloud Bus открывается *другая* конечная точка актуатора `Actuator`, `/bus/refresh`, которая будет публиковать сообщение в адрес подключенного посредника RabbitMQ, а тот, в свою очередь, станет запускать на самообновление *все* подключенные к нему узлы. С помощью автоматической конфигурации `spring-cloud-starter-bus-amqp` можно отправить следующее сообщение в адрес *любого* узла, что приведет к запуску обновления во *всех* подключенных узлах (пример 3.15).

Пример 3.15. Выдача события обновления с помощью канала Spring Cloud Event Bus

```
curl -d{} http://127.0.0.1:8080/bus/refresh
```

Резюме

В данной главе охвачен *довольно обширный* материал! Вооружившись этими сведениями, вы сможете легко упаковать один компонент, а затем перемещать его из одной среды в другую, не изменяя.

4 Тестирование

Приложения, ориентированные на выполнение в облаке, оптимизированы для быстрой реакции на каждом уровне, от компонентов до целых систем. Основной способ прокладки пути к этому циклу реагирования — тестирование. Микросервисы являются составными частями системы. Spring Boot предоставляет надежную поддержку комплексного тестирования, поддержку, позволяющую уловить объединение компонентов в системе.

По мере роста распространенности приложений существенно изменяются стратегии эффективного написания тестов. Приемы *комплексного тестирования* концентрируются на написании и выполнении тестов, охватывающих группу модулей программного кода, зависящих друг от друга. Такое тестирование — стандартная методика в создании программного обеспечения. С ее помощью разработчики, пишущие отдельные модули или компоненты, могут автоматизировать набор сценариев тестирования, позволяющий получить гарантии того, что ожидаемое функционирование объединения модулей не нарушено, *особенно* когда в код были внесены изменения, оказывающие влияние на это объединение. Зачастую может складываться ситуация, при которой для комплексного тестирования понадобится выполнение тестов в общей объединяющей среде. При подобном сценарии приложения могут стать субъектами одновременного совместного использования внешних ресурсов, таких как база данных или сервер приложения.

Облачные приложения разработаны с прицелом на использование эфемерной облачной среды; нам следует сконцентрироваться на таком создании комплексных тестов, чтобы они могли выполняться в среде, не связанной с другими приложениями. Во многих случаях приложения, предназначенные для выполнения в облаке, зависят от опорных сервисов. Там, где это необходимо, эти зависимости должны быть сымитированы таким образом, чтобы комплексные тесты могли выполняться в разобщенной среде сборки и тестирования. При невозможности имитировать внешнюю зависимость любые внешние опорные сервисы должны быть предоставлены по требованию и демонтированы по завершении тестирования.

Основное внимание в этой главе мы уделим тестированию и двум самым важным темам комплексного тестирования. Первая тема будет посвящена проектированию и созданию основных комплексных тестов для приложений Spring Boot. В рамках этой темы мы исследуем инструменты и свойства Spring, играющие важную роль в создании полноценных комплексных тестов, разобщенных с внешними зависимостями.

Вторая тема будет посвящена исследованию способов выполнения сквозного комплексного тестирования в архитектуре микросервисов. Здесь основное внимание мы уделим тестированию, выполняемому в отношении коллекции различных сервисов в полностью сформированной среде тестирования, сходной со средой промышленной эксплуатации.

Компонентный состав теста

В этой главе мы предполагаем, что вы имеете четкое представление о JUnit и Maven. В частности, рассмотрим комплексные тесты. В наших примерах все тесты будут находиться в `src/test/{java,resources}`, хотя многие организации дошли до того, что создали для тестов отдельное исходное дерево с целью прояснить, что именно нужно запускать постоянно и быстро, а что предназначается для менее частых запусков — возможно, при каждом выпуске кода или после его значительной реорганизации. Независимо от принятого организационного подхода, следует обеспечить автоматизацию запуска *всех* тестов перед введением двоичного кода в эксплуатацию.

Тестирование в Spring Boot

Тестирование в приложениях Spring Boot разбивается на два отдельных стиля: блочные тесты и комплексные. Ко второму относится любой тест, которому в ходе выполнения требуется доступ к контексту приложения Spring (`ApplicationContext`). Блочные тесты создаются таким образом, что им не требуется контекст приложения Spring.

Чтобы включить в вашем приложении Spring Boot поддержку тестирования, нужно к файлу `pom.xml` добавить зависимость от области видимости тестирования, `org.springframework.boot : spring-boot-starter-test`. Если для создания новых проектов используется Spring Initializr (http://start.spring.io/) (мы настоятельно рекомендуем это сделать!), то это уже выполнено.

В примере 4.1 внутри класса `ApplicationTests` показан простой комплексный тест, названный `contextLoads`. Он предназначен для утверждения, что `ApplicationContext` был успешно загружен и внедрен в поле `applicationContext`.

Это похоже на нечто подобное Hello world комплексных тестов, проводимых в среде Spring.

Пример 4.1. Основной комплексный тест загружает ApplicationContext

```
package demo;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.junit4.SpringRunner;

@SpringBootTest
@RunWith(SpringRunner.class)
public class ApplicationContextTests {

 @Autowired
 private ApplicationContext applicationContext;

 @Test
 public void contextLoads() throws Throwable {
  Assert.assertNotNull("the application context should have loaded.",
   this.applicationContext);
 }
}
```

В классе `ApplicationTests` имеются две аннотации: `@RunWith` и `@SpringBootTest`. Первая предназначена для среды JUnit. Она сообщает JUnit, какую из стратегий исполнителя тестов применять. В данном случае наши тесты нужно запускать со средой фреймворка *Spring TestContext*, с модулем в Spring, предоставляющим общую поддержку тестов для приложений Spring.

В Spring Boot 1.4 появился исполнитель SpringRunner. Это более заметная и готовая к работе с JUnit 5 альтернатива имеющемуся в среде фреймворка Spring исполнителю SpringJUnit4ClassRunner, с которой некая часть читателей может быть знакома.

Вторая аннотация в классе `ApplicationTests` — `@SpringBootTest`. Она показывает, что этот класс является классом тестирования Spring Boot, и предоставляет поддержку последовательного поиска `ContextConfiguration`, что сообщает классу тестирования, как загружать `ApplicationContext`. Если в качестве параметра к аннотации `@SpringBootTest` классы `ContextConfiguration` не указаны, то исходным поведением станет загрузка контекста приложения `ApplicationContext` с помощью

последовательного поиска аннотации `@SpringBootConfiguration` в классе в корневом каталоге пакета.

Как уже объяснялось в главе 2, аннотация @SpringBootApplication — стереотипное определение, объединяющее в приложении Spring Boot сразу несколько низкоуровневых аннотаций. Одной из них является аннотация @SpringBootConfiguration. Таким образом, класс с @SpringBootTest найдет класс приложения, который загружает нужный контекст приложения ApplicationContext.

В примере 4.2 показана структура папок основного исходного кода приложения Spring Boot. Обратите внимание на наличие двух корневых исходных папок приложения, в которых структуры пакетов идентичны. Чтобы методы тестирования внутри файла `ApplicationTests.java` могли успешно вести последовательный поиск и находить класс приложения Spring Boot, содержащийся в `application.java` и имеющий аннотацию `@SpringBootApplication`, структура пакетов должна быть идентичной в каждой из корневых исходных папок.

Пример 4.2. Основная структура нового проекта Spring Boot

```
.
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    ├── main    ❶
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           └── Application.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test    ❷
        └── java
            └── com
                └── example
                    └── ApplicationTests.java
```

❶ Корневая папка исходного кода содержит класс приложения.

❷ Корневая папка исходного теста содержит тестовые классы приложения.

`Application.java` является простым методом `main(String[] args)` в классе с аннотацией `@SpringBootApplication`.

Усвоив основные положения по использованию зависимости, имеющейся у средства запуска тестов Spring Boot и служащей для создания автоконфигурируемого

тестового класса JUnit, можно приступать к проектированию и созданию комплексных тестов, нацеленных на взаимодействие различных компонентов, загружаемых в `ApplicationContext`.

Комплексное тестирование

Как уже ранее упоминалось, тесты, которые нужно создать для вашего приложения Spring Boot, делятся по стилям на две категории: *блочные* и *комплексные*. Основное их отличие друг от друга заключается в том, что вторые требуют для тестирования сопряжения различных компонентов использования контекста Spring, а первые позволяют протестировать отдельно взятые компоненты без каких-либо зависимостей от библиотек Spring. В этом разделе мы собираемся уделить основное внимание непосредственно написанию комплексных тестов в приложении Spring Boot.

Имеющееся в Spring Boot автоматическое конфигурирование может затруднить написание комплексных и блочных тестов из-за сложностей изолирования тех компонентов, которые изменяются от среды к среде.

В одном из важных руководств по методологии 12-факторных приложений утверждается необходимость минимизации расхождений между режимами создания и эксплуатации. Для большинства сред запуска приложений, отличающихся от Spring Boot, при создании облачного приложения, возможно, будет важно неуклонно следовать этому совету. Для приложений Spring Boot, предназначенных для работы в облаке, правила, относящиеся к соответствию разработочной и производственной сред, могут быть несколько изменены, поскольку автоматическое конфигурирование позволяет переключать внешние зависимости на зависимости от имитаторов. Разумеется, конкретным местом для тестирования сквозного поведения приложения является среда, представляющая собой точное отображение опорных сервисов, используемых при реальной эксплуатации, но более подробно этот вопрос мы рассмотрим в главе 15.

Тестовые срезы

Перед выпуском Spring Boot 1.4 комплексные тесты в основном создавались для выполнения после загрузки всего Spring-контекста приложения. Во многом это был обременительный побочный эффект, связанный с автоматическим конфигурированием, поскольку не всем комплексным тестам требовался доступ ко всему контексту приложения `ApplicationContext`.

Что, если, к примеру, нужно протестировать лишь сериализацию и маршализацию Java-объектов в JSON-формат и обратно? В таком случае вряд ли необходимо загружать автоматическое конфигурирование для *Spring MVC* или *Spring Data JPA*. Разве нужно, чтобы был загружен сервлет-контейнер, если тестируется всего лишь JSON-сериализация? Нет, без него можно обойтись. В версиях, начиная со Spring Boot 1.4,

поддерживается понятие *тестовых срезов* — выборочной активации по срезам в рамках автоматического конфигурирования отдельно взятых слоев комплекса.

Кроме того, тестовые срезы предоставляют возможность совершать четкую замену определенных стартовых проектов, например переключаться со *Spring Data JPA* на *Spring Data MongoDB*, не оказывая при этом никакого влияния на комплексные тесты, не связанные со Spring Data. Кроме тестовых срезов, в Spring Boot 1.4 включена расширенная высококачественная поддержка написания классов комплексных тестов, позволяющая разработчикам проводить декларативную имитацию взаимодействующих компонентов в Spring-контексте с помощью аннотаций.

Имитация, используемая в тестах

Об имитации обычно говорится в контексте блочного тестирования. В наипростейшем смысле *объекты-имитаторы* позволяют изолировать части системы, подвергаемые тестированию, путем замены компонентов, способствующих работе модуля, объектами-имитаторами, нацеленными на тестирование поведения управляемым образом. Если одному из модулей приложения требуется доступ к внешнему опорному сервису, вероятно, в виде вызова другого микросервиса, то можно подключить имитируемую реализацию этого опорного сервиса, выдающую всегда один и тот же результат. Останется только тестируемый компонент и его взаимодействие с имитатором.

Когда речь заходит о различии между комплексным и блочным тестом, имитаторы в Spring Boot становятся довольно тонкой темой. Для приложений Spring Boot имитаторы в комплексных тестах могут быть так же полезны, как и в блочных. При написании в Spring Boot комплексных тестов появляется возможность имитировать только избранные компоненты в `ApplicationContext` тестового класса. Эта весьма полезная особенность позволяет разработчикам тестировать сопряжение компонентов, сотрудничающих друг с другом, сохраняя при этом возможность имитации объектов на границе приложения. Но различие между блочным и комплексным тестами *все же* имеет место, независимо от того, полностью ли используется контекст среды Spring. О данном различии не нужно забывать; оно поможет выстроить общение между сотрудниками одной команды, создающими как блочные, так и комплексные тесты.

В Spring Boot поддерживается аннотация `@MockBean`. Она принуждает среду Spring к предоставлению имитатора bean-компонента в контексте приложения и полному отключению определения исходного, настоящего bean-компонента в контексте приложения. Среда создает внутри `ApplicationContext` имитатор `Mockito`. В примере 4.3 показан класс `AccountServiceTests`, снабженный аннотацией `@RunWith(SpringRunner.class)`, — демонстрация того, что в ходе выполнения теста `ApplicationContext` не будет загружаться. Тестируемым компонентом в этом классе является bean-компонент `AccountService`; его работа зависит от сопряжения с внешним микросервисом, доступ к которому обеспечивается через bean-компонент `UserService`.

Пример 4.3. Имитация компонентов UserService и AccountRepository с помощью аннотации @MockBean

```
package demo.account;

import demo.user.User;
import demo.user.UserService;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.Collections;
import java.util.List;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.BDDMockito.given;

@RunWith(SpringRunner.class)
public class AccountServiceTests {

 @MockBean
 ❶
 private UserService userService;

 @MockBean
 private AccountRepository accountRepository;

 private AccountService accountService;

 @Before
 public void before() {
  accountService = new AccountService(accountRepository, userService); ❷
 }

 @Test
 public void getUserAccountsReturnsSingleAccount() throws Exception {
  given(this.accountRepository.findAccountsByUsername("user")).willReturn(
   Collections
    .singletonList(new Account("user", new AccountNumber("123456789")))); ❸

  given(this.userService.getAuthenticatedUser()).willReturn(
   new User(0L, "user", "John", "Doe")); ❹

  List<Account> actual = accountService.getUserAccounts(); ❺

  assertThat(actual).size().isEqualTo(1);
  assertThat(actual.get(0).getUsername()).isEqualTo("user");
  assertThat(actual.get(0).getAccountNumber()).isEqualTo(
   new AccountNumber("123456789"));
 }
}
```

❶ Создание имитатора Mockito для компонента UserService.

❷ Создание нового экземпляра AccountService с имитируемыми компонентами в виде параметров.

❸ Глушение вызова метода репозитория findAccountsByUsername(String username) с помощью списка Account.

❹ Глушение вызова метода getAuthenticatedUser() с помощью нового экземпляра User.

❺ Вызов метода getUserAccounts(), принадлежащего AccountService, в ходе использования определенных имитаторов.

В этом примере имитируется поведение вспомогательных компонентов, которые позже понадобятся для проведения комплексного тестирования. В этом блочном тесте можно проверить функционирование AccountService, не прибегая к выполнению удаленных HTTP-вызовов опорных сервисов внутри bean-компонента UserService. То же самое относится к компоненту AccountRepository. В обычной ситуации этот репозиторный компонент был бы определен как bean, подпадающий под автоматическое конфигурирование внутри ApplicationContext, что обеспечило бы управление данными для элемента Account, отображенного на таблицу в реляционной базе данных. Чтобы упростить тестирование и свести его к проверке функционирования AccountService, можно создать для компонента AccountRepository имитатор и описать ожидаемое поведение при взаимодействии с ним в ходе тестирования. А теперь взглянем на содержимое AccountService с целью разобраться, что там делается.

Компонент AccountService, для которого создается тест, определен в примере 4.4.

Пример 4.4. Определение компонента AccountService, имеющего зависимость от взаимодействующих компонентов

```
package demo.account;

import demo.user.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Collections;
import java.util.List;
import java.util.Optional;

@Service
public class AccountService {

 private final AccountRepository accountRepository;

 private final UserService userService; ❶

 @Autowired
 public AccountService(AccountRepository ar, UserService us) { ❷
```

```
  this.accountRepository = ar;
  this.userService = us;
 }

 public List<Account> getUserAccounts() {
  ❸
  return Optional.ofNullable(userService.getAuthenticatedUser())
   .map(u -> accountRepository.findAccountsByUsername(u.getUsername()))
   .orElse(Collections.emptyList());
 }
}
```

❶ Поле для bean-компонента `UserService`, которое будет внедряться через конструктор.

❷ Внедрение на основе конструктора вставит bean-компоненты из `ApplicationContext` для каждого параметра.

❸ Метод `getAuthenticatedUser()` выполняет удаленный HTTP-вызов пользовательского микросервиса.

Стоит упомянуть, что `AccountService` задействует внедрение с помощью конструктора. Можно было бы применить внедрение с использованием поля, но тогда мы утратили бы возможность понимать, что компонент ожидает в качестве предварительных условий для создания допустимого объекта. Вместо *внедрения с использованием поля* нужно *всегда* применять *внедрение на основе конструктора*. Наряду с тем что эта идея продуктивна в общем смысле, она также приобретает новое измерение в процессе тестирования.

В `AccountService` Spring `ApplicationContext` предоставляет по возможности ссылку на `AccountRepository`. Чтобы переопределить это поведение, создайте новый экземпляр `AccountService` и выполните непосредственную инициализацию класса со ссылкой на имитатор объекта для `AccountRepository`. Теперь класс тестирования больше напоминает блочный тест, что вполне успешно изолирует работу компонента `AccountService` от его зависимостей от взаимодействующих с ним компонентов.

Посмотрим на компонент `UserService`. В его определении, показанном в примере 4.5, класс `UserService` содержит единственный метод, который будет выполнять удаленный HTTP-вызов к REST API опорного сервиса с целью извлечь аутентифицированный объект `User`.

Пример 4.5. Bean-компонент UserService выполняет удаленные HTTP-вызовы внешнего микросервиса

```
package demo.user;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.HttpHeaders;
import org.springframework.http.RequestEntity;
import org.springframework.stereotype.Service;
```

```
import org.springframework.web.client.RestTemplate;

import java.net.URI;

import static org.springframework.http.MediaType.APPLICATION_JSON_VALUE;
import static org.springframework.http.RequestEntity.get;

@Service
public class UserService {

 private final String serviceHost;

 private final RestTemplate restTemplate;

 @Autowired
 public UserService(RestTemplate restTemplate,
  @Value("${user-service.host:user-service}") String sh) {
  this.serviceHost = sh;
  this.restTemplate = restTemplate;
 }

 public User getAuthenticatedUser() {
  URI url = URI.create(String.format("http://%s/uaa/v1/me", serviceHost));
  RequestEntity<Void> request = get(url).header(HttpHeaders.CONTENT_TYPE,
   APPLICATION_JSON_VALUE).build();
  return restTemplate.exchange(request, User.class).getBody(); ❶
 }
}
```

❶ Выполнение удаленного HTTP-вызова по указанному URL и возвращение экземпляра класса `User`.

Этот компонент `UserService` устроен весьма просто, для GET-запроса по протоколу HTTP в адрес удаленной зависимости используется `RestTemplate`. Создав имитацию и заглушку для метода `getAuthenticatedUser` компонента `UserService`, которые определены в классе `AccountServiceTests`, мы добиваемся еще большей изоляции тестируемой системы, удаляя любые зависимости от внешнего сервиса, поскольку в среде сборки-тестирования удаленный ресурс может быть недоступен.

Аннотация `@MockBean` представляет определенную ценность для комплексного тестирования класса контроллера *Spring MVC*, зависящего от нескольких взаимодействующих компонентов, которые должны объединяться с удаленными сервисами с помощью HTTP, что довольно часто случается при проведении комплексных тестов в микросервисах. Поскольку для тестирования контроллера Spring MVC нужна веб-среда, то потребуется `ApplicationContext`. В данном сценарии необходимо создать лишь имитаторы сервисов, выполняющих вызовы к удаленным приложениям. Благодаря имитации объектов на границе веб-приложения появляется возможность изолировать тестируемые компоненты от их зависимостей от удаленных веб-сервисов. Это позволяет проводить комплексное тестирование взаимодействия модулей приложения, не выходя за границы JVM.

Для комплексных тестов, требующих веб-среду, аннотация `@SpringBootTest` предоставляет дополнительные настраиваемые параметры, уточняющие порядок запуска тестирования в сервлетной среде.

Работа с Servlet Container в @SpringBootTest

Как уже ранее упоминалось в данной главе, Spring Boot теперь предоставляет несколько тестовых аннотаций, помогающих ориентировать тесты на конкретные срезы классов автоматического конфигурирования.

Ранее в этой главе был показан пример использования `@SpringBootTest` для выполнения комплексного теста Spring Boot `ApplicationContext`. В большинстве случаев приложению понадобится доступ к сервлет-контейнеру, даже если нужно всего лишь выставить показатели работоспособности из конечной точки HTTP. Обычно если был затребован сервлет-контейнер, то проводится компиляция приложения Spring MVC в качестве компонента `.war` и его развертывание в запущенном сервере приложения. Теперь же мы рекомендуем для приложений Spring Boot встроенные `.jar`-развертывания.

Когда в приложении Spring Boot нужно создать комплексные тесты в отношении полностью сконфигурированного `ApplicationContext`, должна использоваться аннотация `@SpringBootTest`. Она также позволит сконфигурировать сервлет-среду для контекста вашего теста. В `@SpringBootTest` поддерживается атрибут `webEnvironment`, позволяющий дать описание того, как в Spring Boot нужно сконфигурировать встроенный сервлет-контейнер, который ваше приложение задействует в процессе работы. В таблице 4.1 представлена сводка атрибутов для принадлежащего `@SpringBootTest` атрибута `webEnvironment`.

Таблица 4.1. Атрибут webEnvironment, принадлежащий @SpringBootTest

Режим	Описание
MOCK	Приводит к загрузке WebApplicationContext и предоставляет среду имитации сервлета
DEFINED_PORT	Приводит к загрузке EmbeddedWebApplicationContext и предоставляет реальную сервлет-среду по указанному порту
RANDOM_PORT	Приводит к загрузке EmbeddedWebApplicationContext и предоставляет реальную сервлет-среду по произвольному порту
NONE	Приводит к загрузке ApplicationContext с помощью SpringApplication, но не предоставляет никакой сервлет-среды (ни имитационной, ни какой-либо другой)

В конечном счете решение зависит от того, насколько сильно вы собираетесь задержать ответную реакцию при запуске тестов. Здесь имеет место компромисс между доверием к результату и скоростью итерации.

В большинстве случаев полномасштабная сервлет-среда может стать излишней для ваших комплексных тестов. Сервлет-контейнер придется перезагружать для каждого тестового класса, чтобы запустить содержащиеся в нем тесты. Учитывая, что время, затрачиваемое на запуск отдельно взятого сервлет-контейнера, будет варьироваться в зависимости от среды сборки-тестирования, общее время, затрачиваемое на выполнение каждого комплексного теста, может быть слишком большим. В мире непрерывного развертывания и микросервисов время сборки легко способно стать препятствующим ограничением.

Срезы

Spring Boot предоставляет несколько аннотаций тестирования, нацеленных на конкретные срезы вашего приложения.

Аннотация @JsonTest

Данная аннотация позволяет активировать только ту конфигурацию, которая предназначена для тестирования JSON-сериализации и десериализации. Рассмотрим образец. В нем показан класс `UserTests`, сопровожденный аннотацией `@JsonTest`. Этот класс проверяет порядок сериализации и десериализации объекта `User` (пример 4.6).

Пример 4.6. Использование @JsonTest для тестирования JSON-сериализации и десериализации

```
package demo.user;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.json.JsonTest;
import org.springframework.boot.test.json.JacksonTester;
import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@JsonTest
public class UserTests {

 private User user;

 @Autowired
 ❶
 private JacksonTester<User> json;

 @Before
```

```
 public void setUp() throws Exception {
  User user = new User("user", "Jack", "Frost", "jfrost@example.com");
  user.setId(0L);
  user.setCreatedAt(12345L);
  user.setLastModified(12346L);

  this.user = user;
 }

 @Test
 public void deserializeJson() throws Exception {
  String content = "{\"username\": \"user\", \"firstName\": \"Jack\", "
   + "\"lastName\": \"Frost\", \"email\": \"jfrost@example.com\"}";

  assertThat(this.json.parse(content)).isEqualTo(
   new User("user", "Jack", "Frost", "jfrost@example.com"));
  assertThat(this.json.parseObject(content).getUsername()).isEqualTo("user");
 }

 @Test
 public void serializeJson() throws Exception {
  ❷
  assertThat(this.json.write(user)).isEqualTo("user.json");
  assertThat(this.json.write(user)).isEqualToJson("user.json");
  assertThat(this.json.write(user)).hasJsonPathStringValue("@.username");
  ❸
  assertJsonPropertyEquals("@.username", "user");
  assertJsonPropertyEquals("@.firstName", "Jack");
  assertJsonPropertyEquals("@.lastName", "Frost");
  assertJsonPropertyEquals("@.email", "jfrost@example.com");
 }

 private void assertJsonPropertyEquals(String key, String value)
  throws java.io.IOException {
  assertThat(this.json.write(user)).extractingJsonPathStringValue(key)
   .isEqualTo(value);
 }
}
```

❶ JSON-тестер на основе AssertJ, поддерживаемый Jackson.

❷ Написание объекта `User` в формате JSON и сравнение результата с файлом `user.json`.

❸ Подтверждение того, что фактический результат JSON соответствует ожидаемому значению свойства.

Существует множество вариантов подтверждений того, соответствует ли фактический результат JSON-сериализации ожидаемому результату тестового метода. Возможность отобразить JSON-файл в качестве тестового classpath-ресурса весьма полезна при очистке тела тестовых методов, особенно когда ожидаемый

JSON-результат обладает множеством свойств. Один из способов выполнения сравнения предусматривает использование в теле теста обыкновенных жестко заданных JSON-строк, как в `deserializeJson`.

Тест `serializeJson`, получающий объект `User`, проинициализированный в методе `setup` и предпринимающий попытку его сериализации в формате JSON, показывает альтернативный подход. В нем есть ссылка на ресурс `user.json`, выраженная в виде параметра метода `isEqualTo`. Для проверки того факта, что модуль записи Jackson JSON выдает ожидаемый результат, здесь можно указать в качестве classpath-ресурса JSON-файл. Файлы с расширением `.json` находятся в структуре каталогов в `src/main/resources`, отображаемой на пакеты тестовых классов (пример 4.7).

Пример 4.7. Структура корневого каталога источников тестов, содержащая класс UserTests

```
├── ./src/test
│
├── java
│   └── demo
│       └── user
│           ├── UserControllerTest.java
│           ├── UserRepositoryTest.java
│           └── UserTests.java
└── resources
    ├── data-h2.sql
    └── demo
        └── user
            └── user.json *
```

Посмотрим на содержимое `user.json` (пример 4.8).

Пример 4.8. Содержимое файла user.json в тестовых ресурсах

```
{
  "username": "user",
  "firstName": "Jack",
  "lastName": "Frost",
  "email": "jfrost@example.com",
  "createdAt": 12345,
  "lastModified": 12346,
  "id": 0
}
```

Аннотация @WebMvcTest

Данная аннотация позволяет поддерживать тестирование отдельно взятых контроллеров Spring MVC в приложении Spring Boot. Ее использование приводит к автоматической конфигурации инфраструктуры Spring MVC, необходимой для тестового взаимодействия с методами контроллера (пример 4.9).

Пример 4.9. Использование @WebMvcTest для тестирования контроллера Spring MVC

```
package demo.account;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import java.util.Collections;

import static org.mockito.BDDMockito.given;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders .get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers
   .content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers
   .status;

@RunWith(SpringRunner.class)
@WebMvcTest(AccountController.class)
public class AccountControllerTest {

 @Autowired
 private MockMvc mvc;  (1)

 @MockBean
 private AccountService accountService; (2)

 @Test
 public void getUserAccountsShouldReturnAccounts() throws Exception {
  String content = "[{\"username\": \"user\", \"accountNumber\":
      \"123456789\"}]";

  (3)
  given(this.accountService.getUserAccounts()).willReturn(
   Collections.singletonList(new Account("user", "123456789")));

  (4)
  this.mvc.perform(get("/v1/accounts").accept(MediaType.APPLICATION_JSON))
   .andExpect(status().isOk()).andExpect(content().json(content));
 }
}
```

1. Имитация MVC-клиента для выполнения HTTP-запросов к контроллерам Spring MVC.
2. Имитация компонента `AccountService`, взаимодействующего с `AccountController`.
3. Определение ожидаемого поведения при извлечении данных учетной записи пользователя из bean-компонента `accountService`.
4. И наконец, применение `MockMvc`-клиента для утверждения HTTP-результата, ожидаемого от `AccountController`.

В `AccountControllerTest` показывается использование `MockMvc`-клиента для выполнения имитационных запросов к тестируемому контроллеру Spring MVC. `MockMvc` берется из тестовой среды Spring MVC. Сконфигурированный тест поддерживает все структуры Spring MVC, *за исключением* настоящего сервлет-контейнера, и направляет запросы от клиента к самому контроллеру. Последний создает ответ, отправляемый обратно клиенту. Все происходит почти так же, как и при совершении вызовов к настоящему сервису по каналу связи, за исключением того, что в данном случае нет HTTP-сервиса. Ничто и никогда не занимается организацией и обслуживанием запросов к `ServerSocket`.

Аннотация @DataJpaTest

С выходом Spring Data 1.4 предоставляется новая аннотация тестирования `@DataJpaTest`. Она пригодится приложениям Spring Boot, использующим проект Spring Data JPA. Для его тестирования поддерживается встроенная в память база данных. В дальнейшем, когда в главе 9 пойдет речь о доступе к данным, мы проведем исследование способов конфигурирования различных профилей Spring для комплексного тестирования. Мы рассмотрим, как в ходе выполнения программы можно переключаться между MySQL и H2, встроенной в память реляционной базой данных.

В примере 4.10 показан тестовый класс `AccountRepositoryTest`, имеющий аннотацию `@DataJpaTest`. Под этим срезом активируются только те классы автоматического конфигурирования, которые требуются для выполнения тестов в отношении хранилищ Spring Data JPA. В методе проверки используется `TestEntityManager`, удобный класс из Spring, поддерживающий полезный поднабор подходящего диспетчера JPA `EntityManager`, наряду с некоторыми дополнительными полезными методами, упрощающими идиомы, часто применяемые в тестах. Диспетчер `TestEntityManager` — весьма полезный компонент, используемый в области видимости тестов JPA-хранилища и позволяющий взаимодействовать с базовым хранилищем данных с целью сохранения объектов, при этом не нуждаясь в использовании хранилища. Он служит для сохранения нового экземпляра `Account`,

который будет представлять фактический результат, ожидаемый к возвращению из хранилища `accountRepository`.

Пример 4.10. Использование @DataJpa для тестирования хранилища Spring Data JPA

```
package demo.account;

import demo.customer.CustomerRepository;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@DataJpaTest
public class AccountRepositoryTest {

 private static final AccountNumber ACCOUNT_NUMBER = new AccountNumber(
  "098765432");

 @Autowired
 private AccountRepository accountRepository;  ❶

 @Autowired
 private TestEntityManager entityManager;  ❷

 @Test
 public void findUserAccountsShouldReturnAccounts() throws Exception {
  this.entityManager.persist(new Account("jack", ACCOUNT_NUMBER));  ❸
  List<Account> account =
      this.accountRepository.findAccountsByUsername("jack"); ❹
  assertThat(account).size().isEqualTo(1);
  Account actual = account.get(0);
  assertThat(actual.getAccountNumber()).isEqualTo(ACCOUNT_NUMBER);
  assertThat(actual.getUsername()).isEqualTo("jack");
 }

 @Test
 public void findAccountShouldReturnAccount() throws Exception {
  this.entityManager.persist(new Account("jill", ACCOUNT_NUMBER));
  Account account = this.accountRepository
   .findAccountByAccountNumber(ACCOUNT_NUMBER);
  assertThat(account).isNotNull();
```

```
  assertThat(account.getAccountNumber()).isEqualTo(ACCOUNT_NUMBER);
 }

 @Test
 public void findAccountShouldReturnNull() throws Exception {
  this.entityManager.persist(new Account("jack", ACCOUNT_NUMBER));
  Account account = this.accountRepository
    .findAccountByAccountNumber(new AccountNumber("000000000"));
  assertThat(account).isNull();
 }
}
```

1. Внедрение `AccountRepository` из `ApplicationContext`.
2. Внедрение `TestEntityManager` для управления сохранением без использования хранилища.
3. Сохранение нового экземпляра `Account` в базе данных, сконфигурированной для контекста и организованной в памяти.
4. Нахождение сохраненного `Account`.

Аннотация @RestClientTest

Эта аннотация предоставляет поддержку тестирования имеющегося в Spring шаблона `RestTemplate` и его взаимодействия с REST-сервисами.

В примере 4.11 показан блочный тест, имеющий аннотацию `@RestClientTest`. Мы указываем, что будем ориентироваться на класс `UserService` и нам нужно, чтобы `RestTemplate` был зарегистрирован в качестве части среза теста автоматического конфигурирования. В методе тестирования `getAuthenticatedUserShouldReturnUser()` используется поле `MockRestServiceServer` (сервер) для имитации ожидаемого поведения предполагаемого HTTP-запроса, который выполняется `RestTemplate` внутри `UserService`. Обратите внимание на то, как ожидаемый JSON-ответ загружается из classpath-ресурса, а именно из файла `user.json`.

Пример 4.11. Использование @RestClientTest для имитации ответов RestTemplate

```
package demo.user;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;

//@formatter:off
import org.springframework.boot.test.autoconfigure.web.client.
AutoConfigureWebClient;
import org.springframework.boot.test.autoconfigure.web.client.RestClientTest;
//@formatter:on

import org.springframework.core.io.ClassPathResource;
```

```
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.client.MockRestServiceServer;

import static org.assertj.core.api.Assertions.assertThat;

//@formatter:off
import static org.springframework.test.web.client.match
        .MockRestRequestMatchers.requestTo;
import static org.springframework.test.web.client.response
        .MockRestResponseCreators.withSuccess;
//@formatter:on

❶
@RunWith(SpringRunner.class)
@RestClientTest({ UserService.class })
@AutoConfigureWebClient(registerRestTemplate = true)
public class UserServiceTests {

 @Value("${user-service.host:user-service}")
 private String serviceHost;

 @Autowired
 private UserService userService;

 @Autowired
 private MockRestServiceServer server;

 @Test
 public void getAuthenticatedUserShouldReturnUser() {
  this.server.expect(
   requestTo(String.format("http://%s/uaa/v1/me", serviceHost))).
       andRespond(withSuccess(new ClassPathResource("user.json", getClass()),
       MediaType.APPLICATION_JSON));  ❷

  User user = userService.getAuthenticatedUser();

  assertThat(user.getUsername()).isEqualTo("user");
  assertThat(user.getFirstName()).isEqualTo("John");
  assertThat(user.getLastName()).isEqualTo("Doe");
  assertThat(user.getCreatedAt()).isEqualTo(12345);
  assertThat(user.getLastModified()).isEqualTo(12346);
  assertThat(user.getId()).isEqualTo(0L);
 }
}
```

❶ Регистрация `RestTemplate` для контекста тестового среза автоматического конфигурирования.

❷ Имитация поведения запроса `RestTemplate` по указанному URL.

После использования `MockRestServiceServer` для имитации HTTP-ответа от удаленного сервиса любое применение `RestTemplate`, соответствующее заявленному

ожиданию, будет возвращать взамен содержимое `user.json`. Положительным моментом здесь является возможность ориентации на части системы, подвергаемые тестированию с помощью имитации поведения внешних сервисов. Польза от этой привлекательной особенности наиболее хорошо видна в тестировании микросервисов, которым приходится координировать свою работу с другими сервисами, применяя технологию HTTP. Возникает вопрос: как создаются комплексные тесты, имитирующие фактическое поведение удаленного HTTP-сервиса вместо использования `MockRest ServiceServer`? Ответ дается чуть позже, в подразделе «Тестирование контрактов, ориентированных на потребителя». Такое тестирование позволяет загружать заглушки, публикуемые другими сервисами, чтобы имитировать поведение сервиса для проведения комплексного тестирования.

Сквозное тестирование

Сквозное тестирование играет важную роль, позволяя убедиться в том, что в компоненты выпуска распределенного приложения можно вносить изменения без нарушения работы потребителей. К тому же микросервисная архитектура представляет собой композицию из множества различных сервисов, при которой возможно подключение каждого приложения ко множеству сервисов в ансамбле организованного хаоса, требующего сложной стратегии тестирования. В данном разделе мы рассмотрим несколько технологий тестирования распределенных приложений, ориентированных на выполнение в облаке.

Сквозное тестирование фокусируется на проверке функционирования бизнес-особенностей приложения. В противоположность комплексному сквозное сконцентрировано на особенностях, тестируемых с точки зрения пользователя. Предположим, что таковой захотел зарегистрировать новую учетную запись. Это действие предполагает череду операций, которые пользователь может выполнить для успешной ее регистрации. В большинстве случаев каналом для этих действий становится пользовательский интерфейс, создающий события, взаимодействующие с серверными API для достижения результата предпринимаемых действий. Если проводить инвентаризацию каждого события, выдаваемого в результате действий пользователя, регистрирующего новую учетную запись, то череду событий можно превратить в сквозной тест.

В зависимости от той части системы, которая подвергается проверке, существует множество разновидностей сквозных тестов. Для сторонних разработчиков, создающих микросервисы, сквозной тест для регистрации новой учетной записи, скорее всего, станет, в зависимости от последовательности действий, составной частью API-взаимодействия между отдельными микросервисами. Для сквозной последовательности, которая организует бизнес-функцию, затрагивающую различные микросервисы, важную роль играет управление состоянием.

Тестирование распределенных систем

Обычно о состоянии заходит речь, когда рассматривается вопрос совместимости ПО. Нужно в точности знать о порядке совместного применения состояния и о том, как оно копируется между различными архитектурами при разработке приложений в распределенных системах. Одно из наших излюбленных объяснений состояния относится к временам зарождения вычислительной техники. Алан Тьюринг (Alan Turing) первым написал о состоянии вычисления в своей эпохальной статье On computable numbers, with an application to the Entscheidungsproblem[1]. В этой статье он дал описание состоянию, используя мысленный эксперимент, в котором преподносится теория способа определения машинной принадлежности числа к вычисляемым. Мысленный эксперимент Тьюринга ввел концепцию *машины Тьюринга* — воображаемого механического устройства, позволяющего вычислять числа, применяя таблицу состояний.

Описываемая Тьюрингом машина состоит из бесконечной, разбитой на квадраты ленты, содержащей символы, распространяющиеся только в двух направлениях. Аналогия, использованная Тьюрингом для описания своей машины, чем-то напоминала пишущую машинку. *Машине* позволялось прокручивать ленту либо вправо, либо влево, производя посимвольное сканирование. У машины также имелась таблица состояний, отображающая символ на условие, инструктирующее машину на предмет следующего действия.

Рассмотрим пример. Предположим, что исходное состояние машины Тьюринга — `A`. Первый символ, считанный машиной с ленты, — `0`. После сканирования машина ищет свою инструкцию из таблицы состояний для ее текущего состояния, которым по-прежнему является `A`. Инструкции дают описание условию: для считанного символа `0` нужно записать `1`. Перед перемещением к следующей позиции завершающей инструкцией станет изменение состояния с `A` на `B`. Теперь состояние `B` будет содержать другой набор инструкций, зависящий от символа, считанного на следующей позиции. Машина может опять считать со своей следующей позиции `0`, и поскольку пребывает в состоянии `B`, то по-прежнему запишет `1`, но на этот раз получает инструкцию переместиться *влево*, а не вправо по ленте.

Тьюринг сводит суть программирования поведения его машины к переключениям между инструкциями, проявляющими реакцию и изменяющимися на основе вводимых данных, в процессе чего выполняется вывод данных.

Когда речь заходит о состоянии в современных приложениях, обычно имеется в виду поле статуса в записи базы данных, представленное в виде столбца таблицы. Возьмем, к примеру, пользователя, регистрирующего на сайте новую учетную запись.

---

[1] On computable numbers, with an application to the Entscheidungsproblem. — Режим доступа: https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf. — London Mathematical Society, 1937.

Его статус перейдет из одного состояния в другое в зависимости от введенных данных. Если пользователь только что завершил регистрацию учетной записи путем отправки формы на веб-странице, то предоставленные поля формы обработаются и в базе данных будет сохранена запись с регистрационными данными пользователя. Одним из полей новой записи будет статус, и это поле может содержать одно из нескольких значений в любой отдельно взятый момент времени, которое дает описание текущему состоянию пользователя. Несколько примеров записей приведены в табл. 4.2.

Таблица 4.2. Пользовательские записи, содержащие поле статуса

first_name	last_name	Email	Status
Bob	Dylan	bdylan@example.com	PENDING
Taylor	Swift	tswift@info.com	CONFIRMED
Tracy	Chapman	tchapman@test.com	ARCHIVED
Bruno	Mars	bmars@example.com	INACTIVE

В табл. 4.2 показаны несколько строк таблицы базы данных, содержащей пользовательские записи. У каждого пользователя, фигурирующего в таблице, в столбце статуса имеется свое значение. Для пользователя по имени *Bob Dylan* показано, что его текущий статус имеет значение `PENDING` (рассматриваемый). Теперь поведение приложения с пользовательской точки зрения изменится в зависимости от значения столбца статуса. Статус `PENDING` подразумевает, что прежде, чем пользователь сможет войти в приложение, ему придется подтвердить указанный адрес электронной почты. После этого действия статус перейдет к значению `CONFIRMED` (подтвержденный), что позволит пользователю войти в приложение и получить доступ к другим его функциям. Если пользовательский статус получит значение `ARCHIVED` (заархивированный) или `INACTIVE` (неактивный), то пользователь может не войти в приложение, в зависимости от требований последнего. Такой порядок действий зачастую применяется в приложениях, реализующих форму аутентификации пользователя.

Проблема, возникающая при создании распределенных систем, а точнее, микросервисов, заключается в следующем: рабочие процессы, подобные рассмотренным выше, не будут изолированы границами одного приложения. Когда архитектура монолитного приложения разбивается на множество отдельных сервисов, сетевые разделы разобьют на части процессы, которые в противном случае управляли бы данными в контексте одной транзакции. А что это значит для создателя приложения?

Основная проблема распределенных приложений заключается в том, что статус должен иметь глобальную область видимости для отдельно взятых машин, использующих информацию, обмен которой (для сообщения о состоянии) ведется по

сети. Для одного монолитного приложения, подключенного к единой базе данных, такой как реляционная БД типа MySQL, состояние может постоянно копироваться между отдельными экземплярами базы, обеспечивая его высокую доступность. Сохраняя запись, содержащую состояние сразу на нескольких машинах, мы пользуемся преимуществом возможности масштабирования и обработки возрастающего объема трафика.

Недостаток распределенного состояния лучше всего свести к следующему вопросу: как при каждом считывании скопированной записи с пула машин можно будет убедиться в ее согласованности с состоянием записи, хранящейся на других машинах этого пула?

Распределенное состояние и убежденность в согласованности — серьезнейшие проблемы в компьютерной науке. В границах одной машины и одного процесса довольно просто обеспечить согласованность данных, поскольку при считывании из памяти мы имеем единственный адрес со ссылкой на место хранения данных. Допустим, один из потоков блокирует ссылку на адрес с целью изменить состояние записи; тогда другие потоки перед выполнением данного изменения, на которое указывает эта ссылка, будут видеть блокировку. Если это единственный источник информации о состоянии записи, к нему может быть открыт глобальный доступ без риска несогласованности состояния записи. Ситуация изменяется, когда состояние копируется с выходом за границы одной машины и все копии должны совместно как подвергаться изменениям, так и считываться. Если одна из копий отличается от других, то состояние записи считается несогласованным, что может повлиять на поведение приложения, переключающегося между разными состояниями, сохраненными в различных разделах.

При тестировании приложений, ориентированных на работу в облаке и распределенных по отдельным микросервисам, важно понять, как создавать сквозные тесты, проверяющие согласованность данных. Потенциальная согласованность — лучшее, на что можно рассчитывать при совместном использовании состояния с выходом за границы отдельных микросервисов. Мы поговорим о конструировании потенциальной согласованности, когда будем обсуждать решения по интеграции данных в главе 12. Основной вопрос сводится к разработке условий, позволяющих убедиться в неизменной потенциальной согласованности состояния, возможно не обладающего согласованностью в отдельно взятый конкретный момент времени, но всегда в конечном итоге согласованного без ручного вмешательства. Когда требуется абсолютная согласованность, нельзя допускать совместное использование состояния микросервисами. Это простое правило позволяет узнать, когда именно следует избегать распределенных транзакций.

Еще одна проблема, имеющая отношение к архитектуре микросервисов, заключается в подходе к проверке взаимодействия между различными приложениями. Для этой разновидности тестирования от нас обычно требуется выполнение сквозных

тестов в интеграционной среде, где все приложения и зависимости подстраиваются под работу в среде выполнения, отображая модель развертывания при эксплуатации. Затраты времени, связанные с подстройкой завершенной сквозной среды под комплексное тестирование отдельно взятого микросервиса, могут быть совершенно неприемлемыми.

Для разработчика может быть очень полезным получить возможность выполнять комплексные тесты со множеством зависимостей от микросервисов в максимально сжатые сроки, создавая более жесткие циклы обратной связи с одновременным ограничением конфликтов ресурсов в общей среде сборки. Один из популярных методов тестирования, помогающий разрешить некоторые из этих довольно острых вопросов и быстро приобретающий форму стандарта для тестирования микросервисов, называется *тестированием контрактов, ориентированных на потребителя* (Consumer-Driven Contract Testing).

Тестирование контрактов, ориентированных на потребителя

Тестирование контрактов, ориентированных на потребителя (Consumer-Driven Contract Testing, CDC-T), впервые было введено Яном Робинсоном (Ian Robinson) в 2006 году в качестве практики использования опубликованных контрактов для утверждений и поддержки ожиданий между потребителями и производителями наряду с предоставлением слабой связанности между сервисами. В статье, изначально появившейся на сайте Мартина Фаулера (Martin Fowler), Робинсон дал описание развития сервиса в сервис-ориентированной архитектуре (service-oriented architecture, SOA) через применение контрактов, ориентированных на потребителя (consumer-driven contracts)[1]. В последующие годы практика и шаблоны контрактов, ориентированных на потребителя, введенные Робинсоном, были приняты к использованию для тестирования ожиданий между микросервисами.

Основной замысел в CDC-T — разрешение производителю и потребителю в архитектуре микросервисов публиковать (и выстраивать) заглушки в виде контрактов, ориентированных на потребителя. Контракт описывает интерфейс вызовов сервиса. Использование микросервисами общих библиотек во время выполнения не предполагается. Сервисы могут совместно применять библиотеки в процессе комплексного тестирования в том случае, если между микросервисами имеется циклическая зависимость. Производитель сначала публикует заглушки, определяя контракты и комплексный тест, использующий контракт. Потребители могут имитировать производителя, загружая созданные по версии производителя заглушки из совместно применяемого места, подобного хранилищу компонентов. Производители совместно используют заглушки, созданные через определение

[1] *Robinson Ian.* Consumer-Driven Contracts: A Service Evolution Pattern. — Режим доступа: https://martinfowler.com/articles/consumerDrivenContracts.html. — 2006.

контракта, но не допускают совместного применения типов или клиентских библиотек. CDC-T-тестирование стремится скрыть подробности реализации API производителя.

Spring Cloud Contract

Spring Cloud Contract — проект с открытым кодом, входящий в ассортимент Spring и предоставляющий компоненты фреймворка, использующие разновидность контрактов, ориентированных на потребителя. Такие контракты зачастую используются для компонентов распределенных приложений, подвергаемых комплексному тестированию, таких как REST API, и обменивающихся сообщениями между микросервисами. Spring Cloud Contract поддерживает возможность публикации, моделирования и имитации удаленных сервисов с помощью заглушек.

Например, мы собираемся создать контракт, ориентированный на потребителя, тестирующий взаимодействие между двумя микросервисами. Проверке подвергнем два сервиса: `Account Microservice` и `User Microservice`. Первый обслуживает коллекцию учетных записей, принадлежащих пользователю. Для пользователей, владеющих этими учетными записями, `User Microservice` будет действовать в качестве хранилища записей наряду с обработкой аутентификации.

Взаимодействие двух микросервисов не отличается особой сложностью. `Account Microservice` извлекает учетные записи пользователя, входящего в данный момент на сайт. Для этого данный микросервис контактирует с `User Microservice` для получения имени аутентифицированного пользователя из сессии HTTP-запросов. Имя, возвращенное микросервисом `User Microservice`, послужит для извлечения учетных записей для аутентифицированного пользователя в `Account Microservice`. Взаимоотношения между `Account Microservice` и `User Microservice` показаны на рис. 4.1.

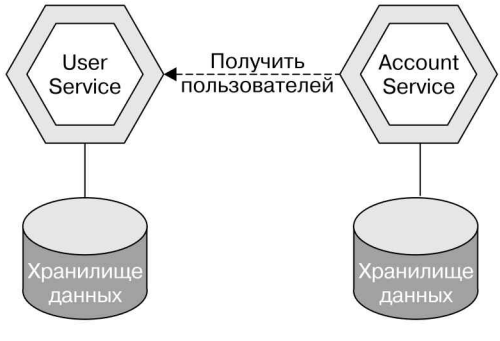


Рис. 4.1. Account Service — потребитель User Service

А теперь исследуем исходный код для двух проектов Spring Boot, начиная с `User Microservice` (пример 4.12).

Пример 4.12. Классы приложения Spring Boot по имени User Microservice

```
.
├── ./src/main
│   ├── java
│   │   └── demo
│   │       ├── UserApplication.java
│   │       ├── config
│   │       │   └── JpaConfiguration.java
│   │       ├── data
│   │       │   └── BaseEntity.java
│   │       └── user
│   │           ├── AuthService.java
│   │           ├── User.java
│   │           ├── UserController.java
│   │           ├── UserRepository.java
│   │           └── UserService.java
│   └── resources
│       ├── application.yml
│       ├── bootstrap.yml
│       ├── data-e2e.sql
│       ├── data-mysql.sql
│       └── schema-mysql.sql
└── ./src/main/test
    └── ...
```

В примере 4.12 показан исходный код приложения Spring Boot для `User Microservice`. Структура каталогов здесь сконцентрирована на файлах классов приложения в каталоге `src/main/demo`. Этот пакет содержит код приложения для `User Microservice`. Поскольку нам уже известно, как работает обычное приложение Spring Boot, то наши намерения сводятся лишь к исследованию основных функций данного сервиса, который будет проверяться с помощью контракта, ориентированного на потребителя.

Пакет `user` этого микросервиса содержит функции, которые интересно будет протестировать. Здесь мы найдем два компонента сервиса, содержащих бизнес-логику приложения: `AuthService` и `UserService`. Компонент `UserService` включает метод, служащий для получения аутентифицированного пользователя сессии. Компонент `AuthService` будет применяться для получения сессионной информации об аутентифицированном пользователе. Затем компонент `UserService` поведет поиск конкретных данных из `UserRepository`, применяя в качестве ключа поиска идентификатор, возвращенный компонентом `AuthService`.

В примере 4.13 показан метод `getUserByPrincipal`, возвращающий экземпляр класса `User`. Этот класс — сущность той области, в которой содержатся поля, дающие описание аутентифицированному участнику `Principal`. В рассматриваемом методе используется свойство `name` класса `Principal`, которое применяется для поиска записи `User` из `UserRepository`. Для данного примера с целью обмена информацией с базой данных служит хранилище Spring Data JPA. (Более подробное исследование Spring Data JPA будет представлено в главе 9.)

Пример 4.13. Класс UserService извлекает подробности, относящиеся к пользователю-участнику

```
package demo.user;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.security.Principal;
import java.util.Optional;

@Service
public class UserService {

 private final UserRepository userRepository;

 @Autowired
 public UserService(UserRepository userRepository) {
  this.userRepository = userRepository;
 }

 public User getUserByPrincipal(Principal principal) {
  (1)
  return Optional.ofNullable(principal)
   .map(p -> userRepository.findUserByUsername(p.getName())).orElse(null);
 }
}
```

(1) Извлечение записи `User` с помощью поля `name` параметра `Principal`.

Этот пример нельзя признать полностью реалистичным в том смысле, что он иллюстрирует обычный порядок действий в микросервисах, который нуждается в реализации безопасности на уровне пользователя, но не реализует поставщика информации по аутентификации. Компонент `AuthService` в данном примере содержит всего лишь бутафорский метод, извлекающий пустую реализацию java.security.Principal. В настоящем приложении основная нагрузка в этом смысле была бы возложена на Spring Security.

Исследуем класс `UserController`, в котором определяется REST API, применяемый `Account Service` для удаленного извлечения сведений об аутентифицированном пользователе по технологии протокола HTTP.

Класс `UserController` показан в примере 4.14. В этом классе определяется простой контроллер Spring MVC, отображаемый на конечную точку `/uaa/v1`. Здесь мы располагаем единственным методом контроллера, имеющим название `me`. Данный метод предпринимает попытку получить участника `Principal` для сессии, который содержится в контексте HTTP-запроса. Первым делом нужно получить аутентифицированный `Principal` из `AuthService`. Если `Principal` доступен и не имеет значения `null`, то метод контроллера вызовет `UserService`, чтобы извлечь

запись User из базы данных. И наконец, если извлечь User или Principal не представляется возможным, то метод возвращает ResponseEntity с HTTP-статусом UNAUTHORIZED.

Пример 4.14. Класс UserController, в котором определяется REST-контроллер приложения

```
package demo.user;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.security.Principal;
import java.util.Optional;

@RestController
@RequestMapping(path = "/uaa/v1")
public class UserController {

 private UserService userService;

 private AuthService authService;

 @Autowired
 public UserController(UserService userService, AuthService authService) {
  this.userService = userService;
  this.authService = authService;
 }

 (1)
 @RequestMapping(path = "/me")
 public ResponseEntity<User> me(Principal principal) throws Exception {
  return Optional.ofNullable(authService.getAuthenticatedUser(principal)) (2)
   .map(p -> ResponseEntity.ok().body(userService.getUserByPrincipal(p))) (3)
   .orElse(new ResponseEntity<User>(HttpStatus.UNAUTHORIZED)); (4)
 }
}
```

(1) Описание @RequestMapping для GET-запроса к /uaa/v1/me.

(2) Получение аутентифицированного пользователя Principal из AuthService.

(3) Если Principal не имеет значения null, выполняется поиск UserService.

(4) Если Principal не имеет значения null, то возвращается HTTP-ошибка, свидетельствующая об отсутствии авторизации.

Далее мы собираемся создать тест контракта, ориентированного на потребителя, для того метода UserController, который отображается на конечную точку /uaa/v1/me. Чтобы опубликовать спецификацию REST API, принадлежащего User Microservice, мы воспользуемся *Spring Cloud Contract*. Эта спецификация публикуется как компонент Maven, который может быть извлечен другими микро-

сервисами для имитации веб-сервера, эмулирующего тестируемое поведение `User Microservice`. Таким образом, `User Microservice` станет публиковать только спецификацию, прошедшую все блочные тесты, написанные для `UserController`. Применив данный подход, все потребители `User Microservice` получат в ходе проверки возможность провести комплексные тесты в отношении имитируемого веб-сервера, как если бы он реально существовал!

Исследуем исходники теста, где будет определен контракт, ориентированный на потребителя для `User Microservice`. Структура каталогов для наших пользовательских тестов показана в примере 4.15.

Пример 4.15. Корневой каталог исходников теста User Microservice

```
.
├── ./src/main/test/java
│   └── demo
│       ├── UserServiceBase.java
│       └── user
│           ├── UserControllerTest.java
│           ├── UserRepositoryTest.java
│           └── UserTests.java
└── resources
    ├── contracts
    │   └── shouldReturnUser.groovy *
    ├── data-h2.sql
    │   └── demo
    │       └── user
    │           └── user.json
```

Стратегии тестирования, используемые в этом примере как для блочных, так и для комплексных тестов, аналогичны тем стратегиям, которые применялись ранее в примерах данной главы. На сей раз основное внимание будет уделяться способам определения заглушки Spring Cloud Contract для `User Microservice`. В создаваемом определении заглушки будет установлен ориентированный на потребителя тест для извлечения аутентифицировавшихся пользователей.

Определения заглушек в Spring Cloud Contract используются для описания поведения имитируемых (удаленных) сервисов в тестах, ориентированных на потребителя. Такие варианты поведения называются *ожиданиями*, поскольку являются набором ожиданий со стороны производителя сервиса, касающихся тестируемого поведения методов REST API, открытых для потребителей.

Каждый файл определения заглушки будет охватывать один метод класса `Controller`. Поскольку у нас в `UserController` имеется только один метод, для проекта будет создаваться только один файл определения заглушки — `shouldReturnUser.groovy`. Определения заглушек для Spring Cloud Contract создаются с использованием обычного предметно-ориентированного языка Spring Cloud Contract (Groovy DSL). Изначально наши определения заглушек расположены в каталоге `resource` корневого каталога исходников тестов, который обычно находится в `src/main/test/java/resources`. Посмотрим на это определение (пример 4.16).

Пример 4.16. Определение заглушки в shouldReturnUser.groovy

```
package contracts

org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'
        url '/uaa/v1/me'
        headers {
            header('Content-Type': consumer(regex('application/*json*')))
        }
    }
    response {
        status 200
        body([
                username    : value(producer(regex('[A-Za-z0-9]+'))),
                firstName   : value(producer(regex('[A-Za-z]+'))),
                lastName    : value(producer(regex('[A-Za-z]+'))),
                //@formatter:off
                email       : value(producer(
                        regex('[A-Za-z0-9]+\\@[A-Za-z0-9]+\\.[A-Za-z]+'))),
                //@formatter:on
                createdAt   : value(producer(regex('[0-9]+'))),
                lastModified: value(producer(regex('[0-9]+'))),
                id          : value(producer(regex('[0-9]+')))
        ])
        headers {
            header('Content-Type': value(
            producer('application/json;charset=UTF-8'),
                  consumer('application/json;charset=UTF-8'))
            )
        }
    }
}
```

Если синтаксис Groovy DSL вам незнаком, то не стоит волноваться! DSL-язык для определения заглушки, используемый в Spring Cloud Contract, довольно прост и легок в освоении. Так как здесь мы не намерены слишком сильно вдаваться в подробности Groovy DSL, вы можете найти их в онлайн-документации в справочном руководстве проекта Spring Cloud Contract.

Установите компоненты `user-microservice` в свое локальное хранилище компонентов Maven. Наш `user-microservice` настроен на публикацию не одного, а двух элементов в каждой команде `mvn install`: самого компонента и компонента, содержащего определение контракта. Второй является тем, от чего будет зависеть наш API потребителя, `account-microservice`. В примере 4.17 показана конфигурация для сборки Maven.

Пример 4.17. Конфигурация в сборке Maven для производителя

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
        <goals>
          <goal>repackage</goal>
        </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-plugin</artifactId>
      <version>${spring-cloud-contract.version}</version>
      <extensions>true</extensions>
      <configuration>
        <baseClassForTests>demo.UserServiceBase</baseClassForTests>
        <basePackageForTests>demo</basePackageForTests>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Для подтверждения того, что и компонент, и заглушка для него были развернуты, изучите содержимое своего локального хранилища Maven (пример 4.18).

Пример 4.18. Образец файлов компонента, который должен быть установлен в вашем локальном хранилище Maven

```
.
├── _remote.repositories
├── maven-metadata-local.xml
├── user-microservice-1.0.0-SNAPSHOT-stubs.jar
├── user-microservice-1.0.0-SNAPSHOT.jar
└── user-microservice-1.0.0-SNAPSHOT.pom

0 directories, 5 files
```

При развернутых заглушках в нашем тесте потребителя для установки соответствующего контракту REST API, способного интегрироваться с нашим потребителем, обычно используется средство запуска заглушки. API является настоящим, он запускается на фактически имеющемся порте и производит реальные значения, соответствующие определениям спецификации. Запуск этого теста — обычная и относительно малозатратная операция, поскольку мы не имеем дела с фактическим, реально действующим сервисом, подключенным к связующему ПО и базам данных (пример 4.19).

Пример 4.19. Блок Account Service тестирует контракт, ориентированный на потребителя, принадлежащий User Service

```
package demo;

import demo.user.User;
import demo.user.UserService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

//@formatter:off
import org.springframework.cloud.contract.stubrunner
        .spring.AutoConfigureStubRunner;
//@formatter:on
import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.Assertions.assertThat;
import static org.springframework.boot.test.context.SpringBootTest.*;

//@formatter:off
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
@AutoConfigureStubRunner(
        ids = { "cnj:user-microservice:+:stubs:8081" },
        workOffline = true) ❶
//@formatter:on
public class ConsumerDrivenTests {

 @Autowired
 private UserService service;  ❷

 @Test
 public void shouldReturnAuthenticatedUser() {
  User actual = service.getAuthenticatedUser();

  assertThat(actual).isNotNull();
  assertThat(actual.getUsername()).matches("[A-Za-z0-9]+");
  assertThat(actual.getFirstName()).matches("[A-Za-z]+");
  assertThat(actual.getLastName()).matches("[A-Za-z]+");
  assertThat(actual.getEmail()).matches(
   "[A-Za-z0-9]+\\@[A-Za-z0-9]+\\.[A-Za-z]+");
 }
}
```

❶ Аннотация `@AutoConfigureStubRunner` указывает, где нужно искать компонент заглушки (в виде координат Maven) и на каком порте средство запуска заглушки должно использовать сервис-имитатор.

❷ Наш блочный тест внедряет *фактический* `UserService`. Этот сервис не имеет никакой имитации, кроме ответов, получаемых после совершения HTTP-вызовов в адрес сервиса-заглушки!

Резюме

В соответствии с исследованиями, проведенными в данной главе, стратегии проверки микросервисов могут совершенно отличаться от заурядных блочных и комплексных тестирований, используемых в других разновидностях архитектур веб-приложений. Spring предоставляет весьма эффективный набор компонентов фреймворка и проектов для тестирования приложений Spring Boot.

Spring долгое время обеспечивал солидную поддержку компонентов комплексного тестирования отдельно взятых приложений. В Spring Boot эта поддержка была расширена, что упростило проведение конфигурирования, распределенного по уровням функциональности. В Spring Cloud произошло дальнейшее расширение, дошедшее до тестирования контрактов, ориентированных на потребителя (CDC-T). В этой главе была рассмотрена возможность использования заглушек конечной точки REST с помощью Spring Cloud Contract, хотя стоит упомянуть и о том, что в последнем поддерживаются и конечные точки с работой, основанной на выдаче сообщений.

Тестирование — ключевой аспект непрерывного развертывания, при котором компоненты ПО проходят по конвейеру, что в конечном счете приводит к получению компонентов, готовых к запуску в производство. По мере продвижения ПО по конвейеру непрерывного развертывания тесты становятся все более исчерпывающими и медленными. В идеале блочными и комплексными тестами за весьма короткий период времени охватываются 80–90 % случаев применения, подтверждая при этом, что код, возможно, приобрел состояние, допустимое для его ввода в эксплуатацию.

В данной главе мы *обошли* вниманием блочные тесты или другие, более охватывающие, но менее распространенные виды проверки, проводимые после комплексных тестов. В главе 2 рассматривалось использование клиента Cloud Foundry Java для автоматизации развертываний в Cloud Foundry. В системе, ориентированной на работу в облаке, автоматизировано *все*, включая развертывания. Это ключ к проведению тестов на отказ и к достижению согласованности.

5 Миграция приложения в облако

Итак, вы располагаете великолепной новой распределенной средой выполнения, потенциалом новых неограниченных возможностей и массой существующих приложений, и что теперь?

Контракт

Платформа Cloud Foundry нацелена на повышение скорости за счет сокращения или по крайней мере согласования неисправностей, связанных с развертыванием приложений, и управления ими. Cloud Foundry — идеальное место для запуска интерактивных сервисов и приложений на основе веб-технологий, объединения сервисов и выполнения вспомогательной обработки информации.

Платформа Cloud Foundry (https://cloudfoundry.org/) оптимизирована для непрерывного развертывания веб-приложений и сервисов путем выдачи предположений о форме запускаемых на ней приложений. *Вводимыми* в Cloud Foundry данными являются приложения: двоичный код Java (`.jar` или, если настаиваете, `.war`), приложения Ruby on Rails, приложения Node.js и т. д. Cloud Foundry предоставляет запускаемым на ней приложениям вполне очевидные эксплуатационные преимущества (агрегирование регистрационных записей, маршрутизацию, самовосстановление работоспособности, динамическое масштабирование как на расширение, так и на сужение, обеспечение безопасности и т. д.). Между платформой и приложениями существует подразумеваемый контракт, который позволяет ей выполнять обещания в отношении запускаемых на ней приложений.

Одним приложениям никогда не суждено соответствовать этому контракту. Другие же могут иметь такую возможность, хотя и с некоторыми незначительными корректировками. Здесь мы рассмотрим потенциальные приемы незначительной реструктуризации для принудительного использования устаревших приложений на платформе Cloud Foundry.

В данном случае цель заключается не в создании приложений, специально *оптимизированных* под работу в облачной среде, а в перемещении уже существующих приложений в облако, чтобы сократить эксплуатационное пространство организации и повысить универсализацию. Когда приложение развертывается на Cloud Foundry, оно по крайней мере не теряет своих прежних качеств, но тех из них, о которых стоило беспокоиться, становится на одно меньше, поскольку теперь нет *разнообразного развертывания*, то есть такого, что всякий раз совершенно неоправданно проводится по-разному. Чем меньше подобных качеств, тем лучше.

Подобную разновидность миграции приложений, в отличие от создания *приложений, оптимизированных для работы в облаке*, мы называем *миграцией приложений*. Большое место в нынешних обсуждениях в блогах Spring (https://spring.io/blog) и Pivotal (https://content.pivotal.io/blog) занимают вопросы разработки приложений облачной среды, то есть тех, которые и живут, и «дышат» в облаке («вдыхают» эту среду и «выдыхают» ее по мере необходимости и сообразно своим потребностям в ресурсах) и полностью используют платформу. Данная тема, несомненно, будет в центре внимания всех других глав этой книги!

Поведение приложения, в широком смысле слова, складывается из суммы среды его выполнения и кода. В данной главе мы рассмотрим стратегии для перемещения прежних Java-приложений из какой-либо среды, в которой они и пребывают. Мы изучим схемы, привычные для разработчиков приложений перед приходом в среду облачных вычислений, и проведем разбор схем, наилучшим образом отвечающих выполнению приложений в облаке. Кроме того, обсудим ряд особых решений и сопутствующий код.

Миграция сред приложения

Существует ряд качеств, присущих всем приложениям, и эти качества (например, маршрутизация RAM и DNS) могут быть сконфигурированы непосредственно через имеющийся в Cloud Foundry инструмент интерфейса командной строки (CLI) `cf`, через различные панели управления или в принадлежащем приложению файле манифеста `manifest.yml`. Если ваше приложение имеет подстраиваемый характер и просто нуждается в большем объеме оперативной памяти (RAM) или пользовательском DNS-маршруте, то вы получите все необходимые рычаги управления, которыми в Cloud Foundry могут пользоваться все приложения.

Оригинальные сборочные пакеты (buildpacks)

Но иногда все складывается не так-то просто. Ваше приложение может запускаться в любом количестве разнообразных сред, а платформа Cloud Foundry выдвигает весьма конкретные предположения о тех средах, в которых запускаются ее приложения. Эти предположения закодированы в какой-то степени в самой платформе и в сборочных пакетах *buildpacks*, позаимствованных у Heroku. Cloud Foundry

и Heroku все равно, какую разновидность приложения они запускают. Им важны Linux-контейнеры, в конечном счете являющиеся процессами операционной системы. В buildpacks платформа Cloud Foundry узнает, что нужно делать с учетом Java-файлов `.jar`, Rails-приложения, Java-файлов `.war`, Node.js-приложения и так далее и как превратить все в контейнер, который платформа может рассматривать в качестве процесса. Сборочный пакет представляет собой набор *функций обратного вызова* — сценариев оболочки, откликающихся на общеизвестные вызовы, которые в ходе выполнения будут использоваться для завершающего создания запускаемого Linux-контейнера. Этот процесс называется *продвижением* (staging).

В Cloud Foundry предоставляется множество оригинальных *системных сборочных пакетов buildpacks* (http://docs.pivotal.io/pivotalcf/2-0/buildpacks/). Эти пакеты могут быть подстроены или даже полностью заменены. Если нужно запустить приложение, для которого нет оригинального сборочного пакета (предоставляемого сообществом Cloud Foundry (https://github.com/cloudfoundry-community/cf-docs-contrib/wiki/Buildpacks), Heroku (https://www.heroku.com/) или Pivotal (https://pivotal.io/)), то по крайней мере будет нетрудно разработать и развернуть собственный (http://docs.pivotal.io/pivotalcf/2-0/buildpacks/custom.html). Существуют сборочные пакеты для всех разновидностей сред и приложений, включая и так называемый *Sourcey* (https://github.com/oetiker/sourcey-buildpack), просто компилирующий ваш собственный код! (Признайтесь: вы знаете по крайней мере об одном устаревшем бизнес-приложении где-либо в недрах вашей организации, которому требуется компилятор языка C и некое жертвоприношение, не так ли?)

Заказные (или подстраиваемые) сборочные пакеты

Эти сборочные пакеты служат для предоставления весьма рациональных исходных установок, сохраняя притом достаточную гибкость. В качестве примера можно отметить, что исходный сборочный пакет Java/JVM поддерживает компоненты `.war` (https://github.com/cloudfoundry/java-buildpack) (которые будут запускаться внутри актуальной версии Apache Tomcat), созданные в стиле Spring Boot выполняемые компоненты `.jar`, приложения веб-среды Play, Grails-приложения и др. Кроме того, он поддерживает подключение множества широко известных Java-агентов, таких как New Relic.

Если сборочные пакеты, имеющиеся в системе, вам не подходят и нужно применить что-либо иное, то следует всего лишь указать платформе Cloud Foundry, где искать код для сборочного пакета, воспользовавшись ключом `-b` в команде `cf push`, как показано в примере 5.1.

Пример 5.1. Основной сервлет Java EE

```
cf push -b https://github.com/a/custom-buildpack.git#my-branch custom-app
```

В качестве альтернативы можно указать сборочный пакет в файле `manifest.yml`, сопровождающем приложение. Предположим, у нас имеется приложение Ja-

va EE, которое по сложившимся обстоятельствам было развернуто с помощью IBM-средства WebSphere. IBM поддерживает весьма эффективный сборочный пакет WebSphere Liberty. Чтобы продемонстрировать это, допустим, что нужно развернуть и запустить основной `Servlet` (пример 5.2). (В этот раз проигнорируем наличие возможности запуска Servlet-компонентов в приложениях Spring Boot, как показано в разделе «Использование API Servlet в приложениях Spring Boot» приложения.)

Пример 5.2. Простейший Java EE Servlet

```
package demo;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/hi")
public class DemoApplication extends HttpServlet {

 protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {

  response.setContentType("text/html");
  response.getWriter().print("<html><body><h3>Hello Cloud</h3></body></html>");
 }
}
```

Чтобы запустить код, мы указали сборочный пакет WebSphere Liberty в файле манифеста приложения, показанном в примере 5.3.

Пример 5.3. Файл manifest.yml для нашего приложения WebSphere Liberty

```
---
applications:
- name: wsl-demo
  memory: 1024M
  buildpack: https://github.com/cloudfoundry/ibm-websphere-liberty-buildpack.git
  instances: 1
  host: wsl-demo-${random-word}
  path: target/buildpacks.war
  env:
    SPRING_PROFILES_ACTIVE: cloud
    DEBUG: "true"
    debug: "true"
    IBM_JVM_LICENSE: L-JWOD-9SYNCP
    IBM_LIBERTY_LICENSE: L-MCAO-9SYMVC
```

Некоторые пакеты поддаются настройке. Сборочный пакет Java (https://github.com/cloudfoundry/java-buildpack), изначально разработанный специалистами, создавшими платформу Heroku, а затем *радикально* расширенный командами Spring и Cloud Foundry, поддерживает настройки через переменные среды. Этот пакет предоставляет исходную конфигурацию в каталоге `config` для различных аспектов своего поведения. Заданное в предоставляемом файле конфигурации поведение можно переопределить, предоставив переменную среды (имеющую префикс `JBP_CONFIG_`) с таким же именем, как у файла конфигурации без расширения `.yml`. Тем самым, позаимствовав пример из замечательной документации (https://github.com/cloudfoundry/java-buildpack/blob/master/README.md), при желании переопределить версию JRE и конфигурацию памяти (которая в сборочном пакете находится в файле `config/open_jdk_jre.yml`) можно сделать то, что показано в примере 5.4.

Пример 5.4. Конфигурирование JDK и JRE, имеющихся в сборочном пакете Java

```
cf set-env custom-app JBP_CONFIG_OPEN_JDK_JRE \
    '[jre: {version: 1.7.0_+}, memory_calculator: {memory_heuristics:
    {heap: 85, stack: 10}}]'
```

Приложения в контейнере

Приложения в мире Java, разработанные для сервера приложений J2EE/Java EE, *весьма трудно* поддаются миграции за пределы этого сервера. Приложения Java EE, при всей своей хваленой переносимости, используют загрузчики классов, ведущие себя непоследовательно, предлагая различные подсистемы, которые сами зачастую требуют применения собственных конфигурационных файлов, и для заполнения множества пробелов они довольно часто предлагают сервер приложения, то есть конкретные API. Если ваше приложение совершенно не поддается переработке и различные способы воздействия, рассмотренные выше, не обеспечивают достаточными механизмами управления миграцией, то надежда все же есть!

Прежде чем обратиться к этому последнему прибежищу, все же поищите решение в использовании сборочных пакетов, предлагаемых сообществом. Существуют сборочные пакеты, поддерживающие, к примеру, IBM-продукт WebSphere (с участием IBM, поскольку у данной компании есть PaaS на основе Cloud Foundry!) и RedHat-продукт WildFly.

В Cloud Foundry также поддерживается запуск приложений в контейнере (например, Docker) (https://docs.pivotal.io/pivotalcf/1-10/adminguide/docker.html). Применение этой технологии может стать альтернативой, если вы уже получили приложение в контейнере и хотите лишь развернуть его и управлять его работой с помощью той же цепочки инструментов, что и для всех других приложений. Мы не рекомендуем использовать такой подход в силу разных причин.

- Поверхность рабочей зоны для контейнера на Docker-основе, который функционирует на Diego, остается несколько выше, чем имеющаяся у приложения сборочного пакета, так как Docker позволяет пользователям указывать содержимое их корневой файловой системы целиком. Приложение сборочного пакета работает на доверенной корневой файловой системе.
- Сборка контейнера — дополнительные действия, возлагаемые на создателей и специалистов, сопровождающих работу приложения.
- Для контейнерных файловых систем труднее даются внесение исправлений и обновления, поскольку для вступления изменений в силу требуется повторная сборка. В случае необходимости нет простого способа централизованной переустановки и повторного развертывания каждого отдельно взятого контейнера.

Для непрозрачных контейнеризированных приложений мы изучили целую гамму операций, от обычного конфигурирования до переопределения сборочных пакетов, специфичных как для приложений, так и для среды его выполнения. Любая попытка перенести приложение в облако начинается в этом порядке, и сначала идут простейшие настройки. Цель — свести их объем к минимуму и позволить Cloud Foundry проделать как можно больше работы.

Незначительная реструктуризация для перемещения вашего приложения в облако

В последнем разделе мы рассмотрели действия, которые можно предпринять для перемещения целого приложения из существующей для него среды в новую без внесения изменения в код. Мы изучили приемы перемещения простых приложений. Мы увидели, что существуют способы для всех, кроме виртуализированных приложений, и переместили их в Cloud Foundry, но при этом не рассматривали приемы, указывающие приложениям на потребляемые ими опорные сервисы (базы данных очереди сообщений и т. д.). Чтобы не усложнять ситуацию, мы также игнорировали существование отдельных классов приложений, которые можно было бы заставить работать исключительно на Cloud Foundry с незначительными, иногда довольно нудными, но зачастую простыми изменениями.

Использование при реструктуризации комплексного тестового набора всегда окупится сторицей. Мы понимаем, что у некоторых прежних приложений не будет такого набора. Действовать нужно *осторожно*, но быстро!

В основном мы рассмотрим *незначительные* корректировки, которые можно вносить для достижения работоспособности приложения, надеясь при этом, что риск нанесения ему вреда будет минимальным. Понятно, что без тестового набора

модульный код будет легче поддаваться изменениям. По иронии, у приложений, больше всего нуждающихся в комплексном тестовом наборе (а это крупные монолитные устаревшие приложения), его как раз таки и нет. При *наличии* набора вы можете не располагать тестами на отказ, проверяющими развертывание приложения и его подключение к связанным с ним сервисам. Создание данного набора дается, как правило, труднее, но он был бы полезен именно при выполнении таких действий, как миграция устаревших приложений в новую среду.

Обращение к опорным сервисам

Опорными называются сервисы, потребляемые приложением (базы данных, очереди сообщений, сервисы электронной почты и т. д.). Приложения Cloud Foundry используют опорные сервисы, выискивая их местонахождение и учетные данные в значении переменной среды `VCAP_SERVICES`. Отличительная черта данного подхода — его простота: код на любом языке может забрать переменную из среды и провести разбор кода в формате JSON для извлечения таких данных, как хосты сервисов, порты и учетные записи.

Приложения, зависящие от опорных сервисов, обслуживаемых Cloud Foundry, способны заставить платформу создать эти сервисы по требованию. Создание сервиса может быть также названо *предоставлением услуг* (provisioning). Точное значение зависит от контекста; для всех сервисов электронной почты это может означать предоставление для электронной почты нового имени пользователя и пароля. Для опорного сервиса MongoDB — создание новой базы данных Mongo и предоставление доступа к этому экземпляру MongoDB. Жизненный цикл опорного сервиса модулируется экземпляром сервис-брокера платформы Cloud Foundry. Такие брокеры представляют собой REST API, с которыми платформа сотрудничает в целях управления опорными сервисами. (Более подробно сервис-брокеры будут рассмотрены в главе 14.)

После регистрации на Cloud Foundry брокер становится доступным с помощью команды `cf market place` и может быть предоставлен по требованию благодаря команде `cf create-service`. Рассмотрим гипотетический пример создания сервиса. Первый параметр, `mongo`, — название сервиса. Здесь мы воспользуемся некоторой обобщенной формой, но с тем же успехом это может быть `New Relic`, или `MongoHub`, или `ElephantSQL`, или `SendGrid` и т. д. Второй параметр — название плана, представляющего собой ожидаемые уровень и качество сервиса. Иногда более высокие уровни сервиса подразумевают и более высокие цены. Третьим параметром выступает вышеуказанное логическое имя (пример 5.5).

Пример 5.5. Создание нового опорного сервиса в Cloud Foundry

```
cf create-service mongo free my-mongo
```

Создать сервис-брокер нетрудно, но объем работы может быть больше необходимого. Если вашему приложению нужно обратиться к уже существующему

статичному сервису, перемещение которого маловероятно, и вам требуется всего лишь указать ему на ваше приложение, то пригодятся *сервисы, предоставляемые пользователем* (https://docs.cloudfoundry.org/devguide/services/user-provided.html). Подобный сервис можно представить себе в виде фразы «Возьмите эту информацию о подключении, и присвойте ей логическое имя, и сделайте что-нибудь, что может рассматриваться как и любой другой управляемый опорный сервис» — JSON-запись в `VCAP_SERVICES`.

Опорный сервис, созданный с помощью команды `cf create-service` или в виде сервиса, предоставленного пользователем, невидим никаким приложениям-потребителям до тех пор, пока не будет связан с приложением; такая связь образуется добавлением соответствующей информации о подключении к переменной `VCAP_SERVICES` этого приложения.

Если в Cloud Foundry поддерживается нужный опорный сервис (такой как MySQL или MongoDB) и ваш код был написан таким образом, что в нем централизованно выполняется инициализация или получение этих опорных сервисов — в идеале с использованием чего-то вроде внедрения зависимости (с чем Spring справляется весьма легко!), — то переключение сводится к простому переписыванию этой изолированной зависимости. Если ваше приложение было написано с прицелом на поддержку конфигурации стиля 12 факторов, где такие вещи, как полномочия, хосты и порты, обслуживаются в среде или в крайнем случае носят по отношению к сборке приложения внешний характер, то вы сможете просто указать вашему приложению на его новые сервисы даже без его повторной сборки. Чтобы поглубже заглянуть в эту тему, вернитесь к рассмотрению конфигурации сервисов для 12-факторных приложений, предложенному в главе 3.

Зачастую все не так просто. Классические приложения J2EE/Java EE часто решают вопросы с сервисами за счет их поиска в широко известном контексте: JNDI (Java Naming and Directory Interface). Если код был написан с прицелом на использование внедрения зависимости, то будет относительно легко переписать приложение с целью разрешить вопрос с информацией о его подключении из среды Cloud Foundry. Если же нет, то потребуется переработка кода, причем в идеале путем ввода механизма внедрения зависимостей, чтобы оградить приложение от дальнейшей продублированности кода. Наличие возможности указать на одно (и только одно) место в кодовой основе, которое взаимодействует с JNDI или с Cloud Foundry, говорит о том, что все вышло удачно. Четко организованная кодовая основа будет иметь дело с типами, такими как `javax.sql.DataSource` и `javax.jms.ConnectionFactory`, а не с поисками JNDI.

Достижение паритета сервисов с помощью Spring

В этом разделе мы рассмотрим сложности, преодолеваемые при перемещении приложений в облегченные контейнеры и в более широком смысле в облако. И это далеко не все.

Вызовы удаленных процедур

Платформа Cloud Foundry (да и преобладающее большинство облачных сред) базируется в первую очередь на использовании протокола HTTP. Она поддерживает отдельно адресуемые узлы и даже имеет поддержку для немаршрутизируемых пользовательских портов, но эти особенности не приветствуются и поддерживаются не в каждой среде. Если, к примеру, вызовы удаленных процедур (RPC) выполняются с помощью RMI/EJB, то их приходится проводить через HTTP. Игнорируя пока здравый смысл в вопросах применения RPC, будет проще совершать RPC через HTTP. Для этого есть множество способов, включая XML-RPC, SOAP (!) и даже имеющиеся в среде Spring экспортеры и клиенты сервисов, использующие технологию HTTP Invoker (https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#remoting), которые переводят полезную нагрузку RMI через HTTP. Удобство последнего варианта обуславливается его поддержкой развитых API и Java-типов через Java-сериализацию, но рабочая нагрузка проводится по HTTP. Это неудобно, поскольку клиент и производитель теперь связаны друг с другом и должны согласовывать типы; но мы отвлеклись. В примере 5.6 продемонстрирован способ экспорта bean-компонента Spring, Simple MessageService, через интерфейс с применением Spring-сервиса HTTP Invoker.

Пример 5.6. Экспорт сервиса с помощью HttpInvokerServiceExporter

```
package demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter;

@SpringBootApplication
public class DemoApplication {

 public static void main(String[] args) throws Exception {
  SpringApplication.run(DemoApplication.class, args);
 }

 @Bean
 MessageService messageService() {  ❶
  return new SimpleMessageService();
 }

 ❷
 @Bean(name = "/messageService")
 HttpInvokerServiceExporter httpMessageService() {
  HttpInvokerServiceExporter http = new HttpInvokerServiceExporter();
  http.setServiceInterface(MessageService.class);
  http.setService(this.messageService());
  return http;
 }
}
```

1. Сама реализация нуждается в добавлении как минимум интерфейса сервиса, указанного в `HttpInvokerServiceExporter`.
2. `HttpInvokerServiceExporter` отображает переданный bean-компонент на конечную точку HTTP (`/messageService`) под Spring `DispatcherServlet`.

Для поддержки сериализации в `Message` нужна реализация `java.io.Serializable`, как и при использовании естественной RMI-сериализации. Spring предоставляет компоненты зеркального отображения для создания клиентов для этих удаленных сервисов на основе согласованного интерфейса сервиса. В примере 5.7 для создания посредника удаленного сервиса на стороне клиента, связанного с тем же сервисным контрактом, будет применяться `HttpInvokerProxyFactoryBean`. Кстати, этот совместно используемый контракт является качеством RPC, на которое наложено следующее ограничение: оно привязывает клиента к типам сервиса и мешает развитию сервиса без вывода из строя клиента.

Существует ряд новых RPC-сред, таких как gRPC, в которых поддерживаются дополнительные изменения рабочих нагрузок. В этих технологиях клиенты и сервисы могут взаимодействовать до тех пор, пока согласован общий поднабор.

Создать клиента для связи с этой конечной точкой с помощью `HttpInvokerProxyFactoryBean` довольно просто. Зеркальное отображение `HttpInvokerServiceExporter` выглядит следующим образом.

Пример 5.7. В нашем тесте в DemoApplicationTests.java сначала устанавливается, а затем и вызывается RPC-клиент

```
package demo;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean;

@Configuration
public class DemoApplicationClientConfiguration {

 @Bean
 HttpInvokerProxyFactoryBean client() {
  HttpInvokerProxyFactoryBean client = new HttpInvokerProxyFactoryBean();
  client.setServiceUrl("http://localhost:8080/messageService");  (1)
  client.setServiceInterface(MessageService.class);  (2)
  return client;
 }
}
```

1. Указанный здесь URL является строковым именем, определенным для bean-компонента сервиса.
2. Код на стороне клиента должен внедрить тип `messageService`, чтобы получить возможность совершать вызовы к нисходящему сервису.

HTTP-сессии со Spring Session

Платформа Cloud Foundry (и в основном все облачные среды) не очень хорошо работает с многоадресной сетью. Репликация HTTP-сессии обычно связывается с такой сетью. На традиционных серверах приложений репликация сессии является функцией широкого сетевого вещания на одном кластере. К сожалению, эта традиционная схема репликации сессии не имеет особой эффективности, надежности или переносимости.

Помощь может прийти от проекта Spring Session (https://spring.io/projects/spring-session). Данный проект — замена для Servlet HTTP Session API, полагающаяся при обработке синхронизации на последовательный периферийный интерфейс SPI. Реализации SPI могут обращаться к любым разновидностям внутреннего интерфейса, включая Redis, Apache Geode и Hazelcast. Особое внимание в этих проектах уделяется, в частности, созданию надежных кластерных и межкластерных репликаций. Кроме того, большинство облачных платформ поддерживают их с легкостью. К примеру, Redis заранее развернут на большинстве установок Cloud Foundry, где также имеется очень хороший сервис-брокер Hazelcast (https://docs.pivotal.io/partners/hazelcast/).

Технологии, подобные Redis и Hazelcast, играют важную роль, но увеличат объем операционных издержек. В идеале их полномочия должны быть переданы платформе и автоматизированы. Так как вы располагаете средствами с примерно одинаковыми функциональными возможностями, следует выбрать то из них, которое проще ввести в действие.

Вам нужно просто установить Spring Session, ничего при этом не делая со своим кодом HTTP-сессии. Спецификация HTTP-сервлета обеспечивает замену таким образом, что работа по реализациям сервлетов ведется согласованно. В свою очередь, Spring Session записывает состояние сессии с помощью SPI. Проект Redis на платформе Cloud Foundry доступен в качестве опорного сервиса. При развертывании нескольких узлов все они обращаются к одному и тому же Redis-кластеру и пользуются имеющейся в Redis репликацией состояния мирового уровня. Чтобы заставить Redis и Spring Session действовать, добавьте к вашему приложению Spring Boot `org.springframework.boot : spring-boot-starter-redis` и `org.springframework.session : spring-session`.

Servlet API требует, чтобы любые объекты, записываемые в HTTP-сессию и предназначенные для репликации, поддерживали Java-сериализацию. Это ограничение может оставаться в силе и для реализаций Spring Session! С другой стороны, его может и не быть, поскольку некоторые внутренние интерфейсы способны в своей работе не зависеть от Java-сериализации.

В качестве бесплатного приложения Spring Session предоставляет ряд других функций:

- поддержку работы с HTTP-сессией из WebSockets;
- простую поддержку функций типа «Обеспечьте мне выход из моей учетной записи» (Sign me out of my account);
- поддержку обращения к сессиям, логически отличающимся друг от друга. То есть ничто не мешает двум отдельным приложениям совместно использовать одно и то же состояние сессии.

Дополнительные сведения можно найти в статье The Portable, Cloud-Ready HTTP Session (https://spring.io/blog/2015/03/01/the-portable-cloud-ready-http-session) в блоге Spring.

Чтобы посмотреть код примера 5.8 в действии, загрузите Redis с помощью Redis CLI, а затем уберите его, применив команду `FLUSHALL`.

Пример 5.8. DemoApplication.java

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpSession;
import java.util.HashMap;
import java.util.Map;
import java.util.Optional;
import java.util.UUID;

@SpringBootApplication
public class DemoApplication {

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

@RestController
class SessionController {

 private final String ip;

 @Autowired
 public SessionController(@Value("${CF_INSTANCE_IP:127.0.0.1}") String ip) {
```

```
  this.ip = ip;
 }

 @GetMapping("/hi")
 Map<String, String> uid(HttpSession session) {
  (1)
  UUID uid = Optional.ofNullable(UUID.class.cast(session.getAttribute("uid")))
   .orElse(UUID.randomUUID());
  session.setAttribute("uid", uid);

  Map<String, String> m = new HashMap<>();
  m.put("instance_ip", this.ip);
  m.put("uuid", uid.toString());
  return m;
 }
}
```

(1) В этом примере атрибут сохраняется в HTTP-сессии в том случае, если его еще там нет. Последующие вызовы, адресованные к конечной точке `/hi`, будут возвращать одно и то же значение, кэшированное в HTTP-сессии.

Выполнение кода примера 5.9 приведет к удалению всего, что есть в вашей базе данных Redis!

Пример 5.9. Опустошение базы данных Redis

```
127.0.0.1:6379> flushall
OK

127.0.0.1:6379> keys *
(empty list or set)
```

Затем вызовите веб-приложение из http://localhost:8080/hi. Обновите его данные несколько раз, и увидите, что значение в `uuid` после первого обновления зафиксировалось. Это привело к запуску новой HTTP-сессии, которую Spring Session будет хранить в Redis. Убедитесь в вышесказанном, запустив команду `KEYS *` в отношении экземпляра Redis еще раз (пример 5.10).

Пример 5.10. База данных Redis, хранящая данные сессии

```
127.0.0.1:6379> keys *
(empty list or set)

127.0.0.1:6379> keys *
1) "spring:session:expirations:1490589000000"
2) "spring:session:sessions:expires:7c82002e-dd4d-4196-b9ae-1d93037192f4"
3) "spring:session:sessions:7c82002e-dd4d-4196-b9ae-1d93037192f4"
```

Если запустить команду `FLUSHALL` еще раз, то база данных будет перезапущена и при следующем обновлении экрана браузера появится новое значение `uuid`.

Сервис сообщений Java

Об удачном решении JMS для Cloud Foundry нам ничего не известно. Как бы это агрессивно ни звучало, вполне очевидно, что для использования протокола AMQP, о котором говорится в спецификации платформы RabbitMQ, основной объем кода JMS придется переделать. Если вы применяете Spring, то элементарные средства, работающие с JMS или с RabbitMQ (или, конечно же, с поддержкой механизма публикации-подписки, имеющегося в Redis), выглядят и действуют одинаково. На платформе Cloud Foundry доступны RabbitMQ и Redis. Альтернативный подход, который может сработать, заключается в применении клиента RabbitMQ JMS (https://github.com/rabbitmq/rabbitmq-jms-client).

Распределенные транзакции, использующие протокол X/Open XA и JTA

Если вашему приложению требуются распределенные транзакции с использованием протокола XA/Open и JTA, то можете выполнить настройку автономных XA-провайдеров с помощью Spring (https://spring.io/blog/2011/08/15/configuring-spring-and-jta-without-full-java-ee/), и проще всего это сделать с применением Spring Boot (https://spring.io/blog/2014/11/23/bootiful-java-ee-support-in-spring-boot-1-2). Вам не нужен Java EE-контейнер, принимающий диспетчера XA-транзакций. В примере 5.11 определяется слушатель сообщений JMS и сервис на основе JPA.

Пример 5.11. Использование JTA наряду с автоматическим конфигурированием JTA, имеющимся в Spring Boot

```
package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.transaction.Transactional;

@SpringBootApplication
public class DemoApplication {

 public static void main(String[] args) throws Exception {
```

```
  SpringApplication.run(DemoApplication.class, args) ;
 }
}

interface AccountRepository extends JpaRepository<Account, Long> {
}

@Entity
class Account {

 @Id
 @GeneratedValue
 private Long id;
 private String username;

 Account() {
 }

 public Account(String username) {
  this.username = username;
 }

 public String getUsername() {
  return this.username;
 }
}

@Component
class Messages {

 private Log log = LogFactory.getLog(getClass());

 @JmsListener(destination = "accounts")
 Public void onMessage(String content) {
  log.info("----> " + content);
 }
}

@Service
@Transactional
class AccountService {

 @Autowired
 private JmsTemplate jmsTemplate;

 @Autowired
 private AccountRepository accountRepository;

 public void createAccountAndNotify(String username) {
  this.jmsTemplate.convertAndSend("accounts", username);
  this.accountRepository.save(new Account(username));
```

```
  if ("error".equals(username)) {
   throw new RuntimeException("Simulated error");
  }
 }
}
```

Spring Boot автоматически включает в список ресурсов в глобальной транзакции JDBC `XADataSource` и JMS `XAConnectionFactory`. В блочном тесте это демонстрируется путем запуска отката с последующим подтверждением отсутствия побочных эффектов либо в JDBC `DataSource`, либо в JMS `ConnectionFactory` (пример 5.12).

Пример 5.12. Применение JTA

```
package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import static org.junit.Assert.assertEquals;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = DemoApplication.class)
public class DemoApplicationTests {

 private Log log = LogFactory.getLog(getClass());

 @Autowired
 private AccountService service;

 @Autowired
 private AccountRepository repository;

 @Test
 public void contextLoads() {
  service.createAccountAndNotify("josh");
  log.info("count is " + repository.count());
  try {
   service.createAccountAndNotify("error");
  }
  catch (Exception ex) {
   log.error(ex.getMessage());
  }
  log.info("count is " + repository.count());
  assertEquals(repository.count(), 1);
 }
}
```

В Spring Boot поддерживаются несколько свойств, которые можно указать при настройке места, в котором реализации основного интерфейса JTA (Bitronix, Atomikos) будут хранить учетные записи о транзакциях.

В идеале учетные записи о транзакциях должны где-то храниться вечно. У приложений, выполняемых на платформе Cloud Foundry, нет гарантированной файловой системы. Если экземпляр приложения выходит из строя, то нет никаких гарантий того, что содержимое файловой системы будет доступно другим узлам или приложению после его перезапуска (который также, возможно, произойдет на другом узле). Для получения абсолютной гарантии восстановления нужно будет воспользоваться чем-то более постоянным.

Облачные файловые системы

Cloud Foundry не предоставляет долговременную файловую систему. На этой платформе вы можете применять файловые системы на FUSE-основе, наподобие SSHFS. FUSE представляет собой API уровня C/C++ для создания реализаций файловых систем в пространстве пользователя. Существует множество файловых систем на основе FUSE, предоставляющих HTTP API, SSH-подключения, файловые системы MongoDB и другие для операционных систем UNIX-стиля. Это позволяет подключаться в пользовательском пространстве удаленной файловой системы с помощью, к примеру, SSH. Естественно, понадобится удаленная машина с имеющимся на ней SSH-доступом, но это всего лишь один из допустимых вариантов. Он особенно удобен в случае применения интерфейса JTA, которому требуется настоящий `java.io.File`, чтобы сдерживать обещания, касающиеся целостности данных. Этот вариант медленнее остальных.

Если нужно выполнять операции считывания и записи байтов и вас не волнует, с помощью чего происходят операции ввода-вывода: `java.io.File` или альтернативного, похожего на файловую систему опорного сервиса, можно обратиться к одному из многих подходящих альтернативных вариантов, такому как решение на основе MongoDB GridFS (http://www.mkyong.com/mongodb/spring-data-mongodb-save-binary-file-gridfs-example/) или S3 на основе сервисов Amazon Web. (Подключение к Amazon S3 мы изучим при рассмотрении вопроса создания сервис-брокеров в главе 14.) Эти опорные сервисы предлагают похожий на файловую систему API; байты будут считываться, записываться и запрашиваться по логическому имени. Spring Data MongoDB предоставляет очень удобный шаблон `GridFsTemplate`, позволяющий довольно быстро выполнять операции чтения и записи данных.

Рассмотрим пример, в котором загруженные файлы сохраняются с помощью внутреннего интерфейса MongoDB (пример 5.13). После запуска приложения откройте страницу на http://localhost:8080, чтобы получить возможность работать с REST API.

Такие технологии, как MongoDB, играют весьма важную роль, но увеличат объем операционных издержек. В идеале их полномочия должны быть переданы платформе и автоматизированы. При использовании платформы Cloud Foundry в каталоге сервисов имеется сервис MongoDB.

Пример 5.13. Применение MongoDB GridFS в качестве опорного сервиса для REST API, считывающего и записывающего данные

```
package demo;

import com.mongodb.gridfs.GridFSDBFile;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.mongodb.core.query.Query;
import org.springframework.data.mongodb.gridfs.GridFsCriteria;
import org.springframework.data.mongodb.gridfs.GridFsTemplate;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import java.io.ByteArrayOutputStream;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

@SpringBootApplication
public class DemoApplication {

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

@Controller
@RequestMapping(value = "/files")
class FileController {

 @Autowired
 private GridFsTemplate gridFsTemplate;

 private static Query getFilenameQuery(String name) {
  return Query.query(GridFsCriteria.whereFilename().is(name));
 }

 ❶
 @RequestMapping(method = RequestMethod.POST)
```

```
String createOrUpdate(@RequestParam MultipartFile file) throws Exception {
 String name = file.getOriginalFilename();
 maybeLoadFile(name).ifPresent(
  p -> gridFsTemplate.delete(getFilenameQuery(name)));
 gridFsTemplate.store(file.getInputStream(), name, file.getContentType())
  .save();
 return "redirect:/";
}

❷
@RequestMapping(method = RequestMethod.GET)
@ResponseBody
List<String> list() {
 return getFiles().stream().map(GridFSDBFile::getFilename)
 .collect(Collectors.toList());
}

❸
@RequestMapping(value = "/{name:.+}", method = RequestMethod.GET)
ResponseEntity<?> get(@PathVariable String name) throws Exception {
 Optional<GridFSDBFile> optionalCreated = maybeLoadFile(name);
 if (optionalCreated.isPresent()) {
  GridFSDBFile created = optionalCreated.get();
  try (ByteArrayOutputStream os = new ByteArrayOutputStream()) {
   created.writeTo(os);

   HttpHeaders headers = new HttpHeaders();
   headers.add(HttpHeaders.CONTENT_TYPE, created.getContentType());
   return new ResponseEntity<byte[]>(os.toByteArray(), headers, HttpStatus.OK);
  }
 }
 else {
  return ResponseEntity.notFound().build();
 }
}

private List<GridFSDBFile> getFiles() {
 return gridFsTemplate.find(null);
}

private Optional<GridFSDBFile> maybeLoadFile(String name) {
 GridFSDBFile file = gridFsTemplate.findOne(getFilenameQuery(name));
 return Optional.ofNullable(file);
}
}
```

❶ Конечная точка `/files` получает разбитые на несколько частей данные выкладываемого файла и записывает их в MongoDB GridFS.

❷ Конечная точка `/files` просто возвращает листинг файлов в GridFS.

❸ Конечная точка `/files/{имя}` считывает байты из GridFS и отправляет их обратно клиенту.

В качестве альтернативы, если ваше приложение использует файловую систему непостоянно, для выкладывания промежуточного файла или выполнения каких-либо иных действий можно задействовать временный каталог вашего приложения на Cloud Foundry. Однако при этом имейте в виду, что платформа не дает никаких гарантий долговечности существования подобных данных.

HTTPS

Платформа Cloud Foundry доводит HTTPS-запросы до широкодоступного прокси-сервера, защищающего все приложения. Любой вызов, который вы направляете к приложению, откликнется также и на HTTPS. При использовании платформы Cloud Foundry, находящейся на вашем оборудовании, можете предоставлять собственные сертификаты централизованно.

Email

Что задействует ваше приложение: SMTP/POP3 или IMAP? Если электронная почта используется из Java-приложения, то, скорее всего, применяется JavaMail. Компонент JavaMail — API для поддержки обмена данными по электронной почте на основе SMTP/POP3/IMAP. Существует множество поставщиков электронной почты в виде сервиса. SendGrid (https://docs.spring.io/spring-boot/docs/1.3.0.BUILD-SNAPSHOT/api/org/springframework/boot/autoconfigure/sendgrid/SendGridAutoConfiguration.html), поддерживаемый в исходном состоянии средой Spring Boot, представляет собой поставщика электронной почты, предназначенного для работы в облачной среде и имеющего простой API. Рассмотрим пример (пример 5.14).

Технологии, подобные SendGrid, являются идеальным вариантом, поскольку вопросы их размещения и управления решаются за вас. Можно даже создать привязку к их сервису, используя каталог сервисов на некоторых вариантах Cloud Foundry, подобных Pivotal Web Services.

Пример 5.14. Использование имеющейся в Spring Boot поддержки SendGrid

```
package demo;

import com.sendgrid.SendGrid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class DemoApplication {

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

@RestController
class EmailRestController {

 @Autowired
 private SendGrid sendGrid;

 ❶
 @RequestMapping("/email")
 SendGrid.Response email(@RequestParam String message) throws Exception {
  SendGrid.Email email = new SendGrid.Email();
  email.setHtml("<hi>" + message + "</h1>");
  email.setText(message);
  email.setTo(new String[] { "user1@host.io" });
  email.setToName(new String[] { "Josh" });
  email.setFrom("user2@host.io");
  email.setFromName("Josh (sender)");
  email.setSubject("I just called.. to say.. I (message truncated)");
  return sendGrid.send(email);
 }

}
```

❶ REST API берет сообщение в качестве параметра запроса и отправляет сообщение электронной почты, применяя автоматически сконфигурированный Java-клиент SendGrid. Автоматическое конфигурирование предполагает использование надлежащего имени пользователя SendGrid (`spring.sendgrid.username`) и пароля (`spring.sendgrid.password`). Следует помнить, что Spring Boot приводит свойства к общему виду и эти значения могут быть предоставлены в виде переменных среды: `SPRING_SENDGRID_USERNAME`, `SPRING_SENDGRID_PASSWORD` и т. д.

Управление идентификацией

В распределенных системах управление идентификацией — аутентификация и авторизация — играет важную роль. Без возможности централизованного описания пользователей, ролей и разрешений просто не обойтись. В платформу Cloud

Foundry включен высокоэффективный сервис аутентификации и авторизации под названием UAA, обратиться к которому можно с помощью Spring Security. Альтернативой и вполне достойным сервисом со сторонним размещением можно считать Stormpath (https://stormpath.com/), способный работать в качестве своеобразного фасада перед другими поставщиками средств идентификации или же сам выступать в роли поставщика таких средств. Он очень просто интегрируется как в Spring Boot, так и в Spring Security (https://github.com/stormpath/stormpath-sdk-java)!

Важность таких технологий, как Okta, обуславливается тем, что вопросы их размещения и управления полностью решаются за вас. С ними связан куда менее существенный объем издержек, чем от чего-то, подобного такому средству, как Active Directory, обслуживание которого может стать штатной работой администратора!

В начале 2017 года компания Okta приобрела Stormpath, обязавшись сохранить алгоритм взаимодействия для приложений, разработанных с расчетом на применение Stormpath, но с подключением их к внутреннему интерфейсу Okta.

Резюме

Надеемся, вы нашли в данной главе сведения, помогающие перенести в облако приложение, слабо поддающееся изменениям и созданное в прежние времена! Если именно это вы и пытаетесь сделать, то обратите внимание на написание тестов для автоматизации проверки того, что, по вашему мнению, станет противиться этой миграции. При ожидаемых проблемах с миграцией доступа к базе данных, доступа к файловой системе и к средствам работы с электронной почтой можете настроить конечную точку, подключаемую к базе данных, и конечную точку, отправляющую сообщения электронной почты, а также конечную точку, считывающую данные из файлов предпочтительным для вас способом. Внося изменения в приложение с целью его миграции, воспользуйтесь для развертывания технологией непрерывной интеграции и протестируйте эти конечные точки. Как только все конечные точки начнут функционировать и вдобавок вы убедитесь в работоспособности всего остального, перейдите к более обстоятельному запуску и получите сертификат приложения. Если вы волнуетесь за то, что эти конечные точки остаются в приложении, то спокойно удалите их, но не забудьте добавить подробнейшее описание необходимых дальнейших действий (TODO) или невыполненной работы, чтобы впоследствии добавить актуатор Spring Boot. Он предоставляет легко защищаемую конечную точку `/health`, дающую точно

такую же ответную реакцию. При наличии резерва времени приоритетом должно стать написание действенных тестов!

Целью этой главы было рассмотрение общих вопросов, характерных для действий по переносу существующих ранее разработанных приложений в облако. Обычно при такой миграции задействуются некоторые комбинации описанных действий. Как только миграция будет успешно завершена, выпейте полный стакан... воды или того, что пожелаете! Вы заслужили это. Количество проблем, требующих управления и пристального внимания, теперь уменьшилось на одну. И ваши заботы будут переложены на платформу.

Часть II
Веб-сервисы

6 REST API

Компания Amazon переживала трудные времена, когда начала расширять масштабы разработки разных функций, возлагая эти действия на различные команды, поскольку стала зависимой от совместного доступа команд к нескольким базам данных. Это отрицательно повлияло на возможность отдельно взятой команды изменять функциональные свойства, не оказывая влияния ни на кого другого в рамках компании. Поэтому, как в 2006 году объяснил технический директор Amazon.com Вернер Фогельс (Werner Vogels) (https://queue.acm.org/detail.cfm?id=1142065), вся работа по интеграции должна выполняться в понятиях API, а не в вызовах, обращаемых к базам данных.

Это стало первым весьма важным шагом на пути к использованию микросервисов: все должно идти через API. Передача репрезентативного состояния (Representational State Transfer (REST)) — несомненно, самый популярный протокол, поддерживаемый миллионами API, применяемых в веб-программировании.

Первоначально протокол REST был предложен доктором Роем Филдингом (Roy Fielding) в качестве составляющей его докторской диссертации (https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm) в 2000 году. Филдинг поспособствовал определению спецификации HTTP и хотел помочь проиллюстрировать, как веб-технология, прошедшая проверку, способная к широкому масштабированию, децентрализованная, отказоустойчивая структура, может использоваться для создания сервисов. Существующим обоснованием *права* архитектуры REST *на существование* является протокол HTTP.

В отличие от того, что предыдущие подходы к распределенным сервисам (такие как CORBA, EJB, RMI и SOAP) в большей или меньшей степени фокусировались на предоставлении объектно-ориентированного интерфейса и методов в виде удаленных доступных сервисов (RPC), REST сосредотачивается на манипулировании удаленными *ресурсами* или объектами. На наименованиях, а не на операциях. На объектах, а не на действиях.

Модель зрелости Леонарда Ричардсона

Сутью REST является раскрытие изменений бизнес-состояния с помощью операций (`GET`, `PUT`, `POST`, `DELETE` и т. д.) и идиом (HTTP-заголовков, кодов состояний и др.), которые HTTP уже предоставляет для поддержки обмена данными между сервисами. Лучшие REST API стремятся применять максимальное количество возможностей, предоставляемых HTTP. REST при всех своих преимуществах — архитектурное ограничение HTTP, а не какой-то *стандарт*, и поэтому мы увидели распространение API различной степени соответствия принципам RESTful. Леонард Ричардсон (Leonard Richardson) выдвинул свою модель зрелости REST (https://martinfowler.com/articles/richardsonMaturityModel.html), чтобы помочь оценить соответствие API принципам REST.

- *Уровень 0: трясина POX.* (Он *действительно* должен быть с нулевым индексом!) Дает описание API, применяющих HTTP исключительно в качестве транспортного протокола. К примеру, веб-сервисы на основе протокола SOAP задействуют HTTP, но *могут* и просто использовать JMS. Они в данном случае работают на основе HTTP. Такой сервис применяет HTTP главным образом в качестве туннеля через один URI. Образцами могут послужить SOAP и XML-RPC. Они обычно используют только одну операцию HTTP (`POST`).
- *Уровень 1: ресурсы.* Дает описание API, использующих несколько URI, чтобы различать связанные наименования, а в остальном сторонится применения всех функций HTTP.
- *Уровень 2: HTTP-операции.* Дает с целью совершенствования сервиса описание API, использующих транспортные свойства (такие как операции HTTP и коды состояния). Если со Spring MVC, или JAX-RS, или любой другой современной REST-средой *все* делается неправильно, то вы *все равно*, скорее всего, получите API, совместимый с уровнем 2! Это отличная отправная точка.
- *Уровень 3: средства управления гипермедиа — Hypermedia controls (HATEOAS, что означает Hypermedia as the Engine of Application State, то есть гипермедиа в качестве обработчика состояния приложения).* Дает описание API, которым для перемещения по сервису не требуется *предварительное* знание этого сервиса. Такой сервис способствует долговечности использования благодаря единому интерфейсу для опроса структуры сервиса.

В этой главе мы рассмотрим способы применения Spring для создания REST-сервисов, начиная с уровня 2. Чуть позже изучим приемы использования гипермедиа с целью помочь созданию однообразных, самостоятельно описываемых сервисов. REST API предназначены для представления HTTP-контрактов между сервисами, имеющимися в системе. Они будут применяться API-клиентами, но люди должны вести в отношении их специальные разработки, поэтому особое внимание мы также уделим разработкам доступных, правильно и последовательно документированных API.

В этой главе мы собираемся использовать Spring MVC. Кроме того, Spring хорошо взаимодействует с JAX-RS. Если вы задействуете Spring Initializr (http://start.spring.io/), то выберите Jersey (JAX-RS) для веб-приложения, поддерживающего JAX-RS. Не составит большого труда получить и другие реализации, работающие со Spring. Дополнительные сведения об интеграции других сред на основе сервлетов можно найти в материале по применению традиционных API Java EE (подобных сервлетам) из приложений Spring Boot в разделе «Использование API Servlet в приложениях Spring Boot» на с. 599.

REST — важная составляющая приложений, ориентированных на выполнение в облаке. Хотя ничто, касающееся микросервисов, не предусматривает или даже не требует REST, эта технология наиболее распространена в вопросе предоставления сервисов.

Технология HTTP вполне подходит для приложений, ориентированных на выполнение в облачной среде. Она идеальна для кэширования сервисов, поскольку у нее отсутствует состояние на стороне клиента. А каждый запрос является автономным. Это также означает, что сервисы могут легко поддаваться горизонтальному масштабированию, до тех пор пока состояние, представленное REST API, тоже способно масштабироваться горизонтально. Кэширование сокращает время до получения первого байта. За счет кэширования и GZip-сжатия из отдельных HTTP-запросов можно выжать немало дополнительной производительности. С помощью уже заметного на горизонте HTTP 2 эта ситуация в ближайшем будущем, похоже, значительно улучшится.

Технология HTTP безразлична к типу содержимого и включает поддержку согласования контента; один клиент может поддерживать только XML, а другой — JSON или Protocol Buffers компании Google. Все перечисленное могут обслуживать одни и те же сервисы. Кроме того, это расширяемый механизм.

Простые REST API, создаваемые с помощью Spring MVC

Чтобы привнести сервлет-контейнер и полную конфигурацию для Spring MVC, добавьте к вашей сборке `org.springframework.boot : spring-boot-starter-web`. Spring MVC — HTTP-среда, ориентированная на запрос-ответ. В Spring MVC на HTTP-запросы и последующее предоставление ответов отображаются методы обработчиков, имеющиеся в *контроллере*. Рассмотрим REST API, предназначенный для взаимодействия с объектами `Customer` (пример 6.1).

Пример 6.1. Наш первый @RestController, CustomerRestController

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.mvc.method.annotation.
MvcUriComponentsBuilder;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import java.net.URI;
import java.util.Collection;

@RestController
@RequestMapping("/v1/customers")
public class CustomerRestController {

 @Autowired
 private CustomerRepository customerRepository;

 ❶
 @RequestMapping(method = RequestMethod.OPTIONS)
 ResponseEntity<?> options() {

  //@formatter:off
  return ResponseEntity
   .ok()
   .allow(HttpMethod.GET, HttpMethod.POST,
          HttpMethod.HEAD, HttpMethod.OPTIONS,
          HttpMethod.PUT, HttpMethod.DELETE)
          .build();
   //@formatter:on
 }

 @GetMapping
 ResponseEntity<Collection<Customer>> getCollection() {
  return ResponseEntity.ok(this.customerRepository.findAll());
 }

 ❷
 @GetMapping(value = "/{id}")
 ResponseEntity<Customer> get(@PathVariable Long id) {
  return this.customerRepository.findById(id).map(ResponseEntity::ok)
   .orElseThrow(() -> new CustomerNotFoundException(id));
 }

 @PostMapping
 ResponseEntity<Customer> post(@RequestBody Customer c) {  ❸

  Customer customer = this.customerRepository.save(new Customer(c
   .getFirstName(), c.getLastName()));

  URI uri = MvcUriComponentsBuilder.fromController(getClass()).path("/{id}")
   .buildAndExpand(customer.getId()).toUri();
```

```
  return ResponseEntity.created(uri).body(customer);
 }

 ❹
 @DeleteMapping(value = "/{id}")
 ResponseEntity<?> delete(@PathVariable Long id) {
  return this.customerRepository.findById(id).map(c -> {
   customerRepository.delete(c);
   return ResponseEntity.noContent().build();
  }).orElseThrow(() -> new CustomerNotFoundException(id));
 }

 ❺
 @RequestMapping(value = "/{id}", method = RequestMethod.HEAD)
  ResponseEntity<?> head(@PathVariable Long id) {
   return this.customerRepository.findById(id)
    .map(exists -> ResponseEntity.noContent().build())
    .orElseThrow(() -> new CustomerNotFoundException(id));
 }

 ❻
 @PutMapping(value = "/{id}")
 ResponseEntity<Customer> put(@PathVariable Long id, @RequestBody Customer c)
{
  return this.customerRepository
   .findById(id)
   .map(
    existing -> {
     Customer customer = this.customerRepository.save(new Customer(existing
      .getId(), c.getFirstName(), c.getLastName()));
     URI selfLink = URI.create(ServletUriComponentsBuilder.fromCurrentRequest()
       .toUriString());
     return ResponseEntity.created(selfLink).body(customer);
    }).orElseThrow(() -> new CustomerNotFoundException(id));

 }
}
```

❶ Метод обработчика обозначен аннотацией `@RequestMapping`. Класс должен содержать аннотацию `@RequestMapping`, тогда объявления на уровне метода будут переопределены или добавлены к корневым отображениям на уровне классов. Обработчик реагирует на HTTP-операцию `OPTIONS`, которую клиент станет выдавать, если ему требуется знать, какие еще HTTP-операции поддерживает данный ресурс.

❷ Метод обработчика возвращает отдельные записи, чьи идентификаторы закодированы в виде *переменных пути* (path variables) в `@RequestMapping` URI-синтаксисе: `{id}`.

❸ Данные могут быть переданы от клиента к сервису, и тело данных передается сервису в виде принадлежащего обработчику `@RequestBody`.

4. Обработчик `DELETE` возвращает в случае удачного завершения действий HTTP 204 или же в случае сбоя выдает исключение, выражающееся в конечном итоге в ошибке 404.
5. Обработчик `HEAD` предназначен лишь для подтверждения существования ресурса, поэтому в случае успешного завершения он возвращает управление со статусом 204 или же в случае сбоя выдает исключение, выражающееся в конечном итоге в ошибке 404, точно так же, как и обработчик `DELETE`.
6. Обработчик `PUT` обязан обновить существующую запись. Чтобы указать обновляемую запись, он использует параметры `@PathVariable`. Если запись не существует, то выдается исключение, выражающееся в конечном итоге в ошибке 404.

Это основной контроллер Spring MVC REST. Аннотация `@RequestMapping` приводит к отображению методов обработчика на спецификации типов HTTP-запросов. Данная аннотация позволяет далее различать в запросе данные по заголовкам, по отправленному и возвращенному типу содержимого, по cookie-файлам и по многому другому. Указанный путь относителен по отношению к корневому пути контекста приложения, за исключением того варианта, когда путь указан аннотацией на уровне контроллера `@RequestMapping`, в таком случае путь на уровне контроллера задается относительно пути в контроллере.

В Spring MVC также поддерживаются аннотации, специфичные для методов HTTP, такие как @GetMapping, @PostMapping и т. д. Их действие похоже на действие аннотации @RequestMapping, хотя в них подразумевается метод HTTP. Если вашей конечной точке не нужно поддерживать более одного метода HTTP, то их можно использовать вместо этой аннотации.

Согласование содержимого

Методы обработчика возвращают объекты или иногда конверт `ResponseEntity` с объектами в качестве информационного наполнения. Когда среда Spring MVC видит возвращаемое значение, она применяет стратегический объект, `HttpMessageConverter`, для преобразования объекта в представление, удобное для клиента. Последний указывает, какого типа представление он ожидает, используя для этого *согласование содержимого* (content negotiation). Изначально Spring MVC ищет в заголовке `Accept` поступающего запроса медиатип, например `application/json`, `application/xml` и т. д., который в состоянии создать сконфигурированный компонент `HttpMessageConverter` на основе полученного объекта и желаемого медиатипа. Этот же процесс происходит в обратном порядке для данных, отправленных клиентом сервису и переданных методам обработчика в качестве параметров `@RequestBody`.

Согласование содержимого входит в перечень наиболее эффективных функций HTTP: один и тот же сервис может обслуживать клиентов, у которых приняты

различные протоколы. Медиатипы следует использовать для того, чтобы обозначить клиенту тип обслуживаемого содержимого. Кроме того, обычно медиатип обозначает, как именно клиент должен обрабатывать содержимое ответа. Клиент, к примеру, знает, что ему не нужно предпринимать тщетные попытки извлечения из ответа строк в формате JSON, если указан медиатип `image/jpeg`.

Чтение и запись двоичных данных

До сих пор мы уделяли внимание данным в формате JSON, но ничто не мешает REST-ресурсам обслуживать медиаданные наподобие изображений или самых обыкновенных файлов. Посмотрим, как для конечной точки профиля считываются и записываются данные изображения, `/customers/{id}/photo` (пример 6.2).

Пример 6.2. Считывание и запись двоичных данных (изображения)

```
package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.util.Assert;
import org.springframework.util.FileCopyUtils;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import java.io.*;
import java.net.URI;
import java.util.concurrent.Callable;

//@formatter:off
import static org.springframework.web.servlet
        .support.ServletUriComponentsBuilder.fromCurrentRequest;
//@formatter:on

@RestController
@RequestMapping(value = "/customers/{id}/photo")
public class CustomerProfilePhotoRestController {

 private File root;
 private final CustomerRepository customerRepository;
 private final Log log = LogFactory.getLog(getClass());

 @Autowired
```

```
CustomerProfilePhotoRestController(
        @Value("${upload.dir:${user.home}/images}") String uploadDir,
        CustomerRepository customerRepository) {
 this.root = new File(uploadDir);
 this.customerRepository = customerRepository ;
 Assert.isTrue(this.root.exists() || this.root.mkdirs(),
  String.format("The path '%s' must exist.", this.root.getAbsolutePath()));
}

❶
@GetMapping
ResponseEntity<Resource> read(@PathVariable Long id) {
 return this.customerRepository
   .findById(id)
   .map(
    customer -> {
     File file = fileFor(customer);

     Assert.isTrue(file.exists(),
      String.format("file-not-found %s", file.getAbsolutePath()));

     Resource fileSystemResource = new FileSystemResource(file);
     return ResponseEntity.ok().contentType(MediaType.IMAGE_JPEG)
      .body(fileSystemResource);
    }).orElseThrow(() -> new CustomerNotFoundException(id));
}

❷
@RequestMapping(method = { RequestMethod.POST, RequestMethod.PUT })
Callable<ResponseEntity<?>> write(@PathVariable Long id,
 @RequestParam MultipartFile file) ❸
 throws Exception {
 log.info(String.format("upload-start /customers/%s/photo (%s bytes)", id,
  file.getSize()));
 return () -> this.customerRepository
  .findById(id)
  .map(
   customer -> {
    File fileForCustomer = fileFor(customer);
    try (InputStream in = file.getInputStream();
     OutputStream out = new FileOutputStream(fileForCustomer)) {
     FileCopyUtils.copy(in, out);
    }
    catch (IOException ex) {
     throw new RuntimeException(ex);
    }
    URI location = fromCurrentRequest().buildAndExpand(id).toUri(); ❹
    log.info(String.format("upload-finish /customers/%s/photo (%s)", id,
     location));
    return ResponseEntity.created(location).build();
```

```
  }).orElseThrow(() -> new CustomerNotFoundException(id));
 }

 private File fileFor(Customer person) {
  return new File(this.root, Long.toString(person.getId()));
 }
}
```

1. Spring MVC автоматически возвращает данные `byte[]` — файл, URL-ресурс, `InputStream` и т. д. — из буфера, лежащего в основе `org.springframework.core.io.Resource`, возвращенного из метода обработчика контроллера Spring MVC.
2. Выкладка файла может блокировать и монополизировать пул потоков Servlet-контейнера. Имеющийся в Spring MVC *предварительный* метод обработки `Callable<T>` возвращает значения настроенному пулу потоков `Executor` и высвобождает поток контейнера до готовности ответа.
3. Spring MVC автоматически отобразит выкладываемые данные файла, состоящего из нескольких частей, на параметр запроса `org.springframework.web.multipart.MultipartFile`.
4. Хорошо, когда при создании ресурса возвращается код статуса HTTP 201 (`Created`) и заголовок `Location` с URI только что созданного ресурса.

Считывание медиаданных проходит без затруднений, если данные могут быть представлены с помощью одной из Spring-реализаций `org.springframework.core.io.Resource`. Существует множество реализаций, представленных нестандартно: `FileSystemResource`, `GridFsResource` (на основе Grid File System из MongoDB), `ClassPathResource`, `GzipResource`, `VfsResource`, `UrlResource` и т. д. Spring MVC автоматически возвращает клиенту данные `byte[]` из буфера, лежащего в основе `org.springframework.core.io.Resource`.

Приложения Spring MVC запускаются в сервлет-контейнере. Сам сервлет-контейнер поддерживает пул потоков внешнего интерфейса, используемый для ответа на поступающие HTTP-запросы; при каждом поступлении такого запроса поток выполняет диспетчеризацию, чтобы принять запрос и дать ответ. Важно не допустить монополизации потоков в пуле потоков сервлет-контейнера. Spring MVC станет перемещать поток выполнения методов обработчиков контроллера в отдельный пул потоков (тот, что будет предоставлен с помощью указания bean-компонента `TaskExecutor`), если метод обработчика возвращает `java.util.concurrent.Callable<T>`, `org.springframework.web.context.request.async.DeferredResult` или `org.springframework.web.context.request.async.WebAsyncTask`.

В Spring MVC также поддерживаются веб-сокеты и события, отправляемые сервером — Server-Sent Events (SSE). И то и другое — механизмы асинхронного обмена данными, но сами по себе они разные и не являются тем, чего мы стараемся здесь достичь.

`Callable<T>` — просто специализация интерфейса `Runnable`, возвращающего результат. Spring MVC вызывает `Callable` в отношении указанного пула потоков и, как только вызываемый обработчик выдает ответ, возвращает результат в асинхронном режиме в качестве ответа на исходный HTTP-запрос.

`WebAsyncTask` — практически то же самое; `Callable<T>` заключается в этот объект, но в нем также предоставляются поля для указания исполнителя задачи `TaskExecutor`, на котором нужно запустить `Callbale<T>`, и времени ожидания, по истечении которого будет выдан ответ.

Объект `DeferredResult` не запускает выполнение в другом месте; вместо этого Spring MVC возвращает пулу поток сервлет-контейнера и запускает асинхронный ответ лишь при вызове метода `DeferredResult#setResult`. Экземпляры `DeferredResult` могут кэшироваться и впоследствии обновляться в потоке выполнения, не входящем в пул. Приведем такой пример: конечная точка может спрятать ссылки на `DeferredResult` в совместно используемом `ConcurrentHashMap<K,V>`, отображаемом путем значимого взаимосвязанного идентификатора, который чуть позже могут найти методы, имеющие аннотацию `@EventListener` или `@RabbitListener`. После получения таких асинхронных потоков в них может быть вызыван `DeferredResult#setResult` и подан сигнал о том, что ответ в конечном итоге следует возвратить клиенту.

Наш пример обрабатывает запись в отдельном потоке; предполагается, что записи могут становиться более медленными операциями, чем операции чтения, которые можно оптимизировать за счет кэширования. В примере выкладывание файла, то есть запись, перемещается в отдельный поток путем ее обработки в `Callable<T>`. С точки зрения *сервиса* определенный положительный эффект достигается, но с точки зрения *клиента* никакой пользы от этого не прослеживается. Клиент будет заблокирован в ожидании ответа, пока станет выполняться `Callable<T>`.

Альтернативный подход заключается в *немедленном* возврате ответа с кодом статуса HTTP 202 (`Accepted`) и предоставлении заголовка `Location`, который показывает место либо размещения только что созданного ресурса, либо по крайней мере нахождения информации о статусе его потенциально долгоживущего творения.

Google Protocol Buffers

Согласование содержимого позволяет (по возможности) провести оптимизацию и, кроме этого, предоставляет поддержку отказоустойчивости. Типичным примером может стать формат Google Protocol Buffers, являющийся идеальным выбором для весьма эффективного, межплатформенного обмена данными. Если клиент, отправляющий запрос, не поддерживает Google Protocol Buffers, то Spring способен благодаря согласованию содержимого понизить планку до JSON или XML. Клиенты пособразительнее получают более изощренные ответы, но вписаться в роль может кто угодно.

Google Protocol Buffers представляет собой высокоэффективный формат сериализации. Его сообщения не содержат никакой информации с описанием их *структуры*, только данные. *Структура* берется из `.proto`-определения схемы формата. Ее можно применять для создания языковой привязки с целью взаимодействия; в Google Protocol Buffers используется надежная поддержка множества языков и платформ, которые вы, вероятнее всего, и станете задействовать. Сервисам не нужно изучать поступившее информационное наполнение каждого запроса, чтобы разобраться в их структуре и выполнить ее правильное отображение, это делается только один раз при предоставлении схемы (пример 6.3).

Пример 6.3. Схема Google Protocol Buffers для сообщения Customer

```
package demo;

option java_package = "demo";  ❶

option java_outer_classname = "CustomerProtos";  ❷

message Customer {  ❸
    optional int64 id = 1;
    required string firstName = 2;
    required string lastName = 3;
}

message Customers {  ❹
    repeated Customer customer = 1;
}
```

❶ Пакет или модуль для получающихся в результате типов, ориентированный на конкретный язык.

❷ Если указать `java_outer_classname`, то типы Google Protocol Buffer могут быть вложенными внутренними классами; получающимся в итоге типом будет `demo.CustomerProtos.Customer`.

❸ Схема для одного объекта `Customer`.

❹ Схема для *коллекции* объектов `Customer`.

Полям в определении присваивается целочисленное смещение, чтобы компилятор мог понять, какие двоичные данные ему искать в потоке. Если клиенту известно прежнее определение сообщения, а сервис отвечает более подробным сообщением, то клиент все равно сможет взаимодействовать, пока смещения стабильны. Этот вариант идеален для распределенных систем, где клиенты и сервисы могут не всегда владеть одинаковой *версией* сообщения. Возможность независимого развертывания, то есть возможность развертывания одного сервиса без *развертывания* вместе с ним других, — весьма важная особенность микросервисов. Это позволяет командам повторно выпускать и развивать свои сервисы без постоянных издержек на синхронизацию. Описанная слабая связанность поддерживается и в Google Protocol Buffers.

Для чтения и записи сообщений, описание которых находится в схеме `.proto`, компилятор `protoc` создает привязки, ориентированные на разные языки (Java, Python, C, .NET, PHP, Ruby) и, кроме того, на использование большого количества других технологий. Для нашего определения `Customer` в формате Google Protocol Buffer напишем сценарий создания привязок, ориентированных на Java, Python и Ruby. С помощью этих привязок можно обратиться к нашим конечным точкам REST, усиленным применением Google Protocol Buffers (пример 6.4).

Пример 6.4. Сценарий создания требуемых клиентов Java, Ruby и Python

```
#!/usr/bin/env bash

SRC_DIR=`pwd`
DST_DIR=`pwd`/../src/main/

echo source:           ${SRC_DIR}
echo destination root: ${DST_DIR}

function ensure_implementations(){
    gem list | grep ruby-protocol-buffers || sudo gem install
        ruby-protocol-buffers
    go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
}

function gen(){
    D=$1
    echo $D
    OUT=$DST_DIR/$D
    mkdir -p $OUT
    protoc -I=$SRC_DIR --${D}_out=$OUT $SRC_DIR/customer.proto
}

ensure_implementations

gen java
gen python
gen ruby
```

Получившиеся клиенты Java, Ruby и Python не отличаются особой сложностью, но они все же работают. В примере 6.5 показана в действии привязка к Python.

Пример 6.5. Рассчитанный на язык Python клиент Google Protocol Buffer; customer_pb2 является клиентом, созданным с помощью protoc

```
#!/usr/bin/env python

import urllib

import customer_pb2

if __name__ == '__main__':
```

```
customer = customer_pb2.Customer()
customers_read = urllib.urlopen('http://localhost:8080/customers/1').read()
customer.ParseFromString(customers_read)
print customer
```

В примере 6.6 показана в действии привязка к Ruby.

Пример 6.6. Рассчитанный на язык Ruby клиент Google Protocol Buffer; /customers.pb является клиентом, созданным с помощью protoc

```
#!/usr/bin/ruby

require './customer.pb'
require 'net/http'
require 'uri'

uri = URI.parse('http://localhost:8080/customers/3')
body = Net::HTTP.get(uri)
puts Demo::Customer.parse(body)
```

В Spring MVC включена заказная реализация `HttpMessageConverter`, способная считывать и записывать объекты, созданные с помощью сгенерированных привязок к Java для нашего определения `.proto`. Компилятор `protoc` выдаст привязки для нашего компилятора Google Protobuf, который может послужить при обработке REST-запросов. Посмотрим REST API, созданный для обработки `application/x-protobuf`. Основная часть данного сервиса должна быть вам знакома. Конечно, главное различие между этим и тем, что попадалось ранее, кроме пути URI, — дополнительный неинтересный код, навязанный необходимостью преобразования из `Customer` в подходящий для Protobuffer объект переноса данных (DTO) `CustomerProtos.Customer` и обратно (пример 6.7).

Пример 6.7. REST-сервис, использующий согласование содержимого для обслуживания контента с медиатипом application/x-protobuf

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

//@formatter:off
import org.springframework.web.servlet.mvc.method
        .annotation.MvcUriComponentsBuilder;
import static org.springframework.web.servlet.support
        .ServletUriComponentsBuilder.fromCurrentRequest;
//@formatter:on

import java.net.URI;
import java.util.Collection;
```

```
import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping(value = "/v1/protos/customers")
public class CustomerProtobufRestController {

 private final CustomerRepository customerRepository;

 @Autowired
 public CustomerProtobufRestController(CustomerRepository customerRepository) {
  this.customerRepository = customerRepository;
 }

 @GetMapping(value = "/{id}")
 ResponseEntity<CustomerProtos.Customer> get(@PathVariable Long id) {
  return this.customerRepository.findById(id).map(this::fromEntityToProtobuf)
   .map(ResponseEntity::ok)
   .orElseThrow(() -> new CustomerNotFoundException(id));
 }

 @GetMapping
 ResponseEntity<CustomerProtos.Customers> getCollection() {
  List<Customer> all = this.customerRepository.findAll();
  CustomerProtos.Customers customers = this.fromCollectionToProtobuf(all);
  return ResponseEntity.ok(customers);
 }

 @PostMapping
 ResponseEntity<CustomerProtos.Customer> post(
  @RequestBody CustomerProtos.Customer c) {

  Customer customer = this.customerRepository.save(new Customer(c
   .getFirstName(), c.getLastName()));

  URI uri = MvcUriComponentsBuilder.fromController(getClass()).path("/{id}")
   .buildAndExpand(customer.getId()).toUri();
  return ResponseEntity.created(uri).body(this.fromEntityToProtobuf(customer));
 }

 @PutMapping("/{id}")
 ResponseEntity<CustomerProtos.Customer> put(@PathVariable Long id,
  @RequestBody CustomerProtos.Customer c) {

  return this.customerRepository
   .findById(id)
   .map(
    existing -> {

     Customer customer = this.customerRepository.save(new Customer(existing
```

```
      .getId(), c.getFirstName(), c.getLastName()));

     URI selfLink = URI.create(fromCurrentRequest().toUriString());

     return ResponseEntity.created(selfLink).body(
      fromEntityToProtobuf(customer));
    }).orElseThrow(() -> new CustomerNotFoundException(id));
 }

 private CustomerProtos.Customers fromCollectionToProtobuf(
  Collection<Customer> c) {
  return CustomerProtos.Customers
   .newBuilder()
   .addAllCustomer(
    c.stream().map(this::fromEntityToProtobuf).collect(Collectors.toList()))
  .build();
 }

 private CustomerProtos.Customer fromEntityToProtobuf(Customer c) {
  return fromEntityToProtobuf(c.getId(), c.getFirstName(), c.getLastName());
 }

 private CustomerProtos.Customer fromEntityToProtobuf(Long id, String f,
  String l) {
  CustomerProtos.Customer.Builder builder = CustomerProtos.Customer
   .newBuilder();
  if (id != null && id > 0) {
   builder.setId(id);
  }
  return builder.setFirstName(f).setLastName(l).build();
 }
}
```

Чтобы дать возможность Spring MVC распознать новый тип содержимого, заказную реализацию `HttpMessageConverter` следует зарегистрировать, как показано в примере 6.8.

Пример 6.8. Регистрация заказной реализации HttpMessageConverter с целью преобразовать HTTP-сообщения в медиатип application/x-protobuf и обратно

```
package demo;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.converter.protobuf.
ProtobufHttpMessageConverter;

@Configuration
class GoogleProtocolBuffersConfiguration {

 @Bean
```

```
 ProtobufHttpMessageConverter protobufHttpMessageConverter() {
  return new ProtobufHttpMessageConverter();
 }
}
```

Клиент `RestTemplate` поддерживает тот же механизм согласования содержимого, что и сервис Spring MVC; для *чтения* данных `application/x-protobuf`, сконфигурируйте `ProtobufHttpMessageConverter` на `RestTemplate`.

Обработка ошибок

Различные обработчики в нашем контроллере `CustomerRestController` занимаются существующими записями, при отсутствии которых выдают исключение. Эти обработчики могут явно возвратить `ResponseEntity` с кодом статуса 404, но данный вид обработки ошибок быстро становится повторяющимся, когда дублируется по множеству обработчиков. Логика обработки ошибок требует централизованного управления. Масштабирование начинается с согласованности, а она происходит от автоматизации. Spring MVC поддерживает в своих контроллерах отслеживание условий возникновения ошибок и реагирование на них с помощью методов, имеющих аннотацию `@ExceptionHandler`. Обычно обработчики `@ExceptionHandler` находятся в том же компоненте контроллера, что и обработчики, способные выдавать исключения. Но последние не могут совместно использоваться сразу в нескольких контроллерах. Если нужно получить централизованную логику обработки ошибок, то следует применить компонент `@ControllerAdvice`. Он относится к особому типу компонентов, которые могут вводить поведение (и реагировать на исключения) в любое количество контроллеров. Они представляются естественным местом для размещения централизованных обработчиков `@ExceptionHandler`.

Ошибки — важная часть эффективного API. Они должны однозначно и кратко указывать на условия своего возникновения, чтобы автоматизировать клиентов и помочь тем специалистам, которым в конечном итоге придется устранять эти ошибки или как минимум разбираться в последствиях их возникновения. Крайне важно, чтобы описания ошибок приносили максимальную пользу. Коды статуса HTTP при всей своей значимости *не дают* конкретной полезной информации. По сложившейся практике коды ошибок возвращаются с каким-либо представлением ошибки и сообщением, предназначенным для глаз специалистов. Официального стандарта относительно способа кодирования ошибок не существует, за исключением кодов статуса HTTP, но стандартом де-факто является тип содержимого `application/vnd.error` (https://github.com/blongden/vnd.error); в Spring HATEOAS поддерживается именно это представление ошибок. Добавьте к пути к классам classpath `org.springframe work.boot : spring-boot-starter-hateoas`. В Spring HATEOAS включены `VndError` и `VndErrors` в качестве объектов-конвертов, представляющих отдельно взятую ошибку или соответственно коллекцию ошибок.

Рассмотрим пример 6.9, в котором показан простой компонент `@ControllerAdvice`, перехватывающий все выдачи исключений и соответствующим образом их обрабатывающий, возвращая статус HTTP и `VndError`.

Пример 6.9. Централизованная обработка ошибок с помощью @ControllerAdvice

```
package demo;

import org.springframework.hateoas.VndErrors;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestController;

import java.util.Optional;

@ControllerAdvice(annotations = RestController.class)
public class CustomerControllerAdvice {

 ❶
 private final MediaType vndErrorMediaType = MediaType
  .parseMediaType("application/vnd.error");

 ❷
 @ExceptionHandler(CustomerNotFoundException.class)
 ResponseEntity<VndErrors> notFoundException(CustomerNotFoundException e) {
  return this.error(e, HttpStatus.NOT_FOUND, e.getCustomerId() + "");
 }

 @ExceptionHandler(IllegalArgumentException.class)
 ResponseEntity<VndErrors> assertionException(IllegalArgumentException ex) {
  return this.error(ex, HttpStatus.NOT_FOUND, ex.getLocalizedMessage());
 }

 ❸
 private <E extends Exception> ResponseEntity<VndErrors> error(E error,
  HttpStatus httpStatus, String logref) {
  String msg = Optional.of(error.getMessage()).orElse(
   error.getClass().getSimpleName());
  HttpHeaders httpHeaders = new HttpHeaders();
  httpHeaders.setContentType(this.vndErrorMediaType);
  return new ResponseEntity<>(new VndErrors(logref, msg), httpHeaders,
   httpStatus);
 }
}
```

❶ Встроенного медиатипа для `application/vnd.error` нет, поэтому создаем собственный.

❷ В этом классе два обработчика, и оба отвечают на возможные исключения, выданные в других компонентах. Конкретно этот обработчик отвечает на наше исключение, выдаваемое при неудачном поиске, — `CustomerNotFoundException`, которое, в свою очередь, просто расширяет исключение `RuntimeException`.

❸ Метод `error` создает объект `ResponseEntity`, содержащий информационное наполнение `VndErrors`, и транспортирует желаемый медиатип и код статуса. Просто и полезно.

Гипермедиа

Созданный API будет вполне работоспособен, если клиент знает, к какой конечной точке обращать вызов, выполнять его и с помощью какого HTTP-метода делать это. С последним небольшую помощь оказывает HTTP-метод `OPTIONS`, позволяя понять, какой HTTP-метод доступен для данного ресурса. А все остальное подразумевается, и остается только надежда на наличие *где-нибудь* подробной документации. Проблема ее использования в том, что не все содержат инструкции в актуальном состоянии, а еще меньшее количество специалистов удосуживается их изучить. В конечном итоге документация не синхронизируется с фактической спецификацией сервиса, определяемой его кодом.

Тезис доктора Филдинга (Fielding) относится главным образом к положениям гипермедиа. В нем говорится об идее ссылок в повседневных ресурсах (элементах `<link/>` в разметке HTML), предоставляющих информацию клиенту — HTML-браузерам и их пользователям, которая в конце концов приводит к изменениям в состоянии приложения. Вы можете открыть сессию в Amazon.com, войти в поиск, найти подходящий товар для покупки, нажать кнопку добавления его в корзину, выбрать подсчет общей стоимости, а затем оплату, и все действия совершить за счет переходов от одного HTTP-ресурса к другому. В итоге получится, что вы изменили состояние системы. Эти ссылки направляют вас по верному пути, а поскольку они появляются только там, где уместны (ссылки на возврат еще неоплаченных товаров вы не увидите!), то ведут по *правильному* маршруту. Данные шаги от одного ресурса к другому предполагают наличие *протокола*: серии шагов и взаимодействий в программе, предпринимаемых для того, чтобы дойти до конечного пункта. Эта система работает, так как люди могут разбираться в ссылках, выделять их на основе пользовательского опыта и предполагать последовательность шагов на основе получаемой за счет конструкции сайта зрительной информации, чтобы разобраться в ее целесообразности.

С другой стороны, клиенты в виде машин не так умны, как люди. Там, где люди щелкают на ссылках, выведенных на экран с помощью элементов `<a/>`, автоматизированные клиенты осуществляют переход по элементам `<link/>` (даже притом, что люди не могут щелкать кнопкой мыши на данных элементах). У этих ссылок имеются два важных атрибута: `rel` и `href`. Существует множество применений

этого элемента, но наиболее часто его используют для загрузки данных таблиц стилей, показанной в примере 6.10.

Пример 6.10. Загрузка таблицы стилей документа

```
<link rel="stylesheet" href="bootstrap.css">
```

Эти ссылки предоставляют метаданные о включаемом в них ресурсе. Браузер сначала обращается к атрибуту `rel`, чтобы разобраться в том, к какому сорту ресурса ведет ссылка. При необходимости браузер следует указаниям атрибута `href` с целью загрузить тот ресурс, на который указывает ссылка. Клиент использует атрибут `rel`, чтобы определить *актуальность* обозначаемого ссылкой ресурса. Ресурс, получаемый по ссылке, может находиться где угодно! *Заранее* браузер не выстраивает на этот счет никаких предположений, он просто следует по гиперссылке `href`, содержащейся в элементе `<link/>`, если таковая доступна. Таким образом, клиент *отделен* от местоположения ресурса.

Клиент (браузер) может по-прежнему использовать API, даже если местоположения ресурсов в последнем изменились. Клиент способен определить, к какому ресурсу проследовать, изучив содержимое атрибута `rel`, чтобы разобраться в актуальности того ресурса, к которому ведет ссылка. Атрибут `rel` становится контрактом: пока не будет нарушена его стабильность, клиент никогда не даст сбой.

Вернемся к протоколу оформления, имеющемуся в API, наподобие того, что используется на сайте Amazon.com. Конечно же, элемент `<link/>` относится к XML-элементу, но нет никакого смысла придерживаться такого же подхода, поскольку информационное наполнение, сопровождаемое расширяющими ссылками, не сможет работать на нас в других системах кодирования, таких как JSON. Для описания ссылочных элементов в формате JSON существует даже стандарт де-факто. Специализацией JSON с типом содержимого `application/hal+json` является язык гипертекстовых приложений — HAL (Hypertext Application Language) (http://stateless.co/hal_specification.html). Так что вместо того, чтобы тратить время на создание специальных структур для описания REST-ресурсов, можно положиться на HAL и реализовать HATEOAS (пример 6.11).

Для представления гипермедиа существуют и другие конкурирующие стандарты. HAL просто набрал популярность и доказал свою практичность.

Пример 6.11. Ранее используемый REST-ресурс Customer, получивший описание с помощью HAL

```
{
  "id":1,
  "firstName":"Mark",
  "lastName":"Fisher",
```

```
  "_links": {
    "self": {"href":"http://localhost:8080/v2/customers/1"},
    "profile-photo": {"href":"http://localhost:8080/customers/1/photo/"}
  }
}
```

API с таким описанием способен предоставить переходы для любого HAL-совместимого клиента. В Spring Boot поддерживается очень удобный HAL-клиент *HAL Browser*. Его можно использовать в любом веб-приложении Spring Boot, если добавить зависимости `org.springframework.boot : springboot-starter-actuator` и `org.springframework.data : spring-data-rest-hal-browser`. Он регистрируется в конечной точке актуатора Spring Boot, поэтому понадобится и актуатор (рис. 6.1).

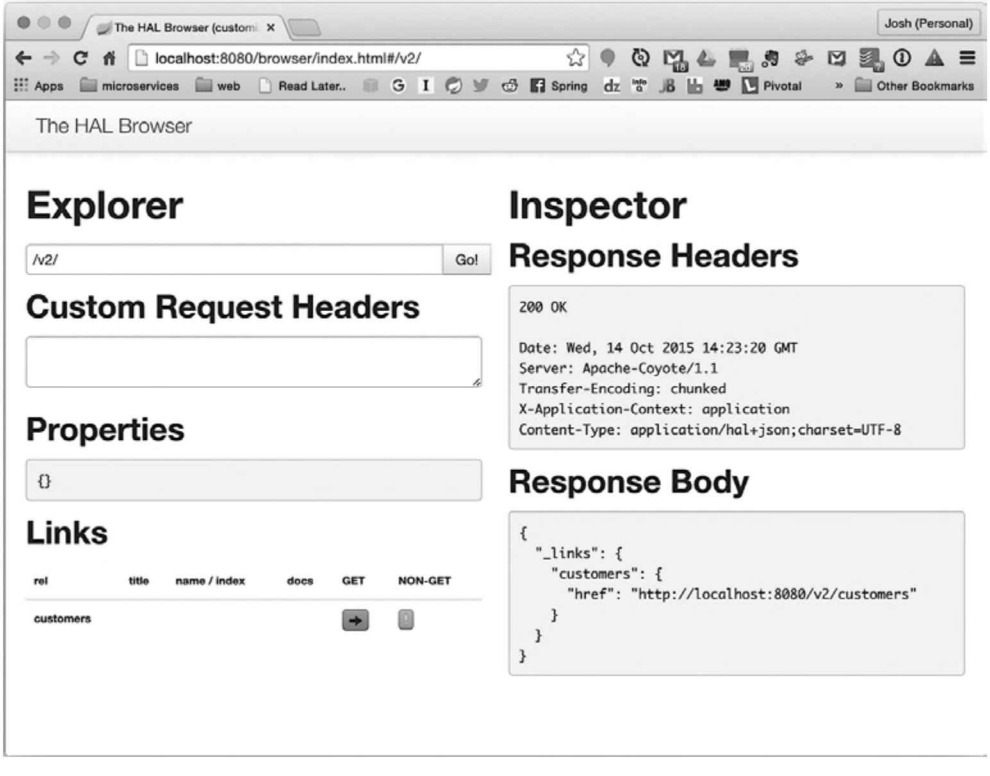


Рис. 6.1. Конечная точка HAL browser Actuator с настройками по умолчанию

Spring HATEOAS (https://spring.io/projects/spring-hateoas) является надстройкой над Spring MVC и предоставляет необходимые подключения для потребления и описания ресурсов в понятиях информационного наполнения и связанных с ним ссылок. Эта надстройка зависит от Spring MVC; вместо потребления и производства

ресурсов типа `T` вы используете и производите ресурсы типа `org.springframework.hateoas.Resource<T>` для одиночных объектов или `org.springframework.hateoas.Resources<T>` для коллекций.

В Spring HATEOAS `Resource` — объект-конверт, который, в свою очередь, содержит информационное наполнение и набор связанных с ним ссылок. Вам придется часто выполнять преобразования объектов типа `T` в объекты типа `Resource<T>` или `Resources<T>`. Создайте код рецепта преобразования в принадлежащем Spring HATEOAS экземпляре `org.springframework.hateoas.ResourceAssembler`. Вернемся к нашему контроллеру `CustomerRestController` и добавим в него гипермедиа и Spring HATEOAS (пример 6.12).

Пример 6.12. Переделанный CustomerHypermediaRestController

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.hateoas.Link;
import org.springframework.hateoas.Resource;
import org.springframework.hateoas.Resources;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

//@formatter:off
import org.springframework.web.servlet.mvc.method
        .annotation.MvcUriComponentsBuilder;
import org.springframework.web.servlet.support
        .ServletUriComponentsBuilder;
//@formatter:on

import java.net.URI;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

❶
@RestController
@RequestMapping(value = "/v2", produces = "application/hal+json")
public class CustomerHypermediaRestController {

 private final CustomerResourceAssembler customerResourceAssembler; ❷

 private final CustomerRepository customerRepository;

 @Autowired
 CustomerHypermediaRestController(CustomerResourceAssembler cra,
                                  CustomerRepository customerRepository) {
```

```
 this.customerRepository = customerRepository;
 this.customerResourceAssembler = cra;
}

❸
@GetMapping
ResponseEntity<Resources<Object>> root() {
 Resources<Object> objects = new Resources<>(Collections.emptyList());
 URI uri = MvcUriComponentsBuilder
  .fromMethodCall(MvcUriComponentsBuilder.on(getClass()).getCollection())
  .build().toUri();
Link link = new Link(uri.toString(), "customers");
objects.add(link);
return ResponseEntity.ok(objects);
}

❹
@GetMapping("/customers")
ResponseEntity<Resources<Resource<Customer>>> getCollection() {
 List<Resource<Customer>> collect = this.customerRepository.findAll().
     stream()
  .map(customerResourceAssembler::toResource)
  .collect(Collectors.<Resource<Customer>>toList());
 Resources<Resource<Customer>> resources = new Resources<>(collect);
 URI self = ServletUriComponentsBuilder.fromCurrentRequest().build().toUri();
 resources.add(new Link(self.toString(), "self"));
 return ResponseEntity.ok(resources);
}

@RequestMapping(value = "/customers", method = RequestMethod.OPTIONS)
ResponseEntity<?> options() {
 return ResponseEntity
  .ok()
  .allow(HttpMethod.GET, HttpMethod.POST, HttpMethod.HEAD, HttpMethod.
      OPTIONS,
   HttpMethod.PUT, HttpMethod.DELETE).build();
}

@GetMapping(value = "/customers/{id}")
ResponseEntity<Resource<Customer>> get(@PathVariable Long id) {
 return this.customerRepository.findById(id)
  .map(c -> ResponseEntity.ok(this.customerResourceAssembler.toResource(c)))
   .orElseThrow(() -> new CustomerNotFoundException(id));
}

@PostMapping(value = "/customers")
ResponseEntity<Resource<Customer>> post(@RequestBody Customer c) {
 Customer customer = this.customerRepository.save(new Customer(c
   .getFirstName(), c.getLastName()));
```

```
  URI uri = MvcUriComponentsBuilder.fromController(getClass())
   .path("/customers/{id}").buildAndExpand(customer.getId()).toUri();
  return ResponseEntity.created(uri).body(
   this.customerResourceAssembler.toResource(customer));
 }

 @DeleteMapping(value = "/customers/{id}")
 ResponseEntity<?> delete(@PathVariable Long id) {
  return this.customerRepository.findById(id).map(c -> {
   customerRepository.delete(c);
   return ResponseEntity.noContent().build();
  }).orElseThrow(() -> new CustomerNotFoundException(id));
 }

 @RequestMapping(value = "/customers/{id}", method = RequestMethod.HEAD)
 ResponseEntity<?> head(@PathVariable Long id) {
  return this.customerRepository.findById(id)
   .map(exists -> ResponseEntity.noContent().build())
   .orElseThrow(() -> new CustomerNotFoundException(id));
 }

 @PutMapping("/customers/{id}")
 ResponseEntity<Resource<Customer>> put(@PathVariable Long id,
  @RequestBody Customer c) {
  Customer customer = this.customerRepository.save(new Customer(id,
   c.getFirstName(), c.getLastName()));
  Resource<Customer> customerResource =
   this.customerResourceAssembler.toResource(customer);
  URI selfLink = URI.create(ServletUriComponentsBuilder.fromCurrentRequest()
   .toUriString());
  return ResponseEntity.created(selfLink).body(customerResource);
 }
}
```

❶ Ответы отправляются не в виде обычного прежнего `application/json`, а `application/hal+json`.

❷ Вы внедрили специализированную реализацию `ResourceAssembler`, чтобы упростить преобразование `T` в `Resource<T>`.

❸ Корневая конечная точка просто возвращает коллекцию элементов `Link`, работающих как своеобразное меню с предоставлением возможности переходов, начиная их с `/` и продолжая без каких-либо *предварительных* сведений.

❹ Метод `GET` в отношении коллекции `customers` формально такой же, как и прежде, но мы добавили к ресурсу Spring HATEOAS `Link`. Данный элемент создан с использованием имеющегося в Spring MVC весьма удобного средства `ServletUriComponentsBuilder`.

Все остальное в этом классе в основном имеет прежнее содержимое. Большинство методов в `CustomerHypermediaRestController` для выполнения своей работы по-

лагаются на внедренный класс `CustomerResourceAssembler`. Определение данного класса показано в примере 6.13.

Пример 6.13. Определение класса CustomerResourceAssembler, предоставляющего исходный набор ссылок для заданного объекта Customer

```
package demo;

import org.springframework.hateoas.Link;
import org.springframework.hateoas.Resource;
import org.springframework.hateoas.ResourceAssembler;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.mvc.method.annotation
        .MvcUriComponentsBuilder;

import java.net.URI;

@Component
class CustomerResourceAssembler implements
 ResourceAssembler<Customer, Resource<Customer>> {

 @Override
 public Resource<Customer> toResource(Customer customer) {

  Resource<Customer> customerResource = new Resource<>(customer);
  URI photoUri = MvcUriComponentsBuilder
   .fromMethodCall(
    MvcUriComponentsBuilder.on(CustomerProfilePhotoRestController.class).read(
     customer.getId())).buildAndExpand().toUri();

  URI selfUri = MvcUriComponentsBuilder
   .fromMethodCall(
    MvcUriComponentsBuilder.on(CustomerHypermediaRestController.class).get(
     customer.getId())).buildAndExpand().toUri();

  customerResource.add(new Link(selfUri.toString(), "self"));
  customerResource.add(new Link(photoUri.toString(), "profile-photo"));
  return customerResource;
 }
}
```

Класс `ResourceAssembler` — вполне естественное место для включения и исключения ссылок по условиям состояния объекта. Здесь производится проверка наличия профильного фото и, в соответствии со складывающимися условиями, включение ссылки для этой фотографии. При взаимодействии с данной системой доступные ссылки описывают протокол клиента.

Составной частью результата наряду с командами вызова, предоставляемыми HTTP (и HTTP `OPTIONS`), становится гипермедиа, позволяющее легко перемещаться

по этому API, не прибегая к использованию какой-либо *предварительной* документации.

Гипермедиапредставление с применением HAL — один из многих популярных вариантов. В 2009 году был представлен язык описания веб-приложений — Web Application Description Language (https://www.w3.org/Submission/wadl/), являющийся широкомасштабным подходом к описанию HTTP-сервисов. Спецификация ALPS (http://alps.io/spec/) дебютировала совсем недавно и определяет как возможные переходы состояния, так *и* атрибуты ресурсов, участвующих в этих переходах состояния, способом, которому безразличен медиатип. Автоматическая поддержка ALPS предоставляется Spring Data REST.

Медиатип и схема. Чтобы ввести гипермедиа в REST API, мы рассмотрели вопросы применения Spring HATEOAS. Целью ставилось предоставление клиенту как можно большего объема информации, позволяющей выполнять переходы в API без каких-либо предварительных сведений. HTTP уже предписывает использование возможных методов (`PUT`, `POST`, `GET`, `DELETE` и т. д.), а гипермедиа информирует, куда можно направиться за данным ресурсом.

HTTP включает тип содержимого. Теоретически последний дает достаточно информации любому клиенту о характере обслуживаемого содержимого и о способах работы с ним. Но на практике этого недостаточно. Тип содержимого `image/jpg` сообщает о возможности декодирования возвращаемых сервисом байтов с помощью средства чтения изображений в JPEG-формате. Тип содержимого `application/json` или `application/hal+json` сообщает, что байты, возвращенные ресурсом, можно прочитать благодаря средству чтения JSON-формата и что там могут содержаться закодированные в HAL-формате ссылки. Эти типы содержимого *не сообщают*, соответствует ли JSON-документ конкретной структуре. Для этого понадобится такое понятие, как схема.

XML предлагает XML-схему (https://www.w3.org/XML/Schema), закрепляющую структуру XML-ресурса. У представления Google Protocol Buffer также имеется схема. Для JSON ситуация не настолько очевидная, но на самом деле существует множество альтернатив наподобие JSON Schema (http://json-schema.org/), которая включает способ определить, какова структура информационного наполнения.

Управление версиями API

Единственное, что известно доподлинно, — изменения будут и это неизбежно. Несомненно, самым большим преимуществом архитектуры, оптимизированной для выполнения в облаке, является то, что она поддерживает небольшие пакетные изменения (микросервисы), способные развиваться быстро и независимо друг от друга. Изменения следует рассматривать в *позитивном* ключе, поскольку они подразумевают развитие! Нам хочется изменений, но не хочется вносить их в наши

API или создавать помехи для других частей системы или сторонних клиентов. Нужно найти способы развития наших сервисов, не нарушающие без необходимости работу клиентов.

Можно просто попытаться ничего не сломать. Это несколько похоже на избавление процесса выпуска от рисков за счет отказа от самого выпуска! Но выход все же есть. Если изменяются лишь особенности реализации, то область визуального отображения API не нарушится. Крайне важно, чтобы потенциальные нарушения работы обнаруживались как можно раньше. Защитить API от случайных нарушений призвано *тестирование контрактов, ориентированных на потребителя* (Consumer-Driven Contract Testing) (более подробно такое тестирование рассмотрено в подразделе «Тестирование контрактов, ориентированных на потребителя» раздела «Сквозное тестирование» главы 4). Непрерывное интегрирование предоставляет ценный цикл обратной связи, который можно использовать для постоянной защиты от случайных нарушений работы API, тестируя как можно большее количество клиентов в отношении изменяемых API.

Но все подвержено изменениям. Для снижения чувствительности к изменениям клиенты и сервисы должны обладать определенной гибкостью. Мартин Фаулер (Martin Fowler) упоминал идею *лояльного читателя* (https://martinfowler.com/bliki/TolerantReader.html) — API-клиента, проявляющего заботу о том, чтобы избежать чувствительности к изменениям информационного наполнения. Клиент может рассчитывать на поиск в информационном наполнении XML-элемента `order-id` или JSON-элемента. Запросы JSON-Path или XPath позволяют клиентам находить элементы и шаблоны в структуре, оставаясь безразличными к порядку или глубине шаблона в исходной структуре.

Закон Постела, известный также как *принцип надежности* (https://en.wikipedia.org/wiki/Robustness_principle), гласит: реализации сервиса должны быть консервативными в том, что они производят, но либеральными в отношении того, что принимают от других. Если сервису для его работы нужно только информационное наполнение в виде поднабора из заданных сообщений, то он не станет возражать против присутствия в сообщении дополнительной информации, для которой он не может найти немедленное применение.

Пока изменения носят не разрушительный, а эволюционный и дополняющий характер, повышение надежности дается легко. Если же элемент, наличие которого ожидалось в версии 1 сервиса, должен внезапно исчезнуть в версии 2, работа клиентов будет, скорее всего, нарушена. В сервисе необходимо конкретизировать предположения о том, какие клиенты смогут с ним взаимодействовать. Одним из подходов к этому является *семантическое управление версиями*. К семантическим относится одна из версий в форме `MAJOR.MINOR.PATCH`. Версия `MAJOR` должна изменяться только при наличии в API разрушительных изменений по отношению к его предыдущей версии. Версии `MINOR` надлежит изменяться только при условии, что API получил развитие, но существующие клиенты могут продолжить работать

с ним. Версия изменений PATCH сигнализирует об устранении недочетов в существующих функциональных возможностях.

Семантическое управление версиями показывает клиентам, какая версия API доступна, но, даже если клиенты оповещены о характере версии, они могут быть не готовы к переходу к новой версии. Необходимость для клиентов обновляться при обновлении сервиса угрожает одному из основных преимуществ микросервисов — возможности независимого развития. Следовательно, иногда будет возникать надобность содержать несколько версий API. Один из подходов способен предусматривать параллельное развертывание и обслуживание двух различных кодовых баз, но может очень быстро стать слишком обременительным. Вместо этого стоит присмотреться к параллельному содержанию разных версий API в одной и той же кодовой базе. Попробуйте по возможности преобразовать запросы, предназначенные к отправке к старым конечным точкам, и направить их к новым точкам, чтобы все запросы в конце концов были перенесены в одно и то же место. Это снизит объем работ по тестированию.

Клиент должен будет просигнализировать сервису, с какой из версий API он намерен общаться. В REST API имеется ряд устоявшихся подходов: закодировать версию в URL, в произвольном (и специализированном) HTTP-заголовке или в виде составной части типа содержимого, указываемого в предназначенном для запроса заголовке Accept (пример 6.14).

Пример 6.14. VersionedRestController демонстрирует несколько подходов к управлению версиями REST API

```
package demo;

import org.springframework.web.bind.annotation.*;

//@formatter:off
import static org.springframework.http
        .MediaType.APPLICATION_JSON_VALUE;
//@formatter:on

@RestController
@RequestMapping("/api")
public class VersionedRestController {

 //@formatter:off
 public static final String V1_MEDIA_TYPE_VALUE
         = "application/vnd.bootiful.demo-v1+json";

 public static final String V2_MEDIA_TYPE_VALUE
         = "application/vnd.bootiful.demo-v2+json";

 //@formatter:on

 private enum ApiVersion {
```

```
 v1, v2
}

public static class Greeting {

 private String how;

 private String version;

 public Greeting(String how, ApiVersion version) {
  this.how = how;
  this.version = version.toString();
 }

 public String getHow() {
  return how;
 }

 public String getVersion() {
  return version;
 }
}

❶
@GetMapping(value = "/{version}/hi", produces = APPLICATION_JSON_VALUE)
Greeting greetWithPathVariable(@PathVariable ApiVersion version) {
 return greet(version, "path-variable");
}

❷
@GetMapping(value = "/hi", produces = APPLICATION_JSON_VALUE)
Greeting greetWithHeader(@RequestHeader("X-API-Version") ApiVersion version) {
 return this.greet(version, "header");
}

❸
@GetMapping(value = "/hi", produces = V1_MEDIA_TYPE_VALUE)
Greeting greetWithContentNegotiationV1() {
 return this.greet(ApiVersion.v1, "content-negotiation");
}

❹
@GetMapping(value = "/hi", produces = V2_MEDIA_TYPE_VALUE)
Greeting greetWithContentNegotiationV2() {
 return this.greet(ApiVersion.v2, "content-negotiation");
}

private Greeting greet(ApiVersion version, String how) {
 return new Greeting(how, version);
}
}
```

❶ При этом подходе версия закодирована в URL; Spring MVC автоматически отобразит URL-параметры на экземпляры соответствующего перечисляемого типа `ApiVersion`. Протестируйте его с помощью команды `curl http://localhost:8080/api/V2/hi`.

❷ При этом подходе происходит автоматическое преобразование заголовков запроса в экземпляры соответствующего перечисляемого типа `ApiVersion`. Протестируйте его с помощью команды `curl -H "X-API-Version:V1" http://localhost:8080/api/hi`.

❸ Этот обработчик контроллера обрабатывает запросы для медиатипа `application/vnd.bootiful.demo-v1+json`. Протестируйте его с помощью команды `curl -H "Accept:application/vnd.bootiful.demo-v1+json" http://localhost:8080/api/hi`.

❹ Этот обработчик контроллера обрабатывает запросы для медиатипа `application/vnd.bootiful.demo-v2+json`. Протестируйте его с помощью команды `curl -H "Accept:application/vnd.bootiful.demo-v2+json" http://localhost:8080/api/hi`.

Документирование REST API

Манифест разработки легко перестраиваемого ПО (agile manifesto) провозглашает принцип приоритета создания работоспособной программы над написанием подробной документации. То есть сначала нужно заставить ПО работать, а затем задокументировать его. Куда лучше обзавестись рабочей программой, которая сама правильно объясняет принципы своей работы, чем располагать документацией, в которой может содержаться дезинформация. Код сродни живому организму, и, как мы себе это представляем в какой-то момент на основе сложившегося опыта, документация, разрабатываемая параллельно с созданием кода, склонна утрачивать синхронизированность с реалиями кода. Документация служит в качестве карты, а код является нанесенной на эту карту местностью. Если вы заблудились, то лучше быть знакомым с местностью, чем с картой!

Документация, без сомнения, по-прежнему вполне естественная и удобная вещь. С присущим нам вечным оптимизмом мы, инженеры, проделали огромную работу, чтобы объединить документацию (то есть информацию по API) с самим интерфейсом. Первым широкомасштабным примером такого подхода явился стандарт JavaDoc, хотя с тех пор альтернативными средствами подобного рода обзавелись и другие языки. JavaDoc обслуживается разработчиком, поскольку этот стандарт предусматривает сосуществование документации с кодом. Есть надежда, что разработчики всегда будут следить за тем, отражает ли документация реалии кода, поскольку она живет вместе с ним. Разумеется, мы знаем, что такое случается не всегда! Помимо JavaDoc, вывести всевозможную информацию о коде под силу и другим весьма непростым инструментам.

Множество подсказок насчет использования грамотно спроектированного API содержится в гипермедиа. HTTP-метод `OPTION` описывает, какие именно методы HTTP поддерживает заданный ресурс. Гипермедиассылки информируют нас о родственных связях ресурса. Схема может предоставить сведения о структуре допустимого для данного ресурса информационного наполнения. Но подходящего способа документирования ожидаемых параметров или заголовков HTTP-запроса не существует, как нет и способа объяснить мотивацию или намерение. Документация в лучшем случае описывает мотивацию, а не реализацию. По-прежнему отсутствует максимально автоматизированный способ полного охвата порядка использования API, позволяющий получить гарантии синхронизированности документации с реалиями документируемого кода, то есть с местностью.

В нашем распоряжении есть несколько вариантов, например Swagger. Этот инструмент требует внедряемых изменений самого кода. Swagger полагается на строковые данные, встроенные в код Java, в аннотации самого API. Поддерживать в нем понятный человеку стиль весьма обременительно, а иногда инструмент не справляется с адекватным сбором данных о намерениях API, поскольку предпринимает попытку автоматически отобразить характерные для языка конструкции на HTTP-контракт, подразумеваемый этими конструкциями.

В Spring RESTDocs (https://spring.io/projects/spring-restdocs) применяется другой подход. Этот инструмент в качестве оформления включен в среду тестирования Spring MVC и в виде свободной формы, выраженной в Asciidoctor, в разметке, воспринимаемой настолько естественно, что мы с ее помощью написали эту книгу!

Добавьте Spring REST Docs к пути к классам (classpath) в виде зависимости в области видимости тестов: `org.springframework.restdocs : spring-restdocs-mockmvc`. Выполните документирование вашего API, протестировав его. В конечном итоге мы попытаемся собрать данные о порядке взаимодействий, подтверждаемых в ходе выполнения теста. Рассмотрим пример 6.15.

Пример 6.15. Для отработки действий с API в ApiDocumentation.java используется тестирование API с применением Spring MVC

```
package demo;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.
AutoConfigureRestDocs;
import org.springframework.boot.test.autoconfigure.web.servlet.
AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
```

```
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import javax.servlet.RequestDispatcher;

import static org.hamcrest.Matchers.is;
import static org.hamcrest.Matchers.notNullValue;
import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.
document;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.
MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.status;
//@formatter:off
❶
@AutoConfigureRestDocs(outputDir = "target/generated-snippets")
@RunWith(SpringRunner.class)
@SpringBootTest(classes = Application.class,
        webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureMockMvc
public class ApiDocumentation {
//@formatter:off

 ❷
 // @Rule public final RestDocumentation restDocumentation =
 // new RestDocumentation(
 // "target/generated-snippets");

 @Autowired
 private MockMvc mockMvc;

 @Test
 public void errorExample() throws Exception {
  this.mockMvc
   .perform(
    get("/error")
     .contentType(MediaType.APPLICATION_JSON)
     .requestAttr(RequestDispatcher.ERROR_STATUS_CODE, 400)
     .requestAttr(RequestDispatcher.ERROR_REQUEST_URI, "/customers")
     .requestAttr(RequestDispatcher.ERROR_MESSAGE,
      "The customer 'http://localhost:8443/v1/customers/123' does not exist"))
   .andDo(print()).andExpect(status().isBadRequest())
   .andExpect(jsonPath("error", is("Bad Request")))
   .andExpect(jsonPath("timestamp", is(notNullValue())))
```

```
    .andExpect(jsonPath("status", is(400)))
    .andExpect(jsonPath("path", is(notNullValue())))
    .andDo(document("error-example"));  ❸
  }

  @Test
  public void indexExample() throws Exception {
   this.mockMvc.perform(get("/v1/customers")).andExpect(status().isOk())
    .andDo(document("index-example"));
  }
}
```

❶ Правила тестирования предусматривают поддержку решения общих задач в тестах, на которые они настроены. В данном случае правило `RestDocumentation` приведет при запуске тестов к выдаче Asciidoctor-фрагментов в каталог `target/generated-snippets`.

❷ Статический метод `documentationConfiguration` просто регистрирует объект `RestDocumentation` с объектом `MockMvc`, используемым тестами для отработки действий с API.

❸ В строке `error-example`, передаваемой методу `document`, определяется логический идентификатор для автоматического создания документации для этого теста. В данном случае документация будет находиться *также* под папкой `error-example`, в которой можно найти файлы с расширением `.adoc`, например `curl-request.adoc`, `http-request.adoc`, `http-response.adoc` и т. д. Используйте это разнообразие созданных фрагментов в качестве включений при создании более объемного Asciidoctor-документа.

Обычно дополнительный модуль сборки тестов находит классы, имеющие окончание `Test`, и запускает их. Нужно настроить данный модуль таким образом, чтобы в него были включены и запущены также классы с окончанием `*Documentation` (пример 6.16).

Пример 6.16. Включение *Documentation.java в дополнительный модуль Maven Surefire

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <includes>
      <include>**/*Documentation.java</include>
    </includes>
  </configuration>
</plugin>
```

Побочным продуктом этих тестов будут автоматически созданные Asciidoctor-фрагменты, которые можно легко и просто включить в более объемный Asciidoctor-документ (пример 6.17).

Пример 6.17. Asciidoctor-фрагменты, автоматически созданные после запуска тестов

```
.
├── customer-get-example
│   ├── curl-request.adoc
│   ├── http-request.adoc
│   ├── http-response.adoc
│   ├── links.adoc
│   └── response-fields.adoc
├── customer-update-example
│   ├── curl-request.adoc
│   ├── http-request.adoc
│   ├── http-response.adoc
│   └── request-fields.adoc
├── customers-create-example
│   ├── curl-request.adoc
│   ├── http-request.adoc
│   ├── http-response.adoc
│   ├── links.adoc
│   ├── request-fields.adoc
│   └── response-fields.adoc
├── customers-list-example
│   ├── curl-request.adoc
│   ├── http-request.adoc
│   ├── http-response.adoc
│   └── response-fields.adoc
├── error-example
│   ├── curl-request.adoc
│   ├── http-request.adoc
│   ├── http-response.adoc
│   └── response-fields.adoc
└── index-example
    ├── curl-request.adoc
    ├── http-request.adoc
    └── http-response.adoc

6 directories, 26 files
```

Подходы могут меняться в зависимости от того, какой инструмент используется. Более подробные примеры будут представлены применительно к проекту Spring RESTDocs, но, как показано в примере 6.18, дополнительный модуль `org.asciidoctor : asciidoctor-maven-plugin` поможет получить и AsciiDoctor-документы, созданные с помощью Maven.

Пример 6.18. Применение дополнительного модуля Asciidoctor Maven

```
<plugin>
  <groupId>org.asciidoctor</groupId>
  <artifactId>asciidoctor-maven-plugin</artifactId>
  <version>1.5.2</version>
  <executions>
    <execution>
    <id>generate-docs</id>
```

```
    <phase>prepare-package</phase>
    <goals>
      <goal>process-asciidoc</goal>
    </goals>
    <configuration>
      ❶
      <backend>html</backend>
      <doctype>book</doctype>
      <attributes>
        <snippets>${project.build.directory}/generated-snippets</snippets> ❷
      </attributes>
    </configuration>
    </execution>
  </executions>
</plugin>
```

❶ В соответствии с настройками приложение станет искать в каталоге `src/main/resources` все подходящие файлы с расширением `.adoc`, а затем конвертировать их в форматы `.pdf` и `.html`.

❷ Чтобы упростить разрешение фрагментов и избежать повторений в самой документации, предоставляется атрибут, определяющий каталог `generated-snippets`.

Нам бы не хотелось перепечатывать излишне объемный пример использования Asciidoctor, так как его можно посмотреть в коде к данной книге. Но вы должны понять, что при желании получить документ Asciidoctor можно будет просто включить эти фрагменты в свой код. Фрагмент `response-fields.adoc` можно внести в него следующим образом: `include::{snippets}/error-example/response-fields.adoc[]`.

После написания документации и использования созданных фрагментов по мере необходимости будет очень удобно обслуживать документацию в качестве части ресурсов, обслуживаемых для самого приложения. Настройте ваше средство сборки на копирование сгенерированных данных, выводимых в каталог Spring Boot `src/main/resources/static`. Как это выглядит в случае применения Maven, показано в примере 6.19.

Пример 6.19. Настройка дополнительного модуля ресурсов Maven на включение созданной документации, хранящейся в статическом каталоге Spring Boot, чтобы она стала доступна по адресу http://localhost:8080/docs

```
<plugin>
  <artifactId>maven-resources-plugin</artifactId>
  <executions>
    <execution>
      <id>copy-resources</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>copy-resources</goal>
      </goals>
      <configuration>
      <outputDirectory>${project.build.outputDirectory}/static/docs
        </outputDirectory>
```

```
      <resources>
        <resource>
        <directory>${project.build.directory}/generated-docs</directory>
        </resource>
      </resources>
      </configuration>
    </execution>
  </executions>
</plugin>
```

В результате получается не содержащая ничего лишнего, всегда актуальная документация, которая говорит сама за себя и превращает сборку в зеленую.

Клиентская сторона

В этом разделе мы рассмотрим различные способы работы с REST API, как интерактивные, так и программные.

REST-клиенты для специализированного исследования и взаимодействия

Spring Boot справляется с поддержкой браузера HAL Browser в ваших собственных сервисах для взаимодействия с HAL REST API. Существует множество качественных, зачастую свободно распространяемых инструментов, поддерживающих взаимодействие с REST API. Ниже приведен ряд наших «фаворитов», которые могут помочь в ходе разработки.

- *Встраиваемое в Firefox расширение Poster*. Находится в свободном доступе. Весьма удобная утилита, размещаемая в нижнем правом углу окна браузера в виде небольшого желтого значка. Щелчок на этом значке приведет к появлению диалогового окна, в котором можно описать и выполнить HTTP-запросы. Данная утилита предоставляет простые функции для управления такими операциями, как выкладывание файлов и согласование содержимого (рис. 6.2).
- Команда `curl` имеет весьма почтенный возраст. Представляет собой утилиту командной строки, которая поддерживает любые HTTP-запросы с удобными ключами для отправки HTTP-заголовков, специализированных тел запросов и др. (рис. 6.3).

Авторы также стали поклонниками такого инструмента, как HTTPie (https://httpie.org/).

- *Расширение в Google Chrome под названием Advanced HTTP Client*. Позволяет описывать и выполнять HTTP-запросы, а также сохранять свои настройки для последующего повторного использования (рис. 6.4).

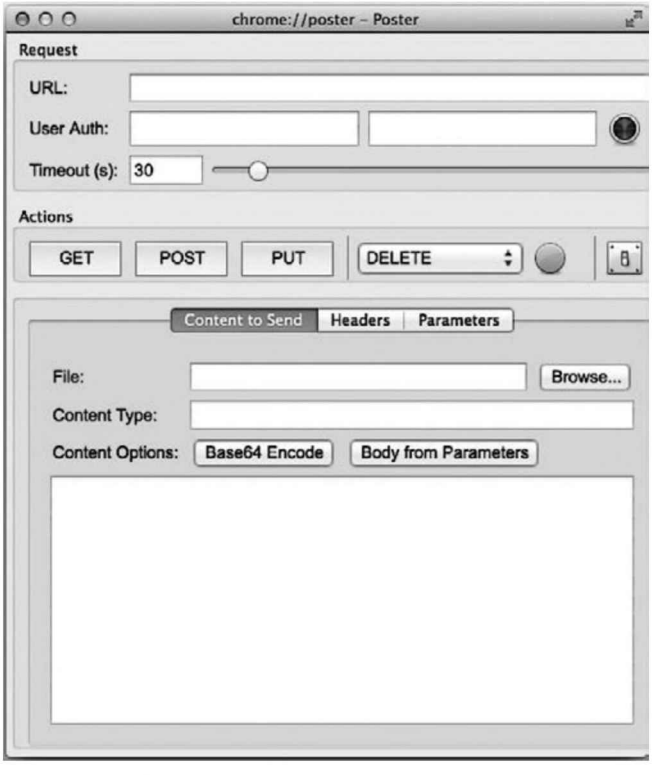


Рис. 6.2. Встраиваемое в Firefox расширение Poster предоставляет множество выгодных свойств

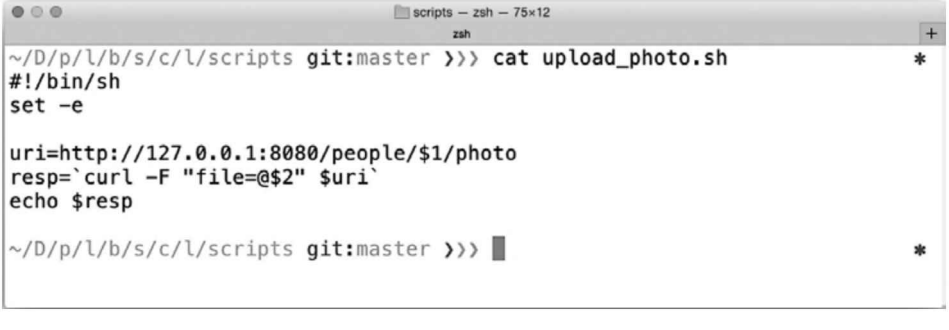


Рис. 6.3. Сценарий, использующий curl

- *Advanced HTTP Client.* Хорошо интегрируется с остальными инструментами панели Google Chrome Developer Tools (Инструменты разработчика Google Chrome). Можно открыть данную панель (имеется в каждой установке Google Chrome) и изучить содержимое вкладки Network (Сеть). В ней можно легко найти запущенный вами запрос, разработанный с помощью Advanced HTTP Client, а затем получить его экспортированным в виде команды `curl` (рис. 6.5).

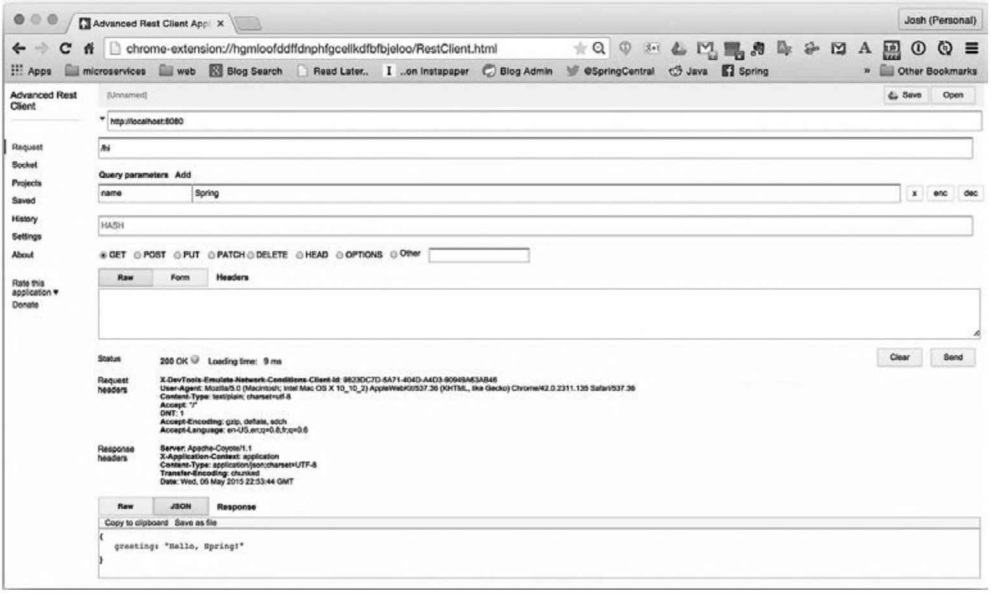


Рис. 6.4. Расширение Google Chrome под названием Advanced HTTP Client

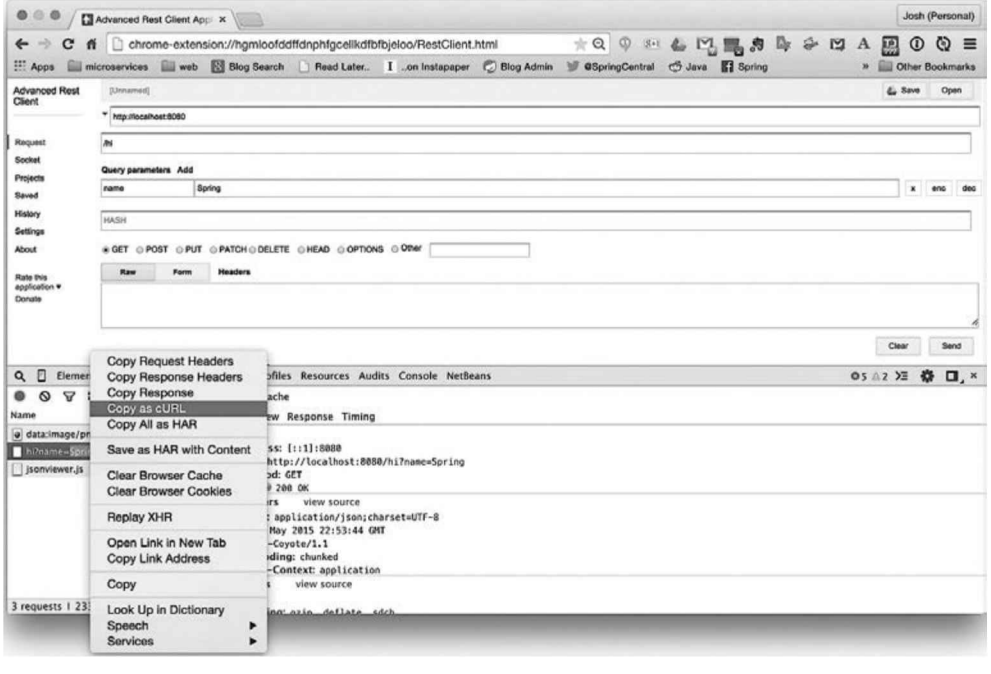


Рис. 6.5. Использование инструментов разработчика в Google Chrome для получения вызова команды curl, повторяющей HTTP-запрос, что очень удобно!

- *PostMan*. Весьма удобное расширение Chrome. Предоставляет сохраненные HTTP-запросы, синхронизацию устройств и коллекции, позволяя управлять коллекциями родственных HTTP-запросов. Становится одним из наших расхожих HTTP-клиентов для серьезной работы (рис. 6.6).

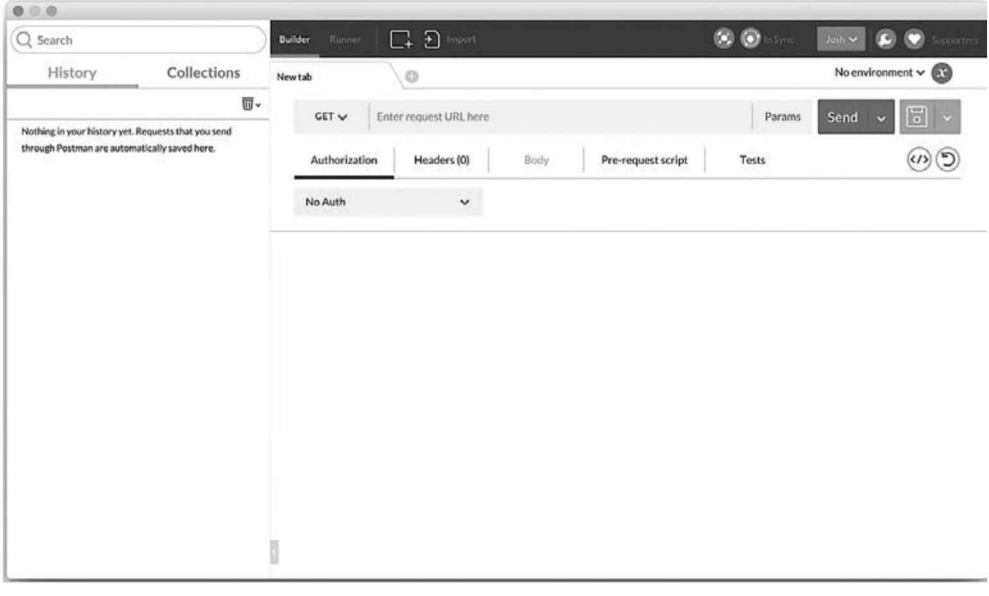


Рис. 6.6. HTTP-клиент PostMan

Шаблон RestTemplate

Имеющийся в Spring шаблон `RestTemplate` является многоцелевым HTTP-клиентом, поддерживающим самые распространенные HTTP-взаимодействия с помощью *шаблонных* методов; обычные HTTP-взаимодействия — `get`, `put`, `post` и т. д. — становятся едиными вставками. Шаблон поддерживает согласование содержимого, используя на стороне сервера ту же реализацию стратегии `HttpMessageConverter`, что и Spring MVC. Он может преобразовывать объекты в приемлемое тело HTTP-запроса, а также HTTP-ответы — в объекты. Шаблон поддерживает подключаемые перехватчики для централизованной обработки общих задач, например аутентификации с помощью HTTP BASIC и OAuth, GZip-сжатия и разрешения имен сервисов.

Исследуем REST API, основанный на Spring Data REST. В последнем автоматически создается гипермедийный API, оснащенный ссылками в стиле HAL, для взаимоотношений на основе указанных хранилищ Spring Data. У объектов `Movie` имеется один или несколько связанных с ними объектов `Actor`.

Нам нужен шаблон `RestTemplate`. Для облегчения конфигурирования экземпляра `RestTemplate` в Spring Boot включен `RestTemplateBuilder` (пример 6.20).

Пример 6.20. Конфигурирование RestTemplate

```
package actors;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpRequest;
import org.springframework.http.client.ClientHttpRequestExecution;
import org.springframework.http.client.ClientHttpRequestInterceptor;
import org.springframework.web.client.RestTemplate;

@Configuration
public class RestTemplateConfiguration {

 @Bean
 RestTemplate restTemplate() {

  Log log = LogFactory.getLog(getClass());

  ClientHttpRequestInterceptor interceptor = (HttpRequest request, byte[] body,
   ClientHttpRequestExecution execution) -> {
   log.info(String.format("request to URI %s with HTTP verb '%s'",
    request.getURI(), request.getMethod().toString()));
   return execution.execute(request, body);
  };

  return new RestTemplateBuilder() (1)
   .additionalInterceptors(interceptor).build();
  }

}
```

(1) Для конфигурирования перехватчиков, преобразователей и мн. др. `RestTemplateBuilder` содержит динамически изменяющийся DSL-язык.

В среде Spring шаблон `RestTemplate` представляет собой некую рабочую лошадку HTTP общего назначения. Задействуем ее в примере 6.21.

Пример 6.21. Использование RestTemplate

```
package actors;

import com.fasterxml.jackson.databind.JsonNode;
```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.hateoas.Resources;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

import java.net.URI;
import java.util.Collections;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

public class RestTemplateTest {

 private Log log = LogFactory.getLog(getClass());

 private URI baseUri;

 private ConfigurableApplicationContext server;

 private RestTemplate restTemplate;

 private MovieRepository movieRepository;

 private URI moviesUri;

 @Before
 public void setUp() throws Exception {

  this.server = new SpringApplicationBuilder()
   .properties(Collections.singletonMap("server.port", "0"))
   .sources(DemoApplication.class).run();

  int port = this.server.getEnvironment().getProperty("local.server.port",
   Integer.class, 8080);

  this.restTemplate = this.server.getBean(RestTemplate.class);
  this.baseUri = URI.create("http://localhost:" + port + "/");
  this.moviesUri = URI.create(this.baseUri.toString() + "movies");
```

```
  this.movieRepository = this.server.getBean(MovieRepository.class);
 }

 @After
 public void tearDown() throws Exception {
  if (null != this.server) {
   this.server.close();
  }
}

 @Test
 public void testRestTemplate() throws Exception {
   ❶
  ResponseEntity<Movie> postMovieResponseEntity = this.restTemplate
   .postForEntity(moviesUri, new Movie("Forest Gump"), Movie.class);
  URI uriOfNewMovie = postMovieResponseEntity.getHeaders().getLocation();
  log.info("the new movie lives at " + uriOfNewMovie);

  ❷
  JsonNode mapForMovieRecord = this.restTemplate.getForObject(uriOfNewMovie,
   JsonNode.class);
  log.info("\t..read as a Map.class: " + mapForMovieRecord);
  assertEquals(mapForMovieRecord.get("title").asText(),
   postMovieResponseEntity.getBody().title);

  ❸
  Movie movieReference = this.restTemplate.getForObject(uriOfNewMovie,
   Movie.class);
  assertEquals(movieReference.title, postMovieResponseEntity.getBody().title);
  log.info("\t..read as a Movie.class: " + movieReference);

  ❹
  ResponseEntity<Movie> movieResponseEntity = this.restTemplate.getForEntity(
   uriOfNewMovie, Movie.class);
  assertEquals(movieResponseEntity.getStatusCode(), HttpStatus.OK);
  assertEquals(movieResponseEntity.getHeaders().getContentType(),
   MediaType.parseMediaType("application/json;charset=UTF-8"));
  log.info("\t..read as a ResponseEntity<Movie>: " + movieResponseEntity);

  ❺

  //@formatter:off
  ParameterizedTypeReference<Resources<Movie>> movies =
          new ParameterizedTypeReference<Resources<Movie>>() {};
  //@formatter:on
  ResponseEntity<Resources<Movie>> moviesResponseEntity = this.restTemplate
   .exchange(this.moviesUri, HttpMethod.GET, null, movies);
  Resources<Movie> movieResources = moviesResponseEntity.getBody();
  movieResources.forEach(this.log::info);
```

```
  assertEquals(movieResources.getContent().size(), this.movieRepository.count());
  assertTrue(movieResources.getLinks().stream()
   .filter(m -> m.getRel().equals("self")).count() == 1);

 }
}
```

1. Шаблон `RestTemplate` предоставляет все ожидаемые вами удобные методы, поддерживающие общие HTTP-операции, например операцию `POST`.
2. Для `RestTemplate` можно задать возвращение ответов, преобразованных в соответствующее представление. Здесь мы задаем возвращаемое значение `JsonNode` в представлении, поддерживаемом библиотекой Jackson. `JsonNode` позволяет взаимодействовать с JSON, как это было возможно с XML DOM-узлом.
3. Там, где это возможно, преобразование применяется к объектам предметной области. Здесь задается отображение JSON (также с помощью библиотеки Jackson) на объект `Movie`.
4. Возвращаемое значение `ResponseEntity<T>` является оболочкой для информационного наполнения ответа, преобразованного рассмотренным ранее образом, а также для заголовков и информации о коде статуса из HTTP-ответа.
5. В REST API используется принадлежащий Spring HATEOAS объект `Resources<T>`, чтобы послужить в качестве коллекций экземпляров `Resource<T>`, которые, в свою очередь, являются оболочками для информационного наполнения и предоставляют ссылки. В предыдущих примерах мы указывали на возвращаемое значение путем предоставления литерала `.class`. Но в Java не предоставляется никакого способа для захвата чего-либо подобного `Resources<Movie>.class` с помощью литералов класса, поскольку общая информация во время компиляции пропадает из-за стирания типов, используемого в Java. Единственный способ сохранить общую информацию о параметрах — заключить ее в иерархию типов, создав подклассы. Эта схема, которая называется паттерном *токена типа* (type token pattern), поддерживается имеющимся в Spring типом `ParameterizedTypeReference`. Данный тип относится к абстрактным и поэтому должен быть выражен в виде подкласса. Фактически здесь `ptr`, при всей своей похожести на переменную экземпляра, является безымянным подклассом, который в ходе выполнения программы способен ответить на вопрос: «Каков ваш общий параметр с `Resources<Movie>`?» В шаблоне `RestTemplate` этим можно воспользоваться для правильной привязки возвращаемых в формате JSON значений.

В нашем гипермедийном API используется HAL, и последний выдает ссылки на соответствующие ресурсы. Ссылки в HAL-стиле подходят идеальным образом, поскольку за одной ссылкой идет другая, как в некой цепочке. Чтобы добраться до конкретного ресурса, следует пройти по ней, перескакивая с одной ссылки на другую, пока в конце концов вы не окажетесь в соответствующем ресурсе.

Это простой в изложении, но весьма шаблонный код. Если извлечение ресурса начинается с `/`, а добраться нужно до `/actors/search/by-movie?movie=Cars`, то придется перескакивать как минимум три раза:

- чтобы получить корневой ресурс `/` и найти ссылку поиска `search`;
- чтобы найти ссылку поиска для конечной точки поиска, запрашиваемой по названию фильма, `by-movie`;
- и чтобы фактически запустить поиск с передачей параметра запроса фильма `Cars`.

В Spring HATEOAS включен весьма удобный клиент `Traverson`, принимающий ссылочный путь и следующий по ссылкам, выдавая конечный результат (пример 6.22).

Клиент Spring Java основывается на коде библиотеки с таким же именем (https://github.com/traverson/traverson), поэтому данная возможность доступна и для клиентов браузера!

Пример 6.22. Настройка клиента Traverson с помощью сконфигурированного шаблона RestTemplate

```
package actors;

import
org.springframework.boot.context.embedded.EmbeddedServletContainer
InitializedEvent;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.event.EventListener;
import org.springframework.hateoas.MediaTypes;
import org.springframework.hateoas.client.Traverson;
import org.springframework.web.client.RestTemplate;

import java.net.URI;

@Configuration
public class TraversonConfiguration {

 private int port;

 private URI baseUri;

 //@formatter:off
 @EventListener
 public void embeddedPortAvailable(
    EmbeddedServletContainerInitializedEvent e) {
```

```
  this.port = e.getEmbeddedServletContainer().getPort();
  this.baseUri = URI.create("http://localhost:" + this.port + '/');
 }
 //@formatter:on

 ❶
 @Bean
 @Lazy
 Traverson traverson(RestTemplate restTemplate) {
  Traverson traverson = new Traverson(this.baseUri, MediaTypes.HAL_JSON);
  traverson.setRestOperations(restTemplate);
  return traverson;
 }
}
```

❶ Настройка клиента `Traverson` с помощью базового URI, экземпляров `MediaType`, которые клиенту требуется обработать, и экземпляра `RestTemplate`, чтобы делегировать ему обусловленные HTTP-вызовы.

Воспользуйтесь клиентом `Traverson`, предоставив в качестве пути описание серии ссылок, а затем проследуйте по этому пути (пример 6.23).

Пример 6.23. Использование клиента Traverson с шаблоном RestTemplate

```
package actors;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.hateoas.Resources;
import org.springframework.hateoas.client.Traverson;

import java.util.Collections;
import java.util.stream.Collectors;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

public class TraversonTest {

 private Log log = LogFactory.getLog(getClass());

 private ConfigurableApplicationContext server;

 private Traverson traverson;

 @Before
```

```
 public void setUp() throws Exception {

  this.server = new SpringApplicationBuilder()
   .properties(Collections.singletonMap("server.port", "0"))
   .sources(DemoApplication.class).run();
  // this.server =
  // SpringApplication.run(DemoApplication.class);
  this.traverson = this.server.getBean(Traverson.class);
 }

 @After
 public void tearDown() throws Exception {
  if (null != this.server) {
   this.server.close();
  }
 }

 @Test
 public void testTraverson() throws Exception {

  String nameOfMovie = "Cars";

  ❶
  Resources<Actor> actorResources = this.traverson
   .follow("actors", "search", "by-movie")
   .withTemplateParameters(Collections.singletonMap("movie", nameOfMovie))
   .toObject(new ParameterizedTypeReference<Resources<Actor>>() {
   });

  actorResources.forEach(this.log::info);
  assertTrue(actorResources.getContent().size() > 0);
  assertEquals(
   actorResources.getContent().stream()
    .filter(actor -> actor.fullName.equals("Owen Wilson"))
    .collect(Collectors.toList()).size(), 1);
 }
}
```

❶ Метод `Traverson#follow` легко справляется с проходом по цепочке ссылок, заменяя при необходимости соответствующие части шаблона параметров URI. Кроме того, метод принимает параметр `ParameterizedTypeReference<T>` так же, как это делает `RestTemplate`.

Шаблон `RestTemplate` — центральная часть создания REST-сервисов с помощью Spring. В Spring версии 5 имеется клиент с полноценной реакцией `WebClient`, предлагающий превосходную альтернативу `RestTemplate` для конкретных классов взаимодействия, в частности тех, что связаны с долговременной обработкой и с операциями ввода-вывода. В числе прочих интеграционных особенностей шаблон поддерживает токены `OAuth` в Spring Cloud Security и выделение сервиса в Spring Cloud.

Резюме

В этой главе мы рассмотрели способы эффективного создания REST API с помощью Spring Boot. REST — один из подходов к созданию масштабируемых сервисов в открытой, мультиплатформенной, многоязычной сети Интернет. Кроме того, REST хорошо сочетается с решением других вопросов, таких как обеспечение безопасности, избыточности и кэширование данных. Наш разговор затрагивал принципы REST в понятиях Spring, но они применимы не только к этой среде, но и к любому языку и платформе.

7 Маршрутизация

Системы, предназначенные для работы в облаке, динамичны; сервисы подключаются и отключаются по мере необходимости. Сервисы должны иметь возможность обнаруживать другие сервисы и связи с ними, даже если один экземпляр сервиса исчезает или к системе добавляются какие-то новые свойства. Не стоит полагаться на фиксированные IP-адреса, связывающие клиентов с сервисами. Чтобы обрести желаемую гибкость динамического перенаправления, нужна косвенная адресация.

Можно было бы задействовать систему DNS, но она не всегда подходит облачной среде. В DNS применяется кэширование на стороне клиентов, пользующихся DNS-сервисами. Кэширование означает, что клиенты при разрешении имен могут привлекать к работе устаревшие и уже не существующие IP-адреса. Быстрого аннулирования подобных данных в кэше можно добиться за счет мизерных значений времени жизни информации (time-to-live, TTL), но тогда придется тратить много времени на новое разрешение DNS-записей. В облаке для DNS требуется дополнительное время на разрешение, поскольку запросы должны покинуть эту среду, а затем снова войти через маршрутизатор. Многие поставщики облачных услуг поддерживают многосетевое DNS-разрешение с закрытым и открытым адресами. В таком случае вызовы из одного сервиса к другому внутри облака выполняются быстро и эффективно и практически без каких-либо затрат (с точки зрения поставщика, в частности, имеется в виду пропускная способность канала), но для этого требуется, чтобы кодовая база была в курсе сложностей, имеющихся в схеме DNS. А это означает сложность реализации, которую также может быть весьма трудно воспроизвести в локальной среде разработчика.

В DNS также используется слишком простой протокол: они не могут отвечать на основные вопросы о состоянии или о топологии системы. Предположим, имеется REST-клиент, вызывающий сервис, запущенный на узле, отображенном на DNS. Если экземпляр выйдет из строя, то клиент будет заблокирован. Остается надеяться, что все мы прониклись предложениями, изложенными в замечательной книге Майкла Нигарда (Michael Nygard) *Release It!* (Pragmatic Programmers), и применением указанных действенных сроков ожидания на стороне клиента.

Большинство этим не занимается. (Проверьте! Каково исходное время ожидания Java-подключения `java.net.URLConnection`? А вы *уверены*, что установили соответствующие сроки ожидания на стороне клиента во всем своем коде? *Неужели?*)

Актуальность маршрутизации возрастает, когда дело доходит до балансировки нагрузки DNS. Это еще одна часть операционной инфраструктуры, подлежащая управлению (или, что бывает чаще, управляемая какой-нибудь другой командой), у которой имеются ограничения. Обычный балансировщик нагрузки будет, к примеру, вполне неплохо справляться с балансировкой нагрузки по принципу циклического перебора, а большинство балансировщиков смогут даже справиться с «липкими сессиями», когда запросы от клиента привязываются к определенному узлу на основе общих заголовков (подобных имеющемуся в Java `JSESSIONID`). Но, когда нужно проявить активность более высокого порядка, они пасуют. Что, если понадобится привязать запросы от клиента к конкретному узлу на основе чего-либо кроме `JSESSIONID`, например к токену OAuth Access или к JSON Web Token (JWT)? Вероятно, вы делаете что-либо с сохранением состояния, скажем, используете потоковое видео, для чего фактически невозможно сбалансировать нагрузку, но при этом не имеете понятия о HTTP-сессии.

Все узлы в пуле балансировки нагрузки DNS должны также, по-видимому, быть маршрутизируемыми. Но порой маршрутизируемость серверов через DNS абсолютно не нужна! Конечно же, безопасность, достигаемая через неизвестность, это вообще не безопасность, но всегда следует стремиться ко всевозможной минимизации открыто выставляемых пространств. Более того, даже если все в порядке с клиентами, подключенными непосредственно к узлу, необходимо убедиться, что пресыщенные узлы выведены из группы балансировки нагрузки должным образом. В противном случае клиент будет перенаправлен на узел, который не сможет обработать запрос. Не у всех балансировщиков нагрузки хватает интеллекта на подобные действия. Некоторые из них смогут запросить экземпляр сервиса и осведомиться о его работоспособности и на основании кода состояния, переданного в ответе, исключить узел из пула. (Для этого можно воспользоваться имеющейся в Spring Boot конечной точкой актуатора `/health`.) У некоторых платформ (подобных JVM) имеются библиотеки, выполняющие кэширование первых IP-адресов, прошедших разрешение из DNS, и повторно применяющие их при последующих подключениях. В целом это вполне рациональная идея, но она сможет расстроить балансировку нагрузки при использовании DNS.

Некоторые из этих ограничений можно обойти, воспользовавшись виртуальным балансировщиком нагрузки, работающим наподобие некоего посредника, но такие инструменты все же не в состоянии решить самую серьезную проблему. Все централизованные балансировщики нагрузки не знают состояния вашей системы и степени ее загруженности. Даже самый идеальный балансировщик нагрузки DNS, функционирующий по принципу циклического перебора, станет распределять исходные подключения по различным сервисам, но это не будет безусловно означать

распределение самой рабочей нагрузки. Не все запросы создаются равными по объему. Одни клиенты потребляют больше ресурсов, чем другие, и на взаимодействие с ними может уходить больше времени. Это способно привести к перегрузке отдельных узлов, притом что в условиях отсутствия разумной стратегии балансировки нагрузки другие узлы будут находиться в простое.

Короче говоря, маршрутизация зачастую относится к вопросам программирования, и DNS в эту тему вписывается слабо.

Абстракция DiscoveryClient

Существуют альтернативные способы получения эффекта от централизованной балансировки нагрузки. Нужно логически отобразить идентификатор сервиса на хосты и порты (узлы), на которых доступен этот сервис. Сюда хорошо вписывается реестр сервисов. Основным недостатком таких реестров считается их вторжение в код вашего приложения. Ваш код должен знать реестр и работать с ним. Для клиентов, которые не в состоянии приспособиться, отдельные реестры сервисов (например, Hashicorp Consul) могут работать с DNS-сервером. По сути, реестр сервисов похож на телефонную книгу для облака. Это таблица экземпляров сервисов, способная предоставить API. У одних регистров сервисов интеллекта больше, чем у других, но все они обладают минимальной поддержкой обнаружения доступных сервисов и тех мест, где находятся экземпляры таких сервисов.

Не все реестры сервисов создаются одинаковыми. Они также подвержены ограничениям вследствие физики и положений CAP-теоремы, в которой утверждается, что для распределенных систем невозможно предоставить более двух из трех следующих свойств: согласованности, доступности и устойчивости к разделению. Их задача сводится к поддержке последовательного видения системы. Для каждого потенциального реестра сервисов нужно дать оценку тому, как он соотносится с треугольником CAP-теоремы. Кроме того, необходимо оценить функции, имеющиеся в конкретном реестре, помимо его основной обязанности. Поддерживает ли он безопасность и кодирование? Может ли использоваться в качестве хранилища в формате «ключ — значение» для централизованной конфигурации или нужно применять сервер Spring Cloud Config, рассмотренный ранее при изучении вопросов конфигурирования в главе 3?

Чтобы облегчить клиентам работу с различными типами реестров сервисов, платформа Spring Cloud предоставляет абстракцию `DiscoveryClient`. Последняя упрощает подключение различных реализаций. Spring Cloud вставляет ее в различные части платформы, делая ее использование практически очевидным. На момент написания этих строк уже существовали такие достойные практического применения реализации `DiscoveryClient` для Cloud Foundry, как Apache Zookeeper, HashicorpConsul и Eureka от компании Netflix, с как минимум еще одной реализацией, не имеющей практической ценности, доступной для ETCD. Абстракция доста-

точно проста для того, чтобы вы при желании могли адаптировать Spring Cloud для работы с другим реестром сервисов. Концептуально абстракция `DiscoveryClient` предназначена только для чтения, как показано в примере 7.1.

Пример 7.1. Абстракция DiscoveryClient

```
package org.springframework.cloud.client.discovery;

import java.util.List;
import org.springframework.cloud.client.ServiceInstance;

public interface DiscoveryClient {
  String description();

  ServiceInstance getLocalServiceInstance();

  List<ServiceInstance> getInstances(String var1);

  List<String> getServices();
}
```

Некоторым реестрам сервисов требуется, чтобы клиент зарегистрировался в реестре. Различные реализации абстракции `DiscoveryClient` делают это за вас при запуске приложения. Данная абстракция, имеющаяся в Cloud Foundry, не требует, чтобы сам клиент регистрировался в реестре, поскольку среде уже известно, на каком хосте и порте находится сервис, он поставил там сервис на первое место!

Существует множество прекрасных вариантов реестров сервисов. Поскольку Spring предоставляет абстракцию, наш выбор не ограничен всего лишь одним вариантом. Впоследствии мы легко можем переключиться на другую реализацию. В этой главе мы рассмотрим реестр Netflix Eureka. Он пользовался полномасштабной поддержкой компании Netflix на протяжении многих лет. Кроме того, для разработки приложения его достаточно просто установить и запустить, применяя лишь Spring Boot, что вдвойне увеличивает его привлекательность!

Несмотря на то что Netflix Eureka можно настроить практически на любые действия, для этого придется приложить некоторые усилия. Очень важно располагать проверенными рецептами достижения нужного уровня безопасности, масштабируемости и избыточности, подобные получаемым при использовании Netflix-установки на основе сервисов Spring Cloud на Pivotal Cloud Foundry или Pivotal Web Services.

Чтобы настроить реестр сервисов Eureka, в проекте Spring Boot нужно воспользоваться записью `org.springframework.cloud : spring-cloud-starter-eureka-server`, после чего, как показано в примере 7.2, следует инициализировать сервис с помощью `@EnableEurekaServer`.

Пример 7.2. Запуск реестра сервисов Eureka в минимальной конфигурации

```
package demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

❶
@EnableEurekaServer
@SpringBootApplication
public class EurekaServiceApplication {

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

❶ Аннотация `@EnableEurekaServer` создает экземпляр реестра сервисов Eureka в минимальной конфигурации.

Эта конфигурация не годится для ввода в производство! В данной книге во имя достижения работоспособности и демонстрации сути различных понятий удобнее просто отмахнуться от многих вещей. Но в условиях реальной эксплуатации на них следует обратить внимание. Netflix Eureka может справиться с нужными вариантами применения, но это совсем не означает, что такие вещи, как кластеризация (https://github.com/spring-cloud/spring-cloud-netflix/issues/203), даются легко! Хотелось бы надеяться, что вы используете платформу (наподобие Cloud Foundry), способную автоматизировать функциональные аспекты запуска чего-либо подобного для вас.

Реестр сервисов требуется настроить. По устоявшемуся соглашению сервис запускается на порте 8761, и не нужно, чтобы реестр пытался зарегистрировать себя с использованием других узлов (пример 7.3).

Пример 7.3. Настройка простого реестра сервисов Eureka

```
server.port=${PORT:8761}
❶
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
❷
eureka.server.enable-self-preservation=false
```

❶ Не нужно, чтобы Eureka пытался зарегистрировать самого себя.

❷ Если существенная часть зарегистрированных экземпляров не станет проявлять признаков жизни в течение какого-то периода времени, то Eureka посчитает, что такое поведение связано с сетью, и не отменит регистрацию сервисов, не отвечающих на запросы. В этом заключается имеющийся в Eureka режим самосохранения. Возможно, оставить его включенным, указав значение `true`, вполне

разумно, но это также будет означать следующее: при наличии небольшого набора экземпляров (что и произойдет при попытке пробных запусков примеров с небольшим количеством узлов на локальной машине) вы не увидите ожидаемого поведения при отмене регистрации экземпляров.

Посмотрите на новый и весьма привлекательный реестр сервисов Eureka на рис. 7.1! Если провести указателем мыши по изображению листочка Spring, то данное изображение оживет. Кому-то это нравится.

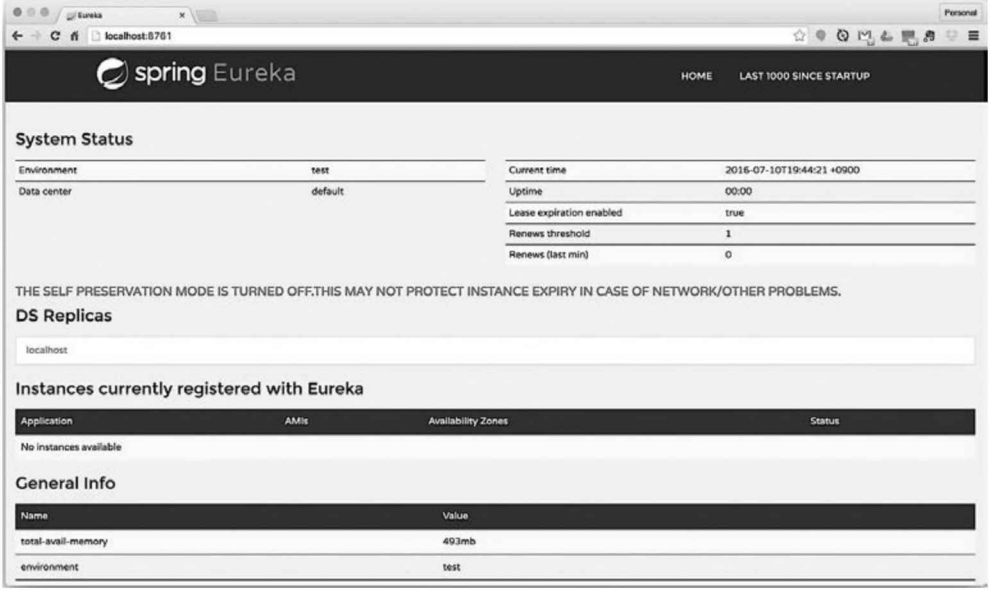


Рис. 7.1. Исходный реестр сервисов Eureka, имеющий минимальную конфигурацию

Вот теперь наш реестр стал доступен. Но полной его доступность не назовешь, поскольку для реальной эксплуатации он вряд ли пригодится. Дело в том, что временами Eureka станет «ругаться» и выводить на консоль окрашенные в кроваво-красные тона сообщения об ошибках. В них будет в основном говориться о необходимости настройки узлов репликации; в противном случае появится риск создания слишком ненадежного развертывания. Не забывайте, что Eureka служит своеобразной нервной системой для наших REST-сервисов: с помощью данного реестра будут общаться друг с другом все остальные сервисы.

Но доступ все же есть. Предоставим клиента и сервис и заставим их обоих зарегистрироваться с помощью реестра. Сначала создадим обычный REST API, чтобы было с чем работать. В этом приложении используется установка `org.springframework.cloud : spring-cloud-starter-eureka` (предоставляющая реализацию Spring Cloud `DiscoveryClient` для Netflix Eureka) и установка `org.springframework.boot : spring-boot-starter-web` (пример 7.4).

Пример 7.4. Простой REST API, возвращающий приветствие

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.util.StringUtils;
import org.springframework.web.bind.annotation.*;
import javax.servlet.http.HttpServletRequest;
import java.util.Collections;
import java.util.Map;
import java.util.Optional;

❶
@EnableDiscoveryClient
@SpringBootApplication
public class GreetingsServiceApplication {

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

@RestController
class GreetingsRestController {

 ❷
 @RequestMapping(method = RequestMethod.GET, value = "/hi/{name}")
 Map<String, String> hi(
  @PathVariable String name,
  @RequestHeader(value = "X-CNJ-Name", required = false)
          Optional<String> cn) {
  String resolvedName = cn.orElse(name);
  return Collections.singletonMap("greeting", "Hello, " + resolvedName + "!");
 }
}
```

❶ Аннотация `@EnableDiscoveryClient` активирует абстракцию Spring Cloud `DiscoveryClient`.

❷ Ответом станет простая JSON-структура с атрибутом `greeting`.

Нашему сервису понадобится лишь выполнить собственную идентификацию и настроить способ взаимодействия с Eureka. Среда Spring Cloud учтет значения, найденные в `application.{yml,properties}`, точно так же, как это делает Spring Boot, но ей нужны некоторые значения *на более ранней стадии* жизненного цикла приложения. Как показано в примере 7.5, она ожидает возможности совершить поиск в `bootstrap.properties` таких значений, как адрес сервера Spring Cloud Config.

Пример 7.5. bootstrap.properties

```
spring.application.name=greetings-service  ❶

server.port=${PORT:0}  ❷

eureka.instance.instance-id=\  ❸
  ${spring.application.name}:${spring.application.instance_id:${random.value}}
```

❶ Приложение будет зарегистрировано как `greetings-service`.

❷ И оно будет запущено на произвольном порте.

❸ Какой уникальный идентификатор нужен для каждого зарегистрированного сервиса? С учетом используемого узла Spring Cloud предоставит подходящий исходный вариант. Мы переопределили исходный зарегистрированный идентификатор, чтобы поддержать уникальность идентификаторов.

Запустите несколько экземпляров `greetings-service`, а затем обновите информацию в реестре сервисов Eureka, после чего можно будет увидеть только что зарегистрированные экземпляры. Теперь они рекламируют свое присутствие и доступны для использования другими сервисами (рис. 7.2).

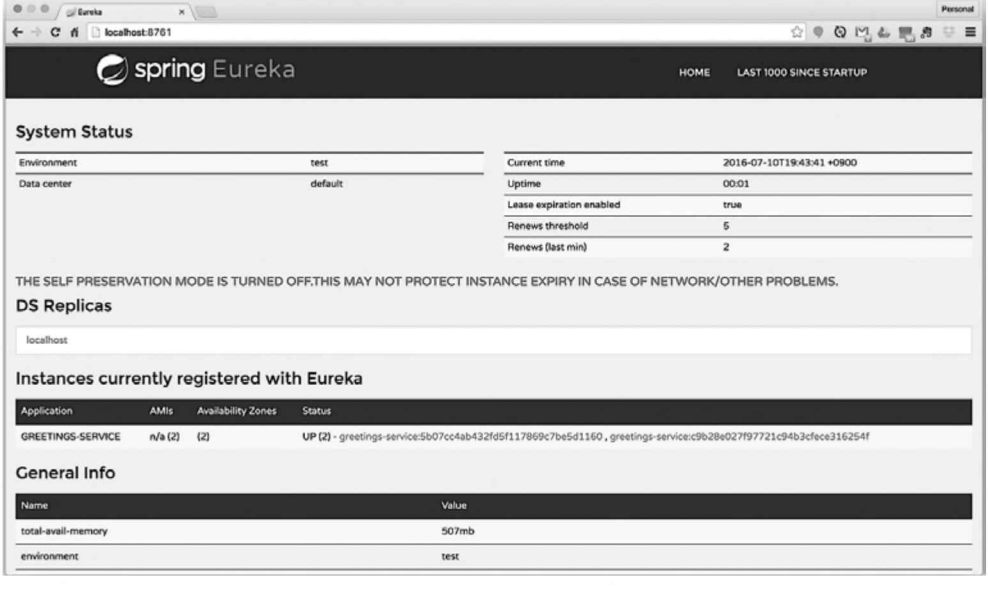


Рис. 7.2. Эврика! Теперь у нас появились зарегистрированные экземпляры!

Установим простой клиент в виде *пограничного сервиса* (edge service). Более подробно пограничные сервисы будут рассматриваться в главе 8, а пока достаточно сказать, что пограничный сервис представлен первым портом вызова для запросов, поступающих в систему. Пограничный сервис станет действовать в качестве

клиента для `greetings-service`, который тот будет разрешать. Он начнет взаимодействовать с `greetings-service` после разрешения сервиса в реестре сервисов.

Кроме того, наш новый клиент применяет `spring-cloud-starter-eureka` и у него имеется аннотация `@EnableDiscoveryClient`. Если нужно провести опрос зарегистрированных экземпляров сервиса, то абстракцию `DiscoveryClient` можно использовать напрямую (пример 7.6).

Пример 7.6. DiscoveryClientCLR.java

```
package com.example;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.stereotype.Component;

@Component
public class DiscoveryClientCLR implements CommandLineRunner {

 private final DiscoveryClient discoveryClient;

 private Log log = LogFactory.getLog(getClass());

 ❶
 @Autowired
 public DiscoveryClientCLR(DiscoveryClient discoveryClient) {
  this.discoveryClient = discoveryClient;
 }

 @Override
 public void run(String... args) throws Exception {

  ❷
  this.log.info("localServiceInstance");
  this.logServiceInstance(this.discoveryClient.getLocalServiceInstance());

  ❸
  String serviceId = "greetings-service";
  this.log.info(String.format("registered instances of '%s'", serviceId));
  this.discoveryClient.getInstances(serviceId)
   .forEach(this::logServiceInstance);
 }

 private void logServiceInstance(ServiceInstance si) {
  String msg = String.format("host = %s, port = %s, service ID = %s",
   si.getHost(), si.getPort(), si.getServiceId());
  log.info(msg);
 }
}
```

❶ Внедрение сконфигурированной реализации Spring Cloud `DiscoveryClient`.

❷ Получение информации о текущем экземпляре, то есть ответ на вопрос о том, какая информация будет зарегистрирована в Eureka для текущего запущенного приложения.

❸ Нахождение и перечисление всех зарегистрированных экземпляров `greetings-service`.

`DiscoveryClient` упрощает взаимодействие с реестром и перечисление всех зарегистрированных экземпляров. При наличии более одного экземпляра зарегистрированного сервиса приходится выбирать конкретную версию. Можно произвольно выбрать тот из них, который способен эффективно сбалансировать нагрузку методом циклического перебора. Возможно, потребуется сделать что-то еще, что может воспользоваться осведомленностью о приложении и экземплярах, например выполнить балансировку нагрузки с учетом производительности и мощности отвечающей системы. Какой бы ни была стратегия, нужно дать ее описание в коде только один раз, а затем повторно задействовать данное описание везде, где выполняется вызов сервиса. В этом и состоит суть балансировки нагрузки на стороне клиента.

Netflix Ribbon — библиотека балансировки нагрузки на стороне клиента. В ней поддерживаются различные стратегии, включая циклический перебор и балансировку нагрузки на основе распределения по возможностям предоставления более объемных ответов. Но ее истинная эффективность заключается в возможности расширения. Рассмотрим пример 7.7, где показано ее применение на низком уровне.

Пример 7.7. RibbonCLR.java

```
package com.example;

import com.netflix.loadbalancer.*;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.stereotype.Component;

import java.net.URI;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

@Component
public class RibbonCLR implements CommandLineRunner {

 private final DiscoveryClient discoveryClient;

 private final Log log = LogFactory.getLog(getClass());

 @Autowired
```

```
 public RibbonCLR(DiscoveryClient discoveryClient) {
  this.discoveryClient = discoveryClient;
 }

 @Override
 public void run(String... args) throws Exception {

  String serviceId = "greetings-service";

  List<Server> servers = this.discoveryClient.getInstances(serviceId).stream()
   .map(si -> new Server(si.getHost(), si.getPort()))
   .collect(Collectors.toList());

  IRule roundRobinRule = new RoundRobinRule();

  BaseLoadBalancer loadBalancer = LoadBalancerBuilder.newBuilder()
   .withRule(roundRobinRule).buildFixedServerListLoadBalancer(servers);

  IntStream.range(0, 10).forEach(i -> {

   Server server = loadBalancer.chooseServer();
   URI uri = URI.create("http://" + server.getHost() + ":" + server.getPort()
    + "/");
   log.info("resolved service " + uri.toString());
  });
 }
}
```

При всей своей работоспособности этот пример все же навевает скуку! К счастью, Spring Cloud автоматически интегрирует балансировку нагрузки на стороне клиента на различных уровнях данной среды. В частности, весьма полезным образцом может послужить автоматическое конфигурирование Spring-шаблона `RestTemplate`. В нем поддерживаются перехватчики до (pre-) и после (post-) обработки совершенных HTTP-запросов. Перехватчик Spring Cloud `@LoadBalanced` можно применить с целью дать Spring Cloud сигнал о том, что требуется отконфигурировать в `RestTemplate` перехватчик балансировки нагрузки, знающий об использовании среды Ribbon (пример 7.8).

Пример 7.8. LoadBalancedRestTemplateConfiguration.java

```
package com.example;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class LoadBalancedRestTemplateConfiguration {

 ❶
 @Bean
```

```
 @LoadBalanced
 RestTemplate restTemplate() {
  return new RestTemplate();
 }
}
```

❶ Аннотация `@LoadBalanced` является определителем. В Spring она используется для устранения неоднозначности в определениях bean-компонентов и установки признаков конкретных bean-компонентов для обработки. Здесь признак конкретно этого экземпляра `RestTemplate` устанавливается нужным образом для того, чтобы был отконфигурирован перехватчик балансировки.

Вот и все, наша работа существенно упростилась! Балансировщик нагрузки извлечет URI из любого HTTP-запроса и посчитает хост за идентификатор сервиса, подлежащий разрешению с помощью `DiscoveryClient` (в данном случае благодаря Netflix Eureka) и подвергаемый балансировке нагрузки с применением Netflix Ribbon (пример 7.9).

Пример 7.9. LoadBalancedRestTemplateCLR.java

```
package com.example;

import com.fasterxml.jackson.databind.JsonNode;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import java.util.Collections;
import java.util.Map;

@Component
public class LoadBalancedRestTemplateCLR implements CommandLineRunner {

 private final RestTemplate restTemplate;

 private final Log log = LogFactory.getLog(getClass());

 ❶
 @Autowired
 public LoadBalancedRestTemplateCLR(@LoadBalanced RestTemplate restTemplate) {
  this.restTemplate = restTemplate;
 }

 @Override
 public void run(String... strings) throws Exception {

  Map<String, String> variables = Collections.singletonMap("name",
```

```
    "Cloud Natives!");

  ❷
  ResponseEntity<JsonNode> response = this.restTemplate.getForEntity(
    "//greetings-service/hi/{name}", JsonNode.class, variables);
  JsonNode body = response.getBody();
  String greeting = body.get("greeting").asText();
  log.info("greeting: " + greeting);
 }
}
```

❶ Одну и ту же спецификационную аннотацию `@LoadBalanced` используют как производитель bean-компонента, так и место инъекции.

❷ Мы задействуем `RestTemplate`, применив альтернативную возможность, указывая для хоста вместо DNS-адреса идентификатор сервиса (`greetings-service`).

Сервисы маршрутизации Cloud Foundry

Балансировка нагрузки на стороне клиента — серьезная составляющая в принятии решений по маршрутизации; она облегчает труд разработчиков, задавая в коде логику маршрутизации. Последняя носит локальный характер по отношению к системе сервисов, и при необходимости внести в нее изменения можно сделать это, не беспокоясь о потенциальном негативном влиянии на какого-либо потребителя сервиса. И тем не менее не стоит ожидать возможности потребовать от всех сторонних клиентов, чтобы они применяли Netflix Ribbon и пользовались вашим API точно таким же способом. Справедливость этого положения особенно проявляется при установке пограничных сервисов, реагирующих на запросы с клиентской стороны, то есть на запросы от любого количества различных устройств и пользовательских интерфейсов. К примеру, iOS-клиентам для выполнения своей работы не свойственно применение Apache Zookeeper и Netflix Ribbon. В данном случае было бы полезно установить промежуточный сервис, занимающийся обработкой запроса, управляющий маршрутизацией, а затем переправляющий запросы нисходящим сервисам. Такой компонент также имеет преимущества вследствие возможности применять его к сервисам всех типов, независимо от языка их реализации.

Cloud Foundry поддерживает компонент особого типа — *сервис маршрутизации*, предоставляющий основную долю всех преимуществ балансировки нагрузки на стороне клиента и централизованный компонент, обеспечивающий поведение маршрутизации. Сервис маршрутизации является в Cloud Foundry еще одной плоскостью расширения, которую можно разработать и подключить к вашей платформе наряду со сборочными пакетами, приложениями и сервис-брокерами. По сути, сервис маршрутизации — HTTP-сервис, принимающий все HTTP-запросы для заданного хоста и домена, а затем делающий с ними все, что ему нужно, включая их обработку, преобразование или пересылку.

Сервисы маршрутизации предназначены для выполнения роли универсальных посредников, которые вставляются в цепочку запросов для (теоретически) любого типа применения, поэтому на них накладываются отдельные ограничения. Они должны поддерживать не только протокол HTTP, но и протокол HTTPS. На данный момент Cloud Foundry не поддерживает выстраивание в цепочку сервисов маршрутизации от запроса к запросу. В этой среде сервисы маршрутизации рассматриваются как еще один тип сервиса, предоставляемого пользователем. Они немного отличаются от других рассмотренных в данной книге типов сервисов, предоставляемых пользователем, поскольку связь сервиса маршрутизации и приложения определяется в понятиях имени хоста и домена приложений, а не в понятиях имени приложения.

Рассмотрим пример. Поскольку нужно принять и просто переслать дальше все поступающие HTTP- или HTTPS-запросы, то потребуется настройка Spring-шаблона `RestTemplate` на доверие всему получаемому и на игнорирование ошибок (пример 7.10).

Пример 7.10. RouteServiceApplication.java

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.http.client.ClientHttpResponse;
import org.springframework.http.client.SimpleClientHttpRequestFactory;
import org.springframework.web.client.DefaultResponseErrorHandler;
import org.springframework.web.client.RestTemplate;

import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManager;
import javax.net.ssl.X509TrustManager;
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.Proxy;
import java.net.URL;
import java.security.KeyManagementException;
import java.security.NoSuchAlgorithmException;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;

/**
 * @author Ben Hale
 */

@SpringBootApplication
public class RouteServiceApplication {

 public static void main(String[] args) {
```

```
 SpringApplication.run(RouteServiceApplication.class, args);
}

❶
@Bean
RestTemplate restOperations() {
 RestTemplate restTemplate = new RestTemplate(
  new TrustEverythingClientHttpRequestFactory()); ❷
  restTemplate.setErrorHandler(new NoErrorsResponseErrorHandler()); ❸
  return restTemplate;
}

private static class NoErrorsResponseErrorHandler extends
 DefaultResponseErrorHandler {

 @Override
 public boolean hasError(ClientHttpResponse response) throws IOException {
  return false;
 }
}

private static final class TrustEverythingClientHttpRequestFactory extends
 SimpleClientHttpRequestFactory {

 private static SSLContext getSslContext(TrustManager trustManager) {
  try {
   SSLContext sslContext = SSLContext.getInstance("SSL");
   sslContext.init(null, new TrustManager[] { trustManager }, null);
   return sslContext;
  }
  catch (KeyManagementException | NoSuchAlgorithmException e) {
   throw new RuntimeException(e);
  }
 }

 @Override
 protected HttpURLConnection openConnection(URL url, Proxy proxy)
  throws IOException {
  HttpURLConnection connection = super.openConnection(url, proxy);
  if (connection instanceof HttpsURLConnection) {
   HttpsURLConnection httpsConnection = (HttpsURLConnection) connection;
   SSLContext sslContext = getSslContext(new TrustEverythingTrustManager());
   httpsConnection.setSSLSocketFactory(sslContext.getSocketFactory());
   httpsConnection.setHostnameVerifier((s, session) -> true);
  }
  return connection;
 }
}

private static final class TrustEverythingTrustManager implements
```

```
  X509TrustManager {

  @Override
  public void checkClientTrusted(X509Certificate[] x509Certificates, String s)
   throws CertificateException {
  }

  @Override
  public void checkServerTrusted(X509Certificate[] x509Certificates, String s)
   throws CertificateException {
  }

  @Override
  public X509Certificate[] getAcceptedIssuers() {
   return new X509Certificate[0];
  }
 }
}
```

❶ Конфигурирование `RestTemplate`...

❷ ...со всеобщим доверием...

❸ ...и игнорированием ошибок.

Этот сервис маршрутизации регистрирует все поступающие запросы до и после их принятия (пример 7.11).

Пример 7.11. Controller.java

```
package com.example;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpHeaders;
import org.springframework.http.RequestEntity;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestOperations;
import org.springframework.web.client.RestTemplate;

import java.net.URI;
import java.util.Arrays;
import java.util.Collections;

/**
 * @author Ben Hale
 */
@RestController
```

```
class Controller {

 ❶
 private static final String FORWARDED_URL = "X-CF-Forwarded-Url";

 private static final String PROXY_METADATA = "X-CF-Proxy-Metadata";

 private static final String PROXY_SIGNATURE = "X-CF-Proxy-Signature";

 private final Log logger = LogFactory.getLog(this.getClass());

 private final RestOperations restOperations;

 @Autowired
 Controller(RestTemplate restOperations) {
  this.restOperations = restOperations;
 }

 ❷
 @RequestMapping(headers = { FORWARDED_URL, PROXY_METADATA, PROXY_SIGNATURE })
 ResponseEntity<?> service(RequestEntity<byte[]> incoming) {

  this.logger.info("incoming request: " + incoming);

  HttpHeaders headers = new HttpHeaders();
  headers.putAll(incoming.getHeaders());
  headers.put("X-CNJ-Name", Collections.singletonList("Cloud Natives"));

  ❸
  URI uri = headers
   .remove(FORWARDED_URL)
   .stream()
   .findFirst()
   .map(URI::create)
   .orElseThrow(
    () -> new IllegalStateException(String.format("No %s header present",
     FORWARDED_URL)));

  ❹
  RequestEntity<?> outgoing = new RequestEntity<>(
   ((RequestEntity<?>) incoming).getBody(), headers, incoming.getMethod(), uri);

  this.logger.info("outgoing request: {}" + outgoing);

  return this.restOperations.exchange(outgoing, byte[].class);
 }
}
```

❶ Cloud Foundry проходит по специализированным заголовкам, предоставляющим контекст для поступающих запросов.

❷ Факт присутствия этих заголовков используется в качестве селектора определения того, должен ли сервис маршрутизации обрабатывать запрос.

❸ В методе обработки запроса мы определяем конечный запрос, извлекая соответствующий заголовок...

❹ ...и адаптируем поступающий запрос в исходящий, выполняемый с помощью `RestTemplate`.

Это простой пример, но имеющиеся возможности безграничны. Можно организовать предварительную обработку запросов и провести аутентификацию или адаптировать один тип аутентификации для другого. Можно вставить логику для ограничения скорости обработки или же перенаправить запросы определенным образом, полагаясь на балансировку нагрузки на стороне клиента. Можно было бы вести учет запросов, поступающих в систему. Можно просто воспользоваться предоставляемыми возможностями, чтобы создать некий общий промежуточный уровень обслуживания, и обогатить тело запроса данными о глобальном состоянии или контекстом запроса (например, информацией о последовательности выполненных пользователем щелчков кнопкой мыши).

Здесь может появиться желание проделать множество операций, связанных с учетом, ограничением скорости обработки, аутентификации и т. д., то есть решить множество общих вопросов, для которых не обязательно создавать код. Со многими этими функциональными возможностями хорошо справляются такие программные продукты, основанные на применении API-шлюзов, как Apigee, помимо прочего распространяющие сервис маршрутизации Cloud Foundry, настраиваемый под ваше приложение. Apigee API-шлюз связан с вашей учетной записью Apigee, и вся централизованно настраиваемая вами политика автоматически применяется к вашим запросам.

Использование сервиса маршрутизации не вызывает особых затруднений, чего нельзя сказать о применении сервисов маршрутизации других типов. Сервис маршрутизации, по сути, просто развернутое вами приложение Cloud Foundry. Предположим, вы уже развернули в Cloud Foundry два приложения: `route-service` (только что рассмотренный нами регистрирующий сервис маршрутизации) и обычное приложение Spring Boot и Spring MVC, `downstream-service`. В примере 7.12 показано подключение приложения (`downstream-service`) к сервису маршрутизации (`route-service`) для приложений, развернутых на Pivotal Web Services, где исходным доменом является `cfapps.io`. (При использовании экземпляра Cloud Foundry, развернутого в другом месте, ситуация будет иной.)

Пример 7.12. Подключение приложения к сервису маршрутизации

```
cf create-user-provided-service route-service -r
   https://my-route-service.cfapps.io
❶

cf bind-route-service cfapps.io route-service -hostname my-downstream-service ❷
```

❶ Создание предоставляемого пользователем сервиса маршрутизации с указанием в нем URL для нашего развернутого `route-service`.

❷ Привязка этого сервиса маршрутизации к любому приложению, чьим маршрутом является `http://my-downstream-service.cfapps.io`.

Теперь несколько раз зайдите на `downstream-service` в браузере, а затем получите последние регистрационные записи для сервиса маршрутизации (`cf logs --recent route-service`), чтобы посмотреть, как запросы отображаются в выводе регистрационной информации.

Резюме

Для сервисов, развертывание которых производится с помощью протокола HTTP (хотя ничто не говорит об обязательности такого развертывания!), маршрутизация является важнейшей задачей. Ее значимость возрастает еще больше, когда жизненный цикл сервиса носит динамический характер и изменяются топологии сервиса. Однотипные клиенты, обращающиеся к сервисам из-за пределов нашей системы, должны иметь возможность спрогнозировать доступность сервисов в фиксированном, широко известном месте, но притом нужно сохранять полную гибкость для обмена данными между сервисами. Решить эти, казалось бы, противоречивые задачи можно при некоторой доле смекалки и с помощью ряда приемов, рассмотренных в данной главе.

8 Пограничные сервисы

Микросервисы не существуют в вакууме. В конечном итоге они обслуживают клиентов. Таковых мириады: это HTML5-, Android- и iOS-клиенты, клиенты PlayStation или XBox, клиенты Smart TV и, конечно же, почти *все*, у кого на сегодняшний день имеется MAC-адрес (и благодаря волшебству ARP — IP-адрес) и кто может действовать в качестве клиента вашего сервиса. *Все*. Дороги в Сингапуре собирают показания датчиков, снабжая ими искусственный интеллект, находящийся в облаке, который помогает регулировать дорожное движение. По планете разгуливают люди с органами, подключенными к сети!

Клиенты располагают разнообразными возможностями, позволяющими информировать о типах информационного наполнения и сервисах, с которыми они могут работать или обмениваться данными:

- часть из них имеют ограниченный объем памяти или вычислительной мощности, что влияет на объем содержимого, с которым клиент может справиться;
- некоторым нужны особые типы содержимого или кодировок;
- какие-то требуют различных моделей документов, оптимизированных под разных клиентов, лишь часть из которых имеет иерархию;
- у отдельных клиентов полезная площадь экрана может потребовать поэтапной, а не одномоментной загрузки данных;
- для одних клиентов более эффективной будет потоковая, а для других — порционная доставка документов;
- операции взаимодействия с пользователем могут изменить требуемые ответы;
- клиенты могут влиять на поля метаданных, методы доставки, модель взаимодействия и т. д.

Затраты на подгонку под каждого нового клиента каждого микросервиса в системе с необычно большим количеством микросервисов непомерно высоки и очень быстро приводят к значительному усложнению системы. Нашей целью станет сохранение адаптивности, позволяющей самим определять дальнейший путь. В качестве альтернативы все эти проблемы будут решаться в промежуточном, так называемом *пограничном сервисе*. Таковые послужат своеобразным порталом для поступающих

в приложение запросов. Они занимают весьма выгодное положение для решения особых клиентских проблем, а также вопросов, касающихся сразу нескольких сервисов. В числе подобных можно назвать: обеспечение безопасности (аутентификация и авторизация), ограничение скорости передачи данных и ведение учета, выполнение маршрутизации (многие из этих вопросов рассмотрены в главе 7, где идет речь о маршрутизации, фильтрации, API-преобразовании, API типа «клиент — адаптер», сервисах-посредниках и др.).

Сервис приветствий

В данной главе мы рассмотрим вопросы, связанные с пограничными сервисами, но делать это будем применительно к простому REST-сервису `greetings-service` и к реестру сервисов Eureka. А теперь создадим данные сервисы. Сначала установим быстрореализуемый экземпляр Netflix Eureka в виде модуля `service-registry`. Это типичное приложение Spring Boot, подобное тем, что создаются в Spring Initializr (http://start.spring.io/). К нему мы добавили установку `org.springframework.cloud: spring-cloud-starter-eureka-server` (пример 8.1).

Пример 8.1. Установка простого сервера Netflix Eureka в модуле service-registry

```
package demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

❶
@EnableEurekaServer
@SpringBootApplication
public class EurekaServiceApplication {

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

❶ Аннотация `@EnableEurekaServer` создает конфигурацию самого простого экземпляра реестра сервисов Eureka.

Но реестру все же понадобится некая минимальная конфигурация (пример 8.2).

Пример 8.2. application.properties в модуле service-registry

```
spring.application.name=eureka-service
❶
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
❷
eureka.server.enable-self-preservation=false
```

❶ Запрет сервису Eureka на регистрацию самого себя.

❷ Пресечение попыток Eureka провести агрессивное кэширование кластерной информации при обнаружении отсутствия ответа узлов после определенного порогового интервала времени. Эта настройка создает некоторые удобства в процессе разработки, но после ввода в эксплуатацию ее можно отключить.

Запустите `service-registry`, и реестр сервисов запустится на порте 8761. Затем понадобится простой REST API, с которым можно будет работать, а именно `greetings-service`, отвечающий приветствием. Он ничем не примечателен, за исключением его участия в регистрации сервисов и возможности его обнаружения, то есть его саморекламы, позволяющей другим сервисам его найти (пример 8.3).

Пример 8.3. GreetingsServiceApplication.java в модуле greetings-service

```
package greetings;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

❶
@EnableDiscoveryClient
@SpringBootApplication
public class GreetingsServiceApplication {

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

❶ Участие в регистрации и обнаружении сервисов с помощью Spring Cloud-абстракции `DiscoveryClient`.

В данной главе мы покажем множество вариантов преобразованных REST API, Поэтому для их включения и выключения при тех или иных условиях мы воспользовались Spring-профилями. Исходный вариант сервиса приветствий `greetings-service`, который будет запускаться без какого-либо другого указанного профиля, показан в примере 8.4.

Пример 8.4. DefaultGreetingsRestController.java

```
package greetings;

import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collections;
```

```
import java.util.Map;

@Profile({ "default", "insecure" })
@RestController
@RequestMapping(method = RequestMethod.GET, value = "/greet/{name}")
class DefaultGreetingsRestController {

 @RequestMapping
 Map<String, String> hi(@PathVariable String name) {
  return Collections.singletonMap("greeting", "Hello, " + name + "!");
 }
}
```

К этому примеру мы еще не раз вернемся в текущей главе. Запустите код, и он зарегистрируется в Eureka как `greetings-service`, и данный сервис изначально запустится на порте 8081.

Простой пограничный сервис

Создадим простой клиентский API — *пограничный сервис* — `greetings-client`, который будет работать как переходник между внешним миром и нижестоящими сервисами. Эта конечная точка станет действовать в качестве клиента для нашего нижестоящего сервиса приветствий `greetings-service`, с которым будут общаться другие клиенты (HTML5-, iOS-, Android-клиенты и т. д.). Рассмотрим другие пути создания API-адаптеров, то есть программных средств, возвращающих представление данных или синтез данных от других сервисов в нечто новое, специально приспособленное для клиента.

Входной класс для нашего нового API-адаптера несложен: в нем, как и в `greetings-service`, используется Spring Cloud-абстракция `DiscoveryClient`, чтобы он обнаруживался другими узлами с помощью сервиса регистрации и обнаружения. Для начала потребуется отдельный порт и `spring.application.name`. В примере 8.5 в качестве имени приложения применен `greetings-client`.

Пример 8.5. GreetingsClientApplication.java

```
package greetings;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

❶
@EnableDiscoveryClient
@SpringBootApplication
public class GreetingsClientApplication {

 public static void main(String[] args) {
```

```
  SpringApplication.run(GreetingsClientApplication.class, args);
 }
}
```

❶ Активация регистрации и обнаружения сервиса.

Посмотрим на нашу первую клиентскую конечную точку. В таких точках используется балансировка нагрузки на стороне клиента (благодаря интеграции Spring Cloud с Netflix Ribbon), и мы можем запрашивать следующий сервис с помощью `RestTemplate`. Каждый запрос предварительно обрабатывается перехватчиком, предоставляемым Spring Cloud, благодаря спецификационной аннотации `@LoadBalanced`. Перехватчик извлекает хост из URI и работает с `DiscoveryClient`, чтобы найти все экземпляры сервиса, фигурирующие в реестре. Обнаруженные экземпляры передаются балансировщику нагрузки на стороне клиента Netflix Ribbon, который затем определяет, к какому узлу должен быть направлен запрос.

Ответ от конечного вызова к REST-сервису имеет формат JSON. Здесь мы задействуем принадлежащий фреймворку Spring шаблон `ParameterizedTypeReference` — реализацию шаблона токена супертипа (super type token), который упрощает принудительное превращение получающихся JSON-данных в тип, пригодный к работе. Для определения того, какой тип ожидается от возвращенных результатов, в шаблоне `RestTemplate` используется последний параметр в вызове `exchange` — аргумент типа `Class<T>` или `ParameterizedTypeReference<T>`. Здесь мы указываем, что ответ в формате JSON нужно преобразовать в `Map<String, String>` (пример 8.6).

Паттерн токена супертипа (super type token pattern) забирает общие параметры в литералах класса, чтобы обойти ограничения, накладываемые стиранием типа — явления в языке Java, суть которого заключается в том, что Java в ходе выполнения программы не сохраняет общие параметры для переменных экземпляра. Изначально он был доведен до нас Нилом Гафтером (Neal Gafter), сотрудником команды JDK компании Sun (http://gafter.blogspot.com.by/2006/12/super-type-tokens.html).

Пример 8.6. RestTemplateGreetingsClientApiGateway.java

```
package greetings;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Profile;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.client.RestTemplate;

import java.util.Map;

@Profile({ "default", "insecure" })
```

```
@RestController
@RequestMapping("/api")
class RestTemplateGreetingsClientApiGateway {

 private final RestTemplate restTemplate;

 @Autowired
 RestTemplateGreetingsClientApiGateway(
   @LoadBalanced RestTemplate restTemplate) {    (1)
  this.restTemplate = restTemplate;
 }

 @GetMapping("/resttemplate/{name}")
 Map<String, String> restTemplate(@PathVariable String name) {

  //@formatter:off
  ParameterizedTypeReference<Map<String, String>> type =
      new ParameterizedTypeReference<Map<String, String>>() {};
  //@formatter:on

  ResponseEntity<Map<String, String>> responseEntity = this.restTemplate
   .exchange("http://greetings-service/greet/{name}", HttpMethod.GET, null,
    type, name);
  return responseEntity.getBody();
 }
}
```

(1) Аннотация `@LoadBalanced` — определитель, связывающий потребителя зависимости с конкретным экземпляром `RestTemplate`, который был настроен Spring Cloud с перехватчиком для поддержки балансировки на стороне клиента.

Данный экземпляр повышает удобство и скорость вызова других сервисов. Да, действительно удобно, если это делается время от времени, но при необходимости выполнить сразу множество REST-вызовов трудоемкость стремительно возрастает. Наши пограничные сервисы будут выполнять вызовы ко множеству различных сервисов, поэтому нужно упростить создание клиентов для работы с нашими сервисами.

Netflix Feign

Работу по вызову нижестоящих сервисов нужно упростить. Один из вариантов — написать API-клиент с последующим многократным его применением. Он пригоден для совместного использования с другими командами, которые с его помощью могут повысить производительность, экономя время. Эта стратегия вполне разумна, если вы не против создания в первую очередь кода клиента и взяли курс на поддержку его соответствия API сервиса (более подробно вопросам обеспечения такой согласованности уделяется внимание в главе 15 при рассмотрении

порядка тестирования контрактов, ориентированных на потребителя). Но всегда есть риск, что клиент окажется слишком интеллектуальным, то есть содержащим заложенную в него самого бизнес-логику, которая должна быть в реализации сервиса. Следовательно, мы могли бы создать собственный клиент, но это может превратиться в нелегкую битву, приводящую к появлению дополнительных ошибок и несоответствий. Здесь может возникнуть соблазн применить технологию RPC и отказаться от преимуществ использования REST. Мы станем отговаривать вас от этого, если только у вас не будет конкретного пользовательского сценария, в который REST просто не вписывается. Вместо этого присмотритесь к использованию таких средств, как Netflix Feign.

Netflix Feign представляет собой библиотеку от компании Netflix, которая сводит получение клиентов сервиса к простому определению интерфейса и к ряду соглашений. Она удачно интегрируется с Spring Cloud. Активировать ее можно, добавив `@EnableFeignClients` к классу конфигурации (или к классу, имеющему аннотацию `@SpringBootApplication`) и установку `org.springframework.cloud : spring-cloud-starter-feign` к пути к классам. Далее приводится простой клиент, созданный в виде интерфейса. Следует обеспечить наличие в начале `greetings-client` Spring-профиля `feign` (пример 8.7).

Пример 8.7. GreetingsClient.java

```
package greetings;

import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import java.util.Map;

❶
@FeignClient(serviceId = "greetings-service")
interface GreetingsClient {

 ❷
 @RequestMapping(method = RequestMethod.GET, value = "/greet/{name}")
 Map<String, String> greet(@PathVariable("name") String name); ❸
}
```

❶ Идентификатор сервиса используется в тандеме с абстракцией `DiscoveryClient` и Netflix Ribbon для балансировки нагрузки на стороне клиента.

❷ Здесь с помощью аннотаций отображения запросов Spring MVC указывается, какие конечные точки должны вызываться и каким образом. Конечно же, привычнее это видеть в реализации *сервиса*, а не клиента!

❸ Возвращаемое значение применяется для определения способа маршализации результатов. Задействовать шаблон токена супертипа больше не нужно!

Затем в своем пограничном сервисе можно воспользоваться Feign-клиентом (пример 8.8).

Пример 8.8. FeignGreetingsClientApiGateway.java

```
package greetings;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.*;

import java.util.Map;

❶
@Profile("feign")
@RestController
@RequestMapping("/api")
class FeignGreetingsClientApiGateway {

 private final GreetingsClient greetingsClient;

 @Autowired
 FeignGreetingsClientApiGateway(GreetingsClient greetingsClient) {
  this.greetingsClient = greetingsClient;
 }

 ❷
 @GetMapping("/feign/{name}")
 Map<String, String> feign(@PathVariable String name) {
  return this.greetingsClient.greet(name);
 }
}
```

❶ Пограничный сервис запускается под Spring-профилем `feign`.

❷ Вызов сервиса становится вызовом метода, но такого, который взамен по-прежнему основывается на применении REST. Наш клиент так же прост и доступен, как и все остальные, основанные на этой технологии! Любой сведущий в использовании Feign сможет быстро ее освоить.

API-адаптеры могут быть весьма сложными и специализированными. Они являются вполне естественным местом для любой, характерной для конкретного клиента обработки, для которой невозможно дать общее описание и которая применима ко всем запросам. По сути, вопросы адаптации API под клиентов могут выглядеть весьма странно, сродни проблемам интеграции. Более подробно эту тему мы обсудим при рассмотрении вопросов интеграции в главе 10.

До сих пор мы говорили об использовании REST для вызова данных от других сервисов и работы с ними. Собственно, так мы и сделали, удачно проигнорировав требования относительно надежных вызовов типа «сервис к сервису». Дополнительные подробности мы изложим при изучении вопроса об интеграции в главе 12.

Фильтрация и проксирование с использованием Netflix Zuul

До сих пор основное внимание уделялось интеграции конкретных клиентов с конкретными конечными точками нижестоящих REST-сервисов. Многие решаемые вопросы носят более общий характер и применяются ко всем запросам от клиентов с определенного хоста к сервису или группе нижестоящих сервисов. Чтобы упростить установку фильтров, решающих такие задачи, как ограничение скорости передачи данных, проксирование и др., можно воспользоваться микропрокси-компонентом Netflix Zuul (https://github.com/Netflix/zuul).

На момент написания этих строк существовал проект-инкубатор Spring Cloud под названием Spring Cloud Gateway, предназначенный для предоставления более адаптируемой альтернативы Zuul, который основывался на Spring Framework 5 reactive runtime. Код пока не относится к разряду общедоступного — generally available (GA) или стабильного, так что внимание будет сосредоточено на надежном, задающем данный жанр Netflix-проекте Zuul по двум причинам: из-за его высокой работоспособности и распространенности. Знакомство с ним окажет вам в дальнейшем хорошую услугу при выборе перехода к использованию Spring Cloud Gateway.

Создатель Zuul Майки Коэн (Mikey Cohen) в ходе своего весьма интересного сообщения на SpringOne Platform 2016 (http://www.slideshare.net/MikeyCohen1/zuul-netflix-springone-platform) представил некоторые Zuul-приложения. Разработанный им компонент используется для обслуживания основной массы интерфейсов для Netflix.com и более чем 1000 типов устройств. Он представляет собой уровень переходников, обрабатывающих сотни разновидностей протоколов и версий устройств. В конкретной среде развертывания Netflix он предшествует более чем 50 балансировщикам нагрузки AWS Elastic Load Balancers и обрабатывает десятки миллиардов запросов в день в трех разных AWS-регионах. Он развернут в более чем 20 промышленных Zuul-кластерах, выходящих на более чем десять исходных систем для Netflix.com. Короче говоря, компонент Zuul — что-то вроде рабочей лошадки.

Фильтры Zuul глубоко укоренились в эксплуатационной среде. Они содержат динамическую логику маршрутизации, управляют балансировкой нагрузки и защитой сервисов, а также механизмами отладки и анализа проблем, связанных с каналами данных. Zuul-шлюз может быть инструментом обеспечения качества, поскольку предоставляет единое место для наблюдения за потоком данных, проходящих через систему.

Фильтры Zuul можно применять в целях создания высокоуровневого контекста для запросов к системе, обрабатывая проблемы геолокации, распространения cookie-файлов и расшифровки маркеров. Фильтры позволяют решать такие сквозные

задачи, как проведение аутентификации. Их можно использовать в приведении запросов или ответов к общему стандарту, в обработке особенностей, присущих тому или иному устройству, подобных требованиям к кодировке фрагментов, в усечении заголовков, в исправлении URL и т. д. Фильтры Zuul — идеальное место для выполнения целевой маршрутизации, возможно, чтобы изолировать вполне определенные (пусть даже и ошибочные) запросы. Они могут использоваться для формирования трафика, выполнения глобальной универсальной маршрутизации, отработки логики обхода нерабочих узлов и аварийного переключения зон, а также обнаружения и предотвращения вредоносных атак.

Все эти возможности носят универсальный характер: они применимы ко всем разновидностям сервисов. А вот для бизнес-логики, соответствующей конкретному сервису, Zuul совсем не подходящее место. Ее нужно выносить за пределы пограничного уровня.

Фильтры Zuul не следует отождествлять с традиционным сервлетом `Filter`. Предполагается, что вы фильтруете результаты *проксированных* запросов, а не те запросы, которые, собственно, направляются к самому пограничному сервису.

Фильтры Zuul — идеальное место для маршрутизации от клиента к нижестоящим сервисам. Если ваши сервисы зарегистрированы в реестре сервисов, на что настроена реализация Spring Cloud `DiscoveryClient`, то все сводится к элементарному добавлению к коду вашего приложения аннотации `@EnableZuulProxy` (наряду с добавлением к пути к классам записи `org.springframework.cloud : spring-cloud-starter-zuul`). После этого Spring Cloud, работая с `DiscoveryClient`, настроит для вас маршруты нижестоящих сервисов через прокси-сервер. Опрос этих маршрутов можно провести, используя `RouteLocator`, как делается в примере 8.9.

Пример 8.9. ZuulConfiguration.java

```
package greetings;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;
import org.springframework.cloud.netflix.zuul.filters.RouteLocator;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableZuulProxy
class ZuulConfiguration {

 @Bean
 CommandLineRunner commandLineRunner(RouteLocator routeLocator) {  ❶
  Log log = LogFactory.getLog(getClass());
  return args -> routeLocator.getRoutes().forEach(
   r -> log.info(String.format("%s (%s) %s", r.getId(), r.getLocation(),
```

```
    r.getFullPath())));
 }
}
```

❶ `RouteLocator` — абстракция, имеющая несколько интересных реализаций. В Spring Cloud при должной конфигурации настраивается реализация, осведомленная о наличии `DiscoveryClient`.

Zuul устанавливает удобные маршруты на основе идентификаторов сервисов. Следовательно, предполагая, что уже установили сервис реестра `service-registry` и сервис приветствий `greetings-service`, в выводе предыдущего примера мы должны увидеть, что имеется сервис, а именно `greetings-service`, доступный для потребления через прокси-сервер. Ожидается, что `greetings-client` будет запущен на http://localhost:8082, вследствие чего `greetings-service` можно вызвать на http://localhost:8082/greetings-service/greet/World, где контекстный путь, `greetings-service`, взят из идентификатора сервиса в реестре. Можно также устанавливать произвольные маршруты. Требуемая для этого конфигурация показана в примере 8.10.

Пример 8.10. application.properties

```
zuul.routes.hi.path = /lets/**  ❶

zuul.routes.hi.serviceId = greetings-service  ❷
```

❶ Путь к пограничному сервису, который может быть отображен на...

❷ ...сервис (или же альтернативно вместо `serviceId` в качестве `.location` указывается полный URL).

В данной конфигурации мы отображаем запросы к `greetings-service` — все от `/` и ниже до `/lets/*` — на пограничный сервис. Запустите приложение и попробуйте им воспользоваться, обратившись по адресу http://localhost:8082/lets/greet/World. Эти маршруты идеально подходят для сохранения в сервере Spring Cloud Config, где позже их можно обновить без перезапуска пограничного сервиса; вы можете заставить Zuul выполнять их динамическую перезагрузку.

Один из способов достижения этого, рассмотренный в ходе изучения конфигурации 12-факторного приложения в главе 3, заключается в простом вызове конечной точки обновления Spring Cloud. Измените конфигурацию в сервере Spring Cloud Config, зафиксируйте изменения (в Subversion или Git), а затем отправьте пустой HTTP POST-запрос (например, `curl -d{} http://localhost:8082/refresh`) для обновления конечной точки или запрос на шину событий Spring Cloud (`/bus/refresh`). В результате состоится публикация `RefreshScopeRefreshedEvent`, на которую компоненты в устройстве Zuul прореагируют собственным переконфигурированием.

Вместо этого, как показано на рис. 8.1, можно воспользоваться конечной точкой актуатора `/routes`, и Zuul проведет ее автоматическую конфигурацию. Если отправить HTTP GET-запрос, то данная точка возвратит сконфигурированные маршруты, а в случае отправки HTTP POST-запроса запустится обновление. Это альтернатива

использованию конечной точки `refresh`, поскольку она применяется только к Zuul-маршрутам и ни к какой другой (потенциально более тяжеловесной) инфраструктуре, способной отлавливать событие обновления.

Рис. 8.1. Zuul-маршруты из конечной точки routes

Zuul также аннулирует и перенастраивает существующие маршруты, основанные на событиях пульса `HeartbeatEvent`, публикуемых с `DiscoveryClient`. Таким образом, если узел должен сняться с регистрации в Eureka (или в любом другом сервисе регистрации, поддерживаемом через `DiscoveryClient`), то Zuul удалит маршрут из своей конфигурации.

Конечно же, при наличии какого-либо внутреннего состояния, зависящего от данных маршрутов, можно напрямую отслеживать эти события и реагировать на них каким-либо образом, подходящим приложению (пример 8.11).

Пример 8.11. RoutesListener.java

```
package greetings;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.cloud.client.discovery.event.HeartbeatEvent;
import org.springframework.cloud.netflix.zuul.RoutesRefreshedEvent;
import org.springframework.cloud.netflix.zuul.filters.RouteLocator;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
```

```
class RoutesListener {

 private final RouteLocator routeLocator;

 private final DiscoveryClient discoveryClient;

 private Log log = LogFactory.getLog(getClass());

 @Autowired
 public RoutesListener(DiscoveryClient dc, RouteLocator rl) {
  this.routeLocator = rl;
  this.discoveryClient = dc;
 }

 ❶
 @EventListener(HeartbeatEvent.class)
 public void onHeartbeatEvent(HeartbeatEvent event) {
  this.log.info("onHeartbeatEvent()");
  this.discoveryClient.getServices().stream().map(x -> " " + x)
   .forEach(this.log::info);
 }

 ❷
 @EventListener(RoutesRefreshedEvent.class)
 public void onRoutesRefreshedEvent(RoutesRefreshedEvent event) {
  this.log.info("onRoutesRefreshedEvent()");
  this.routeLocator.getRoutes().stream().map(x -> " " + x)
   .forEach(this.log::info);
 }
}
```

❶ Отслеживание событий, публикуемых механизмами `DiscoveryClient`.

❷ Отслеживание исключительно тех событий, которые публикуются механизмами маршрутизации Zuul.

По сути, компонент Zuul является прокси-сервером. Можно создать реализации `ZuulFilter`, встроенные в жизненный цикл этого сервера, или же обычные реализации `javax.servlet.Filter`, фильтрующие запросы, сделанные к Zuul Servlet. Посмотрим на простой фильтр `Filter`, предоставляющий заголовки контроля доступа, разрешающие совместное использование сценариев запросов из разных источников, чтобы другие JavaScript-клиенты могли вызывать эти сервисы из другого источника. В простом приложении вполне разумно было бы обслуживать функциональные средства JavaScript и HTML5 из того же приложения, которое предоставляет прокси-сервер Zuul. В более сложном приложении клиентский (статический) код будет, вероятнее всего, существовать в сети доставки контента — content delivery network (CDN). В таком случае мы столкнемся с проблемой: вызовы из кода JavaScript в адрес пограничного сервиса, содержащего прокси-сервер, наткнутся на защиту от подделок запросов ресурсов из разных источников, поскольку

код JavaScript помещается в «песочницу» и не может отправлять запросы за пределы своего исходного домена.

Единственный способ обойти эту проблему — предоставить заголовки `Access-Control-Allow-Headers`, `Access-Control-Allow-Methods`, `Access-Control-Allow-Origin` и при необходимости `Access-Control-Max-Age`, явным образом разрешая запросы от клиента HTML5 к пограничному сервису. Можно напрямую направлять запросы от клиента HTML5 непосредственно к каждому из наших сервисов, фигурирующих в реестре сервисов, но тогда придется вносить изменения в каждый отдельно взятый нижестоящий микросервис, включая эти заголовки контроля доступа. Если бы потребовалось увидеть реакцию в нижестоящих сервисах, то пришлось бы просить все вовлеченные команды обновить разрабатываемый ими код. На это может уйти много времени, и в то же время будет потеряна автономия, к которой мы стремились в первую очередь при переходе к использованию микросервисов. Вместо этого настроим фильтр в пограничном сервисе, который сделает все необходимое.

Самый простой вариант — создать фильтр, разрешающий *все* запросы путем указания символа `*`. Но здесь имеются некоторые ограничения. Если в наш CORS-запрос включаются cookie-файлы, то нужно в нашем коде JavaScript указать также `withCredentials: true` и, кроме того, наш сервис должен возвратить `Access-Control-Allow-Credentials: true`. Даже тогда наш запрос будет широко открыт для всех клиентов! Один из вариантов заключается в содержании в пограничном сервисе динамического белого списка, с помощью которого станут фильтроваться клиенты в зависимости от того, зарегистрированы они в реестре или нет. Все, что нужно для поддержки этого варианта, нам даст `DiscoveryClient`.

Если наши статические ресурсы находятся где-то в CDN-сети, то нужно, чтобы в реестре появились исходные данные CDN (IP-адрес или имя хоста). Реализовать это можно явным образом за счет API сервиса реестра. Кроме того, если используется Netflix Eureka, то для выполнения регистрации без нашего участия можно воспользоваться смежным процессом, например Netflix Prana.

Посмотрим на простой по конструкции фильтр `javax.servlet.Filter`, предназначенный для динамически формируемых запросов белого списка к пограничному сервису (пример 8.12).

Пример 8.12. CorsFilter.java — приложение нужно будет запустить с активированным Spring-профилем cors

```
package greetings;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.ServiceInstance;
```

```
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.cloud.client.discovery.event.HeartbeatEvent;
import org.springframework.context.annotation.Profile;
import org.springframework.context.event.EventListener;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;
import org.springframework.http.HttpHeaders;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.net.URI;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.stream.Collectors;

@Profile("cors")
@Component
@Order(Ordered.HIGHEST_PRECEDENCE + 10)
class CorsFilter implements Filter {

 private final Log log = LogFactory.getLog(getClass());

 private final Map<String, List<ServiceInstance>> catalog =
    new ConcurrentHashMap<>();

 private final DiscoveryClient discoveryClient;

 ❶
 @Autowired
 public CorsFilter(DiscoveryClient discoveryClient) {
  this.discoveryClient = discoveryClient;
  this.refreshCatalog();
 }

 ❷
 @Override
 public void doFilter(ServletRequest req, ServletResponse res,
    FilterChain chain)
  throws IOException, ServletException {
  HttpServletResponse response = HttpServletResponse.class.cast(res);
  HttpServletRequest request = HttpServletRequest.class.cast(req);
  String originHeaderValue = originFor(request);
  boolean clientAllowed = isClientAllowed(originHeaderValue);

  if (clientAllowed) {
   response.setHeader(HttpHeaders.ACCESS_CONTROL_ALLOW_ORIGIN,
```

```
    originHeaderValue);
  }

  chain.doFilter(req, res);
}

❸
private boolean isClientAllowed(String origin) {
 if (StringUtils.hasText(origin)) {
  URI originUri = URI.create(origin);
  int port = originUri.getPort();
  String match = originUri.getHost() + ':' + (port <= 0 ? 80 : port);

  this.catalog.forEach((k, v) -> {
  String collect = v
   .stream()
   .map(
    si -> si.getHost() + ':' + si.getPort() + '(' + si.getServiceId() + ')')
   .collect(Collectors.joining());
 });

  boolean svcMatch = this.catalog
   .keySet()
   .stream()
   .anyMatch(
    serviceId -> this.catalog.get(serviceId).stream()
     .map(si -> si.getHost() + ':' + si.getPort())
     .anyMatch(hp -> hp.equalsIgnoreCase(match)));
  return svcMatch;
 }
 return false;
}

❹
@EventListener(HeartbeatEvent.class)
public void onHeartbeatEvent(HeartbeatEvent e) {
 this.refreshCatalog();
}

private void refreshCatalog() {
 discoveryClient.getServices().forEach(
  svc -> this.catalog.put(svc, this.discoveryClient.getInstances(svc)));
}

@Override
public void init(FilterConfig filterConfig) throws ServletException {
}

@Override
public void destroy() {
}

private String originFor(HttpServletRequest request) {
```

```
  return StringUtils.hasText(request.getHeader(HttpHeaders.ORIGIN)) ? request
   .getHeader(HttpHeaders.ORIGIN) : request.getHeader(HttpHeaders.REFERER);
 }
}
```

❶ Для опроса топологии сервиса мы воспользуемся Spring Cloud `DiscoveryClient`.

❷ Как и следовало ожидать, метод `doFilter`, предназначенный для стандартного сервлета `Filter`...

❸ ...устанавливает заголовки управления доступом, если клиент определен находящимся в пределах известного набора сервисов.

❹ Упреждающее аннулирование локальной кэш-памяти с последующим кэшированием набора зарегистрированных сервисов при каждом получении `DiscoveryClient` события пульса.

После выполнения описанного для испытания понадобится статическое HTML5-приложение. Это приложение в модуле `html5-client` пользуется регистрацией и обнаружением сервиса с помощью абстракции Spring Cloud `DiscoveryClient`. Оно предоставляет конечную точку `/greetings-client-uri`. Та, в свою очередь, возвращает правильный URI для экземпляра пограничного сервиса `greetings-client`, который наш код JavaScript может задействовать для создания HTTP Ajax-вызова пограничного сервиса на другом узле (пример 8.13).

Пример 8.13. Html5Client.java

```
package client;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;
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.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collections;
import java.util.Map;
import java.util.Optional;

@RestController
@EnableDiscoveryClient
@SpringBootApplication
public class Html5Client {

 private final LoadBalancerClient loadBalancerClient;

 @Autowired
 Html5Client(LoadBalancerClient loadBalancerClient) {
```

```
  this.loadBalancerClient = loadBalancerClient;
 }

 public static void main(String[] args) {
  SpringApplication.run(Html5Client.class, args);
 }

 ❶
 //@formatter:off
 @GetMapping(value = "/greetings-client-uri",
         produces = MediaType.APPLICATION_JSON_VALUE)

 //@formatter:on
 Map<String, String> greetingsClientURI() throws Exception {
  return Optional
    .ofNullable(this.loadBalancerClient.choose("greetings-client"))
    .map(si -> Collections.singletonMap("uri", si.getUri().toString()))
    .orElse(null);
 }
}
```

❶ Эта конечная точка возвращает один экземпляр нижестоящей конечной точки `greetings-client`, к чьим конечным точкам можно обращаться благодаря заголовкам пограничных сервисов `Access-Control-*`. Здесь применяется `LoadBalancerClient` для возвращения экземпляра с использованием любой настроенной стратегии балансировки нагрузки. По умолчанию балансировщик нагрузки на стороне клиента станет задействовать Ribbon компании Netflix, который, в свою очередь, прибегнет к балансировке нагрузки методом циклического перебора.

Сам код JavaScript не представляет особого интереса — это jQuery-приложение (кто-нибудь еще помнит jQuery?), использующее разрешение URI из приложения `html5-client` для вызова пограничного сервиса (`greetings-client`), который, в свою очередь, вызывает `greetings-service`. Для отображения ответа приложение обновляет DOM локальной страницы.

Приложение HTML5 вводит jQuery с помощью проекта WebJars (http://www.webjars.org/), позволяющего управлять JavaScript-зависимостями в Maven-сборке. Чтобы получить jQuery, нам нужно было только добавить к нашей сборке две зависимости: org.webjars : jquery : 2.1.1, которая ввела саму библиотеку, и org.webjars : webjars-locator. Последнюю Spring Boot и другие среды на основе JVM можно применять для отображения неопределенных импортов JavaScript, где версия библиотеки намеренно неясна (<script src="/webjars/jquery/jquery.min.js"></script>), на конкретные версии библиотеки, указанные в пути к классам и импортированные с помощью WebJars.

Код показан в примере 8.14.

Пример 8.14. В коде используется jQuery (вы тоже считаете, что это почти ретророман?)

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8"/>
    <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
    <title>Demo</title>
    <meta name="description" content=""/>
    <meta name="viewport" content="width=device-width"/>
    <base href="/"/>
</head>
<body>

<div id="message"></div>

<script src="/webjars/jquery/jquery.min.js"></script>
<script>

   ❶
   var greetingsClientUrl = location.protocol + "//" + window.location.host
       + "/greetings-client-uri";
   $.ajax({url: greetingsClientUrl}).done(function (data) {
       var nameToGreet = window.prompt("who would you like to greet?");
       var greetingsServiceUrl = data['uri'] + "/lets/greet/" + nameToGreet;
       console.log('greetingsServiceUrl: ' + greetingsServiceUrl);
       ❷
       $.ajax({url: greetingsServiceUrl}).done(function (greeting) {
           $("#message").html(greeting['greeting']);
       });
   });
</script>
</body>
</html>
```

❶ Загрузка с текущего хоста информации о том, где находится `greetings-client`.

❷ Вызов сервиса из другого источника с помощью URI, прошедшего операцию разрешения.

Чтобы увидеть все в работе, запустите следующие сервисы (в указанном порядке, иначе придется ждать, пока все они не подключатся к сервису регистрации): `service-registry`, `greetings-service`, `edge-service` (с профилем `cors`) и `html5-client`. Откройте Eureka (http://localhost:8761), чтобы найти IP-адрес, по которому теперь доступен экземпляр `html5-client`. Скопируйте этот адрес и вставьте его в адресную строку браузера с указанием порта 8083, как показано на рис. 8.2.

Рис. 8.2. CORS-запрос от нашего клиента HTML5

Для доступа к клиенту HTML5 важно воспользоваться верным хостом и портом, в противном случае CorsFilter не разрешит запрос, поскольку проверяет поступающие запросы на хосты и порты, зарегистрированные в реестре сервисов. (К примеру, вполне возможно, что он не станет работать с localhost или применять его.)

Неплохо! Простой сервлет `Filter` может выполнить множество интересных действий. Но Servlet API не имеет никаких конкретных сведений о том, какой именно проксированный нижестоящий сервис мы вызвали. Если нужно это знать, то следует создать Zuul-фильтр.

Настраиваемый Zuul-фильтр. `CorsFilter` — стандартный фильтр `Filter`, работающий для всех запросов, сделанных в адрес пограничного сервиса, включая все запросы к Zuul (и Zuul Servlet, который Spring Boot подключает автоматически). При этом в Zuul имеется специальный канал для фильтруемых запросов, специально направляемый через прокси-сервер Zuul. Существует четыре исходных типа Zuul-фильтров, хотя, конечно же, в случае необходимости можно добавить пользовательские типы.

- *Предварительные фильтры (pre)*, применяемые до маршрутизации запроса.
- *Фильтры маршрутизации (routing)*, позволяющие обрабатывать фактическую маршрутизацию запроса.
- *Последующие фильтры (post)*, используемые после маршрутизации запроса.
- *Фильтры ошибки (error)*, задействуемые в случае возникновения ошибки в ходе обработки запроса.

При соблюдении соответствующих условий автоконфигурации в Spring Cloud регистрируются несколько полезных готовых Zuul-фильтров:

- `AuthenticationHeaderFilter` (pre) — фильтр, выискивающий запросы, прошедшие через прокси-сервер, и удаляющий заголовок авторизации перед их отправкой к нижестоящим уровням;
- `OAuth2TokenRelayFilter` (pre) — фильтр, распространяющий маркер доступа OAuth при его доступности в поступившем запросе;
- `ServletDetectionFilter` (pre) — определяет, прошел ли HTTP-сервлет запрос через канал Zuul-фильтра;
- `Servlet30WrapperFilter` (pre) — заключает поступивший HTTP-запрос в оболочку декоратора Servlet 3.0;
- `FormBodyWrapperFilter` (pre) — заключает поступивший HTTP-запрос в оболочку декоратора, содержащего метаданные о выкладке файла, состоящего из нескольких частей;
- `DebugFilter` (pre) — вносит параметры для поддержки отладки, если запрос содержит Archaius-свойство;
- `SendResponseFilter` (post) — выдает на выходе ответ (при условии, что он был предоставлен вышестоящими фильтрами);
- `SendErrorFilter` (post) — если ответ содержит ошибку, то данный фильтр перенаправляет ее на настраиваемую конечную точку;
- `SendForwardFilter` (post) — если ответ содержит перенаправление, то этот фильтр выполняет требуемое действие;
- `SimpleHostRoutingFilter` (route) — фильтр, который получает поступающий запрос и на основе URL решает, на какой узел направить проксируемый запрос;
- `RibbonRoutingFilter` (route) — фильтр, получающий поступающий запрос и решающий на основе URL, на какой узел направить проксируемый запрос. Это делается с помощью настраиваемой маршрутизации и балансировки нагрузки на стороне клиента, работающей на основе Netflix Ribbon.

Предположим, что нужно добавить еще одну возможность общего, сквозного характера. В качестве простого примера приведем нормировщик (ограничитель темпа). Норма помогает системам гарантировать выполнение соглашений об уровне услуг (SLA), распределяя по нижестоящим сервисам запросы, превышающие определенную норму. Алгоритм маркерной корзины (https://en.wikipedia.org/wiki/Token_bucket) основан на аналогии с корзиной фиксированной емкости, в которую маркеры, обычно представляющие блок байтов или один пакет заданного размера, добавляются с фиксированной скоростью. Альтернативный алгоритм «протекающего ведра» (https://en.wikipedia.org/wiki/Leaky_bucket) основан на следующей аналогии: протекающее ведро будет переполнено, если средняя скорость, с которой наливают воду, превысит норму протечки ведра либо в ведро сразу нальют объем воды, превышающий его емкость.

Реализовать простой нормировщик с помощью Zuul-фильтра довольно легко. В этой реализации используется Guava-класс `RateLimiter`. Данный нормировщик весьма примитивен, поскольку ограничивает темп выдачи запросов ко всем

проксируемым сервисам и экземплярам сервисов, а не к экземпляру сервиса или даже не к конкретному сервису. Сначала, как показано в примере 8.15, нужно настроить конфигурацию Guava `RateLimiter`.

Пример 8.15. Guava RateLimiter настраивается на разрешение запроса только каждые десять секунд; это абсурдно низкое значение специально выбрано для иллюстрации происходящего

```
package greetings;

import com.google.common.util.concurrent.RateLimiter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Profile("throttled")
@Configuration
class ThrottlingConfiguration {

 @Bean ❶
 RateLimiter rateLimiter() {
  return RateLimiter.create(1.0D / 10.0D);
 }
}
```

❶ Определение количества запросов, разрешенных в секунду. Здесь указано, что разрешено 0,10 запроса в секунду. Или, иначе, один запрос разрешен каждые десять секунд.

Теперь, располагая этим кодом, можно без особого труда разработать `ZuulFilter`, ограничивающий количество поступающих запросов (пример 8.16).

Пример 8.16. ZuulFilter-нормировщик

```
package greetings;

import com.google.common.util.concurrent.RateLimiter;
import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import com.netflix.zuul.exception.ZuulException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Profile;
import org.springframework.core.Ordered;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.util.ReflectionUtils;

import javax.servlet.http.HttpServletResponse;

@Profile("throttled")

@Component
```

```
class ThrottlingZuulFilter extends ZuulFilter {

 private final HttpStatus tooManyRequests = HttpStatus.TOO_MANY_REQUESTS;

 private final RateLimiter rateLimiter;

 @Autowired
 public ThrottlingZuulFilter(RateLimiter rateLimiter) {
  this.rateLimiter = rateLimiter;
 }

 ❶
 @Override
 public String filterType() {
  return "pre";
 }

 ❷
 @Override
 public int filterOrder() {
  return Ordered.HIGHEST_PRECEDENCE;
 }

 ❸
 @Override
 public boolean shouldFilter() {
  return true;
 }

 ❹
 @Override
 public Object run() {
  try {
   RequestContext currentContext = RequestContext.getCurrentContext();
   HttpServletResponse response = currentContext.getResponse();

   if (!rateLimiter.tryAcquire()) {

   ❺
   response.setContentType(MediaType.TEXT_PLAIN_VALUE);
   response.setStatus(this.tooManyRequests.value());
   response.getWriter().append(this.tooManyRequests.getReasonPhrase());

   ❻
   currentContext.setSendZuulResponse(false);

   throw new ZuulException(this.tooManyRequests.getReasonPhrase(),
    this.tooManyRequests.value(), this.tooManyRequests.getReasonPhrase());
   }
  }
  catch (Exception e) {
```

```
    ReflectionUtils.rethrowRuntimeException(e);
   }
   return null;
  }
}
```

1. Этот конкретный фильтр является pre-фильтром, запускаемым перед выдачей проксируемого запроса.
2. Фильтр должен запускаться как можно раньше...
3. ...и должен запускаться всегда. Но его можно отключить на основе какого-либо атрибута запроса или данных конфигурации.
4. Метод `run()` — центральный элемент `ZuulFilter`. Здесь выполняется фильтрация данного запроса.
5. Фильтр пытается получить разрешение от Guava `RateLimiter` и в случае неудачи отправляет назад вместе с сообщением об ошибке код состояния HTTP `429 – Too Many Requests` (Слишком много запросов).
6. Явно прерывает проксирование запроса. При невозможности сделать это запрос будет проксирован автоматически.

Запустите пограничный сервис `greetings-client` с профилем `throttled`. Если посетить пограничный сервис `greetings-client`, скажем, по адресу http://localhost:8082/lets/greet/Eva, то каждые десять секунд будет приходить корректный ответ в формате JSON; в противном случае ответ будет гласить, что запросов слишком много — `Too Many Requests`.

Обеспечение безопасности в пограничной зоне

Пограничные сервисы представляют первый рубеж обороны от внешних, возможно, вредоносных запросов. Данные сервисы вполне логично определить местом решения таких вопросов общего характера, как обеспечение безопасности. Это весьма кропотливая задача и, поскольку ее решение должно распространяться на каждый сервис, может закончиться ненужным повторением. Обеспечение безопасности — та самая область, в которой проявляются комплексные преимущества использования той или иной платформы (например, Cloud Foundry) и где реально может помочь подход в форме соглашения, заключаемого с помощью конфигурации, поддерживаемый Spring Boot. По мере поступления запросов в систему мы будем проводить их аутентификацию в пограничной зоне, а затем распространять контекст аутентификации на нижестоящие сервисы.

Аутентификация просто означает, что нужно идентифицировать актор, сделавший конкретный запрос. Можно ли избежать применения имен пользователей и паролей? Или x509-сертификатов? Если мы просто пытаемся ответить на вопрос «Кем сделан этот запрос?», то эти и подобные им альтернативы могут показаться достаточными при условии, что выполнены корректно. Но сделать это правильно нелегко

(https://www.youtube.com/watch?v=o4nt9IR8iL8&feature=youtu.be&t=23m3s). В идеале нужно настроить хеш на минимальный период времени для проверки (возможно, на полсекунды). Это значит, что минимальный период времени на проверку имени пользователя и пароля (плюс на всю реальную работу) составляет полсекунды. Пароли затрудняют контроль над тем, кто именно получил доступ к системе, поскольку при утечке одного из них станут недействительными все случаи его применения. Кроме того, люди задействуют в разных системах одни и те же пароли. Обычно вопрос о том, *кем сделан запрос,* не единственный, на который вы пытаетесь найти ответ, даже если гарантируете достаточную произвольность паролей и безопасность их хранения (чего вы, вероятнее всего, не можете гарантировать).

В сегодняшнем мире не все клиенты создаются равными друг другу. Знание клиента, с которого пользователь делает запрос, играет практически такую же важную роль, как и знание о том, кто именно сделал запрос. При установке сервисов для разнообразного мира клиентов нужно брать в расчет оба измерения. Поставщик API может больше доверять одним клиентам, чем другим, и, соответственно, следует ввести ограничение доступа.

В качестве примера подумайте обо всех местах, где используете Facebook. Последний предлагает многочисленные клиенты: на iPad, в качестве веб-страницы, Android-приложения и т. д. На многих сайтах довольно часто попадается кнопка с надписью Sign in with Facebook (Войдите через Facebook). С ее помощью вы отправляетесь на Facebook.com, где вам предложат войти в эту социальную сеть (если вы еще не входили), а затем подтвердить определенные разрешения для запрашивающего сайта. Как только доступ будет подтвержден, Facebook перенаправит вас обратно на исходный сайт, который теперь получит все, что ему необходимо для доступа к информации о вашем профиле на Facebook, и заполнит собственный профиль вашей информацией, проведя вашу фактическую регистрацию. В данном примере вы никогда не вводите ваш пароль для Facebook в стороннем веб-приложении. Сторонний сайт имеет лишь ограниченное разрешение действовать от вашего имени, считывая конкретную информацию и, возможно, выполняя конкретные действия. Как только вы примете решение о закрытии своего профиля на стороннем сайте, вам нужно будет просто зайти на Facebook и аннулировать разрешения для данного сайта. Важно, что при этом вам не придется менять свой пароль. Он сохранит свою действенность, и все ваши другие подключенные клиенты будут по-прежнему актуальны. Различие заключается только в том, что актуальность здесь теряет *ваша* учетная запись, но только при доступе через этот сторонний сайт (или клиент).

Сравните это с впечатлениями от использования официального Facebook-приложения для iPhone. iOS-приложение было разработано специалистами Facebook. Их, естественно, меньше всего беспокоит злоупотребление вашим доверием и выполнение вредоносных действий от вашего имени с информацией вашего профиля и с вашей учетной записью на Facebook.com. В данном случае приложение Facebook.app (ввиду запуска на вашем iOS-устройстве) будет радо возможности позволить вам ввести на устройстве имя пользователя и пароль. Ему не нужно перенаправлять

вас на Facebook.com в браузере, в этом нет необходимости. На Facebook.com уже есть ваш пароль, и они с великой радостью обеспечат безопасность экрану входа в приложение iPhone.

Мы имеем дело с одним профилем и двумя разными клиентами Facebook.com, к одному из которых больше доверия, чем к другому. У каждого клиента были разные разрешения. Официальное iPhone-приложение Facebook.com имеет, по сути, полный контроль над вашей учетной записью, а вот сторонний сайт может только выполнить считывание адреса вашей электронной почты и полное имя.

OAuth

Займемся OAuth. Данный стандарт имеет три воплощения: 1.0, 1.0.a и 2.0. Наиболее актуальный и рекомендуемый выпуск — OAuth 2.0, так что на нем и сконцентрируемся. *OAuth* (это сокращение от выражения «открытая авторизация» — open authorization) представляет собой стандарт для авторизации в Интернете на основе токенов. Они сокращают промежуток времени открытости имен пользователей и паролей. Токены отвязывают клиентов от паролей, гарантируя тем самым абсолютную невозможность блокировки вашего доступа к своей собственной учетной записи со стороны клиента-нарушителя. Токены могут представлять клиента, действующего от имени конкретного пользователя, или же токен без конкретного контекста пользователя.

OAuth является протоколом *авторизации* (отвечая на вопрос «Что разрешено пользователю?»), а не протоколом *аутентификации* (который отвечает на вопрос «Кто этот пользователь?»), вследствие чего вам все равно придется где-то управлять аутентификацией, а затем уже передавать управление OAuth. Протокол OAuth предоставляет клиентам «защищенный делегированный доступ» к сервисам от имени владельца данных, доступных на сервисе. После аутентификации для авторизации запросов клиенты применяют токены. Для каждого клиента используется свой токен.

При рассмотрении вопросов, касающихся OAuth, применяется следующая терминология.

- *Клиент:* приложение, выполняющее защищенные запросы. Зачастую, но не всегда, клиент делает запросы от имени конечного пользователя (владельца ресурса). Это может быть iPhone- или Android-приложение, браузерный клиент HTML5 и даже часы! Но клиентам не обязательно иметь пользовательский контекст. Так, клиент может представлять аналитическую задачу отдела сопровождения операций, которой нужны ограниченные разрешения. В примере с Facebook.com им может быть ваше обычное iOS-приложение или браузер, запущенный на настольном компьютере.
- *Владелец ресурса:* обычно этот термин относится к конечному пользователю (Juergen, Michelle, Dave, Margarette и т. д.), который может предоставить кли-

енту разрешение на доступ к своей информации. В примере с Facebook.com это можете быть вы или я.

- *Сервер ресурса:* сервер, на котором находится защищенный ресурс, способный принимать защищенные запросы, содержащие токены, и отвечать на них. Этот термин имеет отношение к защищенному API, с которым клиент пытается наладить связь. В примере с Facebook.com это может иметь отношение к API Facebook, поддерживающим доступ ко всем элементам: вашему профилю, схеме друзей и т. д.
- *Сервер авторизации:* сервер, выдающий токен доступа клиенту после успешной аутентификации владельца ресурса и получения авторизации. В Facebook.com это сервис, на который вы перенаправляетесь с целью аутентификации и который предоставляет токены. Для всех целей и задач относительно Facebook.com сервер авторизации и сервер ресурса — одно и то же, но не обязательно.

Первый шаг в OAuth — получить авторизацию, в основном для прохождения пункта аутентификации. Достичь этой цели помогут четыре широко известных типа предоставлений или процессов.

Приложения на стороне сервиса

В рассмотренном выше процессе Sign in with Facebook (Войдите через Facebook) клиент перенаправляется из стороннего сайта на экран входа в Facebook.com. В своем браузере вы увидите в адресной строке обнадеживающий значок безопасности в виде висячего замка, показывающий, что вы находитесь в сертифицированном месте, работающем через HTTPS. Вы можете успокоиться, зная, что находитесь на странице Facebook.com. Затем вводите свои имя пользователя и пароль и отправляете данные формы, после чего вам предлагают передать определенные полномочия запрашивающему клиенту. Если щелкнуть на кнопке разрешения, то Facebook.com перенаправит вас к запрашивающему клиенту с кодом авторизации. Клиент, находящийся на серверной стороне, вызовет сервер авторизации Facebook.com и предоставит код авторизации, который будет обменян на токен доступа. Теперь запрашивающий клиент получит токен доступа, сохранит его в базе данных и станет применять для совершения вызовов от вашего имени к серверу ресурсов Facebook.com.

В данном сценарии токен доступа никогда не утечет к коду JavaScript на стороне клиента, запущенного в браузере. Единственным, что будет в поле видимости клиента (или пользователя, управляющего клиентом), станет код авторизации. Эта косвенность защищает от атак через посредника, при которых кто-либо мог бы перехватить токен доступа и применять его для совершения вызовов от вашего имени.

Весь этот процесс называется *предоставлением кода авторизации*. Он используется при необходимости предотвратить утечку токена доступа куда-нибудь, где его можно было бы легко перехватить.

Одностраничные приложения на HTML5 и JavaScript

Одностраничные приложения (single-page applications, SPAs) полностью запускаются в браузере после загрузки исходного кода с веб-страницы. Разумеется, в SPA не стоит держать никаких секретов, поскольку его исходный код можно увидеть, щелкнув на пункте контекстного меню View Source (Просмотр кода страницы). В случае SPA пользователь щелкает на кнопке входа в приложение и получает, как и раньше, приглашение на одобрение доступа. Если пользователь щелкнет на кнопке, разрешающей доступ, то сервис перенаправляет его обратно на сайт одностраничного приложения с токеном доступа во фрагменте URL (в той части, которая следует за символом # URL, например http://some-third-party-service.com/an_oauth_callback#token=12345..). Приложения JavaScript могут прочитать эту часть URL, а изменение фрагмента в URL не заставляет браузер выполнять новый запрос. Это называется *неявным предоставлением*. В данном случае процесс *специально* оптимизирован под утечку токена доступа к коду клиента из-за отсутствия компонента на стороне сервиса.

Приложения без пользователей

Во всех рассмотренных до сих пор примерах речь шла о клиенте, действующем от имени пользователя с целью получить доступ к защищенным ресурсам последнего. Клиент, не имеющий конкретного пользователя, может запросить токен доступа, применяя лишь учетные данные клиента. Это может пригодиться для приложений, нуждающихся в доступе к защищенной информации, но не в конкретном пользовательском контексте вроде пакетных приложений или аналитики либо чего-то, не имеющего интерактивного режима работы. Такой клиент будет аутентифицирован и может иметь свои ограниченные права доступа. Это называется *предоставлением учетных данных клиентов*.

Доверенные клиенты

OAuth 2 поддерживает предоставление пароля, которое может пригодиться при создании клиента для сервиса, доверяющего клиенту. Представим, что разрабатывается пользовательский интерфейс Facebook.app для Facebook.com. В этом случае UI с необходимостью перенаправлять на Facebook.com внутри Facebook.app с последующим *удостоверением* наличия у Facebook.app разрешений на чтение и манипулирование вашими данными Facebook.com кажется несколько избыточным (у Facebook.com уже есть ваши имя пользователя и пароль!). В данном случае доверенный клиент, созданный специалистами, работающими в самой компании Facebook, просто передаст имя пользователя и пароль, принадлежащие данному пользователю. Риск неожиданного получения приложением Facebook.app вашего имени пользователя и пароля, конечно же, отсутствует, поскольку они у него уже есть! Если же вы не можете поверить, что Facebook.app не станет задействовать

ваши данные Facebook.com с целью нанести вред, то, вероятно, вам вообще нужно пересмотреть вопрос применения Facebook! В этом процессе пользователь передает имя и пароль непосредственно серверу авторизации в обмен на маркер доступа. Процедура называется *предоставлением учетных данных владельца ресурса*.

Полноценное изучение OAuth выходит за рамки нашей книги. Мы рекомендуем по этой теме превосходную (и весьма лаконичную!) книгу Getting Started with OAuth 2.0 (http://shop.oreilly.com/product/0636920021810.do), написанную Райаном Бойдом (Ryan Boyd) (O'Reilly).

Мы установим сервер авторизации OAuth и защитим наших клиентов с помощью стандарта OAuth, но сначала усвоим ряд основ среды безопасности Spring Security.

Spring Security

Помнится, мы уже говорили, что OAuth представляет собой протокол делегирования аутентификации и нуждается в чем-то способном ответить на вопрос: «Действительно ли это тот человек, за кого он себя выдает?». Чтобы получить ответ, воспользуемся средой Spring Security. Она интегрируется практически с любым мыслимым типом провайдера идентичности, но даже в противном случае вам не составит особого труда разработать собственный код интеграции.

Аутентификация касается исключительно вопроса о том, *кем* сделан занрос. Это dsyer или jhoeller? В общих чертах аутентификацией в Spring Security занимается реализация интерфейса `AuthenticationManager` (пример 8.17).

Пример 8.17. org.springframework.security.authentication.AuthenticationManager

```
public interface AuthenticationManager {

        Authentication authenticate(Authentication authentication)
                        throws AuthenticationException;
}
```

Экземпляры `AuthenticationManager` возвращают экземпляр `Authentication` со свойством `authenticated`, значение которого установлено в `true`, при условии, что с запрашиваемой аутентификацией все в порядке. Если же запрос терпит неудачу, то они выдают исключение `AuthenticationException` или же, при невозможности принять решение, возвращают значение `null`. На этом уровне не делается никаких предположений насчет провайдера идентичности, в отношении данных которого выполняется запрос аутентификации. Не выстраивается и предположение насчет природы самих запросов — это могут быть HTTP-запросы, вызовы локального метода, вызовы RPC-сервиса, асинхронные сообщения и т. д. Не существует предположений и о природе самой попытки аутентификации; в запросе могут содержаться токены, имена пользователей и пароли, сертификаты и т. д. `AuthenticationManager` характеризуется намеренной неопределенностью.

ProviderManager является общей реализацией AuthenticationManager, который, в свою очередь, делегируется цепочке экземпляров реализации AuthenticationProvider. Деятельность экземпляров AuthenticationProvider практически идентична AuthenticationManager, за исключением того, что они также могут подтвердить свою способность обрабатывать данный Class<?> аутентификации Authentication. Тем самым упрощается эффективная селективная обработка запросов аутентификации. Способ организации ProviderManager означает, что отдельно взятый диспетчер аутентификации AuthenticationManager с помощью делегирования может обрабатывать различные типы аутентификации.

Сами объекты ProviderManager могут иметь родительскую реализацию AuthenticationManager, действующую в качестве альтернативной. Если ни одна реализация AuthenticationProvider не может провести аутентификацию запроса, то за нее принимается родительский диспетчер аутентификации AuthenticationManager, выполняющий запрос на аутентификацию. Возможно наличие глобального AuthenticationManager с последующей вложенностью экземпляров ProviderManager, логически предназначенных для группы защищенных ресурсов. Речь идет об экземплярах ProviderManager, служащих для запросов аутентификации и взаимодействующих с такими сервисами, как LDAP, или с серверной базой данных на основе javax.sql.DataSource. В последнем случае — обращении к хранилищу данных с целью извлечь информацию о пользователе на основе его имени — вполне обычным будет наличие конкретного типа провайдера аутентификации AuthenticationProvider под названием DaoAuthenticationProvider, который, в свою очередь, передает полномочия реализации интерфейса UserDetailsService (пример 8.18).

Пример 8.18. Сервис org.springframework.security.core.userdetails.UserDetails

```
public interface UserDetailsService {

  UserDetails loadUserByUsername(String username)
   throws UsernameNotFoundException;    ❶
}
```

❶ Возвращение реализации UserDetails или выдача исключения. Значение null не должно возвращаться никогда.

Контракт довольно прост: возвратить реализацию объекта UserDetails на основе заданного имени пользователя. Объект UserDetails, предназначенный для инкапсуляции ответов на вопросы Spring Security, будет нуждаться в аутентификации пользователя по его имени и паролю. Какое имя именно у этого пользователя? И какой у него пароль? Активна ли еще учетная запись пользователя, а также не заблокирована ли она или не истек ли срок ее применения? Какими полномочиями обладает данный пользователь? *Полномочие*, в частности, представлено типом GrantedAuthority, в котором, в свою очередь, содержится строка. Значение строки изменяется от системы к системе. Оно может соответствовать понятию полномочий вашей системы, областей видимости, разрешений, ролей или чего-то еще.

Но вы задействуете объект `GrantedAuthority`, предназначенный для поддержки *авторизации*, а не аутентификации.

Авторизация имеет исключительное отношение к выяснению прав доступа, имеющихся у инициатора запроса, то есть к тому, что разрешено или не разрешено пользователю. Здесь основным является интерфейс `AccessDecisionManager`. Он принимает решения по управлению доступом для заданной комбинации объекта аутентификации, коллекции атрибутов конфигурации (экземпляров `ConfigAttribute`) и для контекста принятого параметра `object`. Коллекция `ConfigAttribute` содержит базовые параметры запроса, в качестве которых могут даже выступать инструкции на языке Spring Expression Language. Параметр `object` — контекст, на чьей основе должно приниматься решение о доступе. Пользователю может потребоваться доступ к *чему угодно*: к вызову метода, защищенной конечной точке HTTP, `Message<T>` и т. д. Атрибуты конфигурации и контекст, взятые вместе, предоставляют все, что нужно `AccessDecisionManager` для выполнения авторизации (пример 8.19).

Пример 8.19. org.springframework.security.access.AccessDecisionManager

```
public interface AccessDecisionManager {

  void decide(Authentication authentication, Object context,
    Collection<ConfigAttribute> configAttributes)
      throws AccessDeniedException, InsufficientAuthenticationException; (1)

  boolean supports(ConfigAttribute attribute);
  boolean supports(Class<?> clazz);
}
```

(1) Если запрос не может быть авторизован, то выдается исключение.

В свою очередь, `AccessDecisionManager` передает полномочия экземплярам `DecisionVoter`, что очень похоже на то, как `AuthenticationManager` передает полномочия экземплярам `AuthenticationProvider` (пример 8.20).

Пример 8.20. org.springframework.security.access.AccessDecisionVoter

```
public interface AccessDecisionVoter<S> {

  int ACCESS_GRANTED = 1;
  int ACCESS_ABSTAIN = 0;
  int ACCESS_DENIED = -1;

  boolean supports(ConfigAttribute attribute);

  boolean supports(Class<?> clazz);

  int vote(Authentication authentication,
    S object,
    Collection<ConfigAttribute> attributes);
}
```

Пока все в порядке! Чтобы проанализировать иерархию Spring Security, последние несколько абзацев можно пару раз перечитать. Если усвоить все нюансы еще не удается, то ничего страшного! Из всего этого нужно четко понять только одно: среда Spring Security занимается как *аутентификацией*, так и *авторизацией*, для чего используются две иерархии дискретных и логических типов. Обе иерархии имеют корневой интерфейс, в который включена реализация, в свою очередь передающая полномочия цепочке экземпляров, похожих на корневой интерфейс. Это значит, что обе иерархии поддерживают цепочку ответственности.

Все это находится на самом нижнем уровне. Вам не потребуется знать, не говоря уже об их установке и конфигурировании, о большинстве из указанных типов в сегодняшнем приложении, созданном на основе Spring Boot. Даже в низкоуровневых приложениях Spring Security имеются исходные установки для большинства из упомянутых типов, в том числе без Spring Boot. Так или иначе, при изучении вопросов безопасности веб-приложений будет полезно усвоить то, на фоне чего выполняется их код. На веб-уровне Spring Security устанавливается в виде одного фильтра `javax.servlet.Filter` (в контексте веб-контейнера), в свою очередь передающего полномочия другим *виртуальным* цепочкам экземпляров `Filter` (о которых известно только среде Spring Security). У каждой защищенной конечной веб-точки может иметься собственная цепочка фильтров. Порядок следования этих виртуальных фильтров имеет значение: более конкретные правила должны идти перед более общими, проходными фильтрами. Например, цепочка, реагирующая на запросы для `/api/**`, должна находиться *перед* цепочкой `/**`. Позже, когда обеспечим безопасность частей нашего пограничного сервиса, мы обратим внимание на вопросы ограниченного доступа к частям веб-приложения и укажем на то, как обработать запросы, не прошедшие аутентификацию, а также как выполнять вход и выход для запросов, обращенных к защищенным ресурсам.

Изучив фон, окружающий решение вопросов обеспечения безопасности, вернемся к нашей реализации OAuth. Нам нужно ответить на вопрос о том, *кем делается запрос*. Мы можем сконфигурировать и подключить любой диспетчер аутентификации `AuthenticationManager`, необходимый для выполнения поставленной задачи, но в целях решения вопросов предположим, что у нас имеется база данных из имен пользователей и паролей. Мы можем подключить реализацию сервиса информации о пользователе `UserDetailsService`, в котором для аутентификации имен и паролей применяется JPA (или иная, более подходящая технология). Мы создадим JPA-объект по имени `Account`; его нужно будет задействовать при получении запросов на аутентификацию (пример 8.21).

Как и во многих других главах, мы воспользуемся процессором обработки аннотаций в ходе компиляции Project Lombok (https://projectlombok.org/), чтобы упростить синтез акцессоров, мутаторов, конструкторов и т. д.

Пример 8.21. Account

```
package auth.accounts;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
public class Account {

 @Id
 @GeneratedValue
 private Long id;

 private String username, password; (1)

 private boolean active; (2)

 public Account(String username, String password, boolean active) {
  this.username = username;
  this.password = password;
  this.active = active;
 }
}
```

(1) Ожидаются имя пользователя и пароль.

(2) При реализации нашего сервиса `UserDetailsService` нужно будет ответить на четыре варианта вопроса *об активности учетной записи*.

Быстро разобраться во взаимодействии с записями `Account` в базе данных поможет хранилище Spring Data (пример 8.22).

Пример 8.22. AccountRepository

```
package auth.accounts;

import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface AccountRepository extends JpaRepository<Account, Long> {

 (1)
 Optional<Account> findByUsername(String username);
}
```

❶ При реализации экземпляра `UserDetailsService` мы должны иметь возможность отвечать на запрос с именем пользователя, поэтому разрешим хранилищу Spring Data выяснить его.

И наконец, мы предоставляем реализацию `UserDetailsService` (пример 8.23). Среда Spring Security автоматически подключит ее для нас, если обнаружит реализацию в Spring `ApplicationContext`. Эта реализация отображает `Optional<Account>` из нашего хранилища на конкретную реализацию `UserDetails` (названную в Spring Security `User`) и присваивает значения некоторым статическим экземплярам `GrantedAuthority`.

В более сложном примере экземпляры GrantedAuthority можно динамически определять на основе заданных критериев или же (если нет ничего другого) считывая критерии из базы данных.

Пример 8.23. AccountConfiguration.java — в данном случае мы предоставляем экземпляр UserDetailsService, который знает о наших типах Account

```
package auth.accounts;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;

@Configuration
public class AccountConfiguration {

 @Bean
 UserDetailsService userDetailsService(AccountRepository accountRepository) {
  ❶
  return username -> accountRepository
   .findByUsername(username)
   .map(
    account -> {
     boolean active = account.isActive();
     return new User(account.getUsername(), account.getPassword(), active,
      active, active, active, AuthorityUtils.createAuthorityList("ROLE_ADMIN",
       "ROLE_USER"));
    })
   .orElseThrow(
    () -> new UsernameNotFoundException(String.format("username %s not
      found!",
     username)));
 }
}
```

❶ Контракт для реализации `UserDetailsService` довольно прост: исходя из имени пользователя String, возвратить реализацию `UserDetails` или выдать исключение `UsernameNotFoundException`. Но ни в коем случае не следует возвращать `null`.

Проделав все это, мы просто настроили все, что требуется для работы со средой Spring Security. Наша цель заключается в последующем применении данной конфигурации для проверки токенов от входящих пользователей — но данная конфигурация составляет основную часть того необходимого, что нужно было бы аутентифицировать с помощью HTTP BASIC или из формы входа на какой-нибудь HTML-странице. Это основа Spring Security, и подобный код вы бы написали еще десять и более лет назад, до начала облачной эры.

Spring Cloud Security

Среда Spring Cloud упрощает интеграцию безопасности в наши микросервисы. Она может обращаться к любому сервису авторизации, совместимому с OAuth, и декларативно защищать серверы ресурсов. Мы заблокируем собственные REST API, но сначала запустим сервер авторизации OAuth, воспользовавшись Spring Security OAuth.

Сервер авторизации Spring Security OAuth

Создадим новый микросервис, занимающийся авторизацией в модуле по имени `auth-service`. Этот модуль в пути к классам имеет записи `org.springframework.cloud: spring-cloud-starter-config`, `org.springframework.cloud : spring-cloud-starter-eureka`, `org.springframework.cloud : spring-cloud-starter-oauth2`, `org.springframework.boot : spring-boot-starter-data-jpa`, `org.springframework.boot : spring-boot-starter-web` и `org.springframework.boot : spring-boot-starter-actuator`. Данный сервис авторизации работает с реестром сервисов (нашим экземпляром Netflix Eureka) и поэтому участвует в регистрации и обнаружении сервисов, для чего использует аннотацию `@EnableDiscoveryClient`.

Технологии, подобные Spring Cloud Security Authorization Server, играют важную роль в архитектуре. В данном разделе мы рассмотрим создание собственной упрощенной версии, и притом важно понять, что ее функции ничего не меняют в бизнес-логике. В идеале этим должны заниматься другие специалисты. Сервер аутентификации пользовательской учетной записи (User Account Authentication (UAA) Server), основанный на рассматриваемом здесь сервере Spring Cloud Security Authorization Server, могут предоставить сторонние провайдеры сред идентификации, совместимых с OAuth, такие как Okta или Cloud Foundry. Соблюдение мер безопасности дается нелегко, поэтому повторно применять их нужно везде, где только возможно!

Протоколу авторизации OAuth необходимо знать, как именно выполняется аутентификация пользователей, с которой мы справляемся путем интеграции `UserDetailsService`. Кроме того, ему нужно знать о клиентах, способных подключаться, и о тех полномочиях, которыми обладают эти клиенты. Описания последних должны быть созданы заранее. Точно так же, как и при проведении моделирования пользователей с объектом `Account`, мы применим объект `Client` и адаптируем его под клиентов, ожидаемых средой Spring Security OAuth (пример 8.24).

Пример 8.24. Client.java

```
package auth.clients;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.util.StringUtils;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity

public class Client {

 @Id
 @GeneratedValue
 private Long id;

 private String clientId;

 private String secret;

 private String scopes = StringUtils
  .arrayToCommaDelimitedString(new String[] { "openid" });

 private String authorizedGrantTypes = StringUtils
  .arrayToCommaDelimitedString(new String[] { "authorization_code",
  "refresh_token", "password" });

 private String authorities = StringUtils
  .arrayToCommaDelimitedString(new String[] { "ROLE_USER", "ROLE_ADMIN" });

 private String autoApproveScopes = StringUtils
  .arrayToCommaDelimitedString(new String[] { ".*" });

 public Client(String clientId, String clientSecret) {
```

```
  this.clientId = clientId;
  this.secret = clientSecret;
 }
}
```

Чуть позже нам понадобится возможность поиска клиента по его идентификатору `clientId`, и для этого послужит хранилище Spring Data JPA наряду с настраиваемым методом поиска (пример 8.25).

Пример 8.25. ClientRepository.java

```
package auth.clients;

import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface ClientRepository extends JpaRepository<Client, Long> {

 Optional<Client> findByClientId(String clientId);  ❶
}
```

❶ Нахождение клиента по его `clientId`.

Многие популярные сервисы предлагают возможность регистрации специальных клиентов, которые затем могут использовать свои API. К примеру, Facebook и Twitter позволяют регистрировать сторонних клиентов на своих порталах разработчиков. Вы можете запросто создать собственный клиент, применив только что установленное нами хранилище Spring Data. Фактически если вы проаннотировали хранилище Spring Data для использования Spring Data REST, то можете даже разрешить программную регистрацию клиентов на REST-основе. Во всяком случае предоставить форму самообслуживания, позволяющую командам в вашей организации регистрировать собственных клиентов и ожидаемые типы разрешений, было бы нетрудно. Но для нас целесообразнее будет указать данные клиента в готовом виде.

Среда Spring Security OAuth также требует небольшой настройки. Ей нужно знать, как она станет проводить аутентификацию пользователей, поэтому мы должны направить ее на работоспособный экземпляр `AuthenticationManager` (что будет успешно сделано в автоматическом режиме при установке нашего пользовательского сервиса `UserDetailsService`) и на сервис `ClientDetailsService`, и обе эти задачи мы выполним в `AuthorizationServerConfiguration`.

Наши экземпляры `Client` должны быть адаптированы под интерфейс `ClientDetails`, как мы делали в отношении объектов `Account` для интерфейса `UserDetailsService`. Чтобы выполнить эту задачу, в примере 8.26 воспользуемся особым типом `ClientDetails` под названием `BaseClientDetails`.

Пример 8.26. ClientConfiguration.java

```
package auth.clients;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.cloud.client.discovery.event.HeartbeatEvent;
import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.event.EventListener;
import org.springframework.security.oauth2.provider.ClientDetailsService;
import org.springframework.security.oauth2.provider.
ClientRegistrationException;
import org.springframework.security.oauth2.provider.client.BaseClientDetails;

import java.util.Collections;
import java.util.List;
import java.util.Optional;
import java.util.Set;
import java.util.concurrent.ConcurrentSkipListSet;
import java.util.stream.Collectors;

@Configuration
public class ClientConfiguration {

 private final LoadBalancerClient loadBalancerClient;

 @Autowired
 public ClientConfiguration(LoadBalancerClient client) {
  this.loadBalancerClient = client;
 }

 @Bean
 ClientDetailsService clientDetailsService(ClientRepository clientRepository)
{
  return clientId -> clientRepository
   .findByClientId(clientId)
   .map(
    client -> {

     BaseClientDetails details = new BaseClientDetails(client.getClientId(),
      null, client.getScopes(), client.getAuthorizedGrantTypes(), client
       .getAuthorities());
     details.setClientSecret(client.getSecret());

     ❶
     // details.setAutoApproveScopes
     //    (Arrays.asList(client.getAutoApproveScopes().split(",")));

     ❷
     String greetingsClientRedirectUri = Optional
      .ofNullable(this.loadBalancerClient.choose("greetings-client"))
```

```
      .map(si -> "http://" + si.getHost() + ':' + si.getPort() + '/')
      .orElseThrow(
       () -> new ClientRegistrationException(
        "couldn't find and bind a greetings-client IP"));

     details.setRegisteredRedirectUri(Collections
      .singleton(greetingsClientRedirectUri));
     return details;
    })
   .orElseThrow(
    () -> new ClientRegistrationException(String.format(
     "no client %s registered", clientId)));
 }
}
```

❶ Подумайте о варианте использования Facebook.com. Когда клиент запрашивает токен, ему предлагают как зарегистрироваться, так и предоставить в явном виде все запрашиваемые области видимости. Вместо этого мы можем настроить все так, чтобы все запрошенные области видимости уже предоставлялись путем аутентификации, если настроили `autoApproveScopes` на имена запрошенных областей видимости.

❷ Клиент должен указать URI, на который следует перенаправить сервер авторизации после того, как пользователь санкционировал допуск к запрошенным областям видимости. В данном конкретном случае мы можем автоматически определять, что URI применяет реестр сервисов, а также выполнять балансировку нагрузки с помощью `LoadBalancerClient`, поскольку все клиенты задействуют один и тот же реестр сервисов. В этом примере, основываясь на уже известных нам будущих потребностях данного клиента, мы жестко задали URI, хотя ничто не препятствует тому, чтобы он также содержался в самой JPA-записи клиента.

`AuthorizationServerConfiguration` расширяет `AuthorizationServerConfigurerAdapter`, в свою очередь реализующий `AuthorizationServerConfigurer`. Данный `AuthorizationServerConfigurer` представляет собой интерфейс обратного вызова, который среда Spring Security OAuth будет вызывать в нужной фазе жизненного цикла инициализации, позволяя конфигурирование различных сторон поведения сервиса. Чтобы подключить среду Spring Security OAuth к ранее настроенному диспетчеру аутентификации `AuthenticationManager`, переопределите метод `configure(ClientDetailsServiceConfigurer clients)`. В этом месте среда, управляющая авторизацией, делегирует аутентификацию среде Spring Security (пример 8.27).

Пример 8.27. AuthorizationServerConfiguration.java

```
package auth;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.security.authentication.AuthenticationManager;

//@formatter:off
import org.springframework.security.oauth2
      .config.annotation.configurers.ClientDetailsServiceConfigurer;
import org.springframework.security.oauth2
      .config.annotation.web.configuration.AuthorizationServerConfigurerAdapter;
import org.springframework.security.oauth2
      .config.annotation.web.configuration.EnableAuthorizationServer;
import org.springframework.security.oauth2
      .config.annotation.web.configurers.AuthorizationServerEndpointsConfigurer;
import org.springframework.security.oauth2
      .provider.ClientDetailsService;
//@formatter:on

@Configuration
@EnableAuthorizationServer
class AuthorizationServerConfiguration extends
 AuthorizationServerConfigurerAdapter {

 private final AuthenticationManager authenticationManager;

 private final ClientDetailsService clientDetailsService;

 @Autowired
 public AuthorizationServerConfiguration(
  AuthenticationManager authenticationManager,
  ClientDetailsService clientDetailsService) {
  this.authenticationManager = authenticationManager;
  this.clientDetailsService = clientDetailsService;
 }

 @Override
 public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

  ❶
  clients.withClientDetails(this.clientDetailsService);
 }

 @Override
 public void configure(AuthorizationServerEndpointsConfigurer endpoints)
  throws Exception {
  ❷
  endpoints.authenticationManager(this.authenticationManager);
 }
}
```

❶ Конфигурирование клиентов OAuth.

❷ Подключение Spring Security OAuth к Spring Security с помощью его экземпляра диспетчера аутентификации `AuthenticationManager`.

Защита сервера ресурсов приветствий

Мы зашли уже достаточно далеко, поэтому не пора ли применить полученные знания на практике? Предположим, что у нас есть REST API, требующий защиты: нужно отклонять запросы, не содержащие в себе действующий токен доступа. Сервер ресурсов можно защитить, снабдив его аннотацией `@EnableResourceServer`. При такой конфигурации среда Spring Security OAuth станет отклонять подобные запросы. Если у запроса есть действующий токен доступа, требуется некий способ превратить этот токен в аутентификацию Spring Security `Authentication`. Сам токен не несет в себе никакого смысла. Его необходимо преобразовать, чтобы он связался с пользователем, аутентифицированным в системе. Мы можем указать защищенному серверу ресурсов на конечную точку информации о пользователе, где среда Spring Security OAuth обменяет токен на сведения о нем: для этого на сервере ресурсов `greetings-service` следует указать `security.oauth2.resource.userInfoUri=http://auth-service/uaa/user`.

В этом URL надлежащее имя хоста не задействуется, вместо него применяется идентификатор сервиса, разрешаемый в реестре сервисов. Мы извлекли шаблон `RestTemplate`, осведомленный об использовании реестра сервисов (service-registry-aware) и, кроме того, о применении OAuth (OAuth-aware), в совместно используемый класс автоматического конфигурирования в отдельном модуле `security-autoconfiguration`. В библиотеке содержится просто класс автоматического конфигурирования по имени `TokenRelayAutoConfiguration`, в котором ведется настройка шаблона `RestTemplate`, осведомленного о балансировке нагрузки на стороне клиента (client-side load-balancer-aware). Конфигурация показана в примере 8.28.

Пример 8.28. TokenRelayAutoConfiguration.java

```
package relay;

import feign.RequestInterceptor;

//@formatter:on
import org.springframework.boot.autoconfigure
        .condition.ConditionalOnBean;
import org.springframework.boot.autoconfigure
        .condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure
        .condition.ConditionalOnWebApplication;
import org.springframework.boot.autoconfigure
        .security.oauth2.resource.UserInfoRestTemplateFactory;
//@formatter:off

import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.context.annotation.Lazy;
import org.springframework.context.annotation.Profile;
import org.springframework.http.HttpHeaders;

//@formatter:on
import org.springframework.security
        .oauth2.client.OAuth2ClientContext;
import org.springframework.security
        .oauth2.client.OAuth2RestTemplate;
import org.springframework.security
        .oauth2.client.filter.OAuth2ClientContextFilter;
import org.springframework.security
        .oauth2.config.annotation.web.configuration.EnableResourceServer;
import org.springframework.web.client.RestTemplate;
//@formatter:off

@Configuration
@ConditionalOnWebApplication
@ConditionalOnClass(EnableResourceServer.class)
public class TokenRelayAutoConfiguration {

 public static final String SECURE_PROFILE = "secure";
 @Configuration
 @Profile("!" + SECURE_PROFILE)
 public static class RestTemplateConfiguration {

  ❶
  @Bean
  @LoadBalanced
  RestTemplate simpleRestTemplate() {
   return new RestTemplate();
  }
 }

 @Configuration
 @Profile(SECURE_PROFILE)
 public static class SecureRestTemplateConfiguration {

  ❷
  @Bean
  @Lazy
  @LoadBalanced
  OAuth2RestTemplate anOAuth2RestTemplate( UserInfoRestTemplateFactory factory) {
   return factory.getUserInfoRestTemplate();
  }
 }

 @Configuration
 @Profile(SECURE_PROFILE)
 @ConditionalOnClass(RequestInterceptor.class)
 @ConditionalOnBean(OAuth2ClientContextFilter.class)
```

```
 public static class FeignAutoConfiguration {

  ❸
  @Bean
  RequestInterceptor requestInterceptor(OAuth2ClientContext clientContext )
  return requestTemplate -> requestTemplate.header(HttpHeaders.AUTHORIZATION,
    clientContext.getAccessToken().getTokenType() + ' '
     + clientContext.getAccessToken().getValue());
  }
 }
}
```

❶ Установка обычного балансирующего нагрузку шаблона `RestTemplate` в небезопасном профиле.

❷ Установка балансирующего нагрузку шаблона `RestTemplate`, распространяющего также токен OAuth (если таковой имеется) в безопасном профиле.

❸ Распространение токенов доступа OAuth через вызовы Feign. Эту технологию мы рассмотрим чуть позже.

И наконец, нужно определить конечную точку информации о пользователе, `/uaa/user`. Когда запрос делается в отношении одного из наших защищенных REST-сервисов, фильтр, настроенный средой Spring Cloud Security, видит токен доступа в поступившем запросе, а затем преобразует этот токен в информацию о клиенте и пользователе, отправившем запрос. Что-либо понять из токена невозможно — в нем самом нет никаких сведений, и его следует обменять на полезную информацию. Разумеется, основная ее часть от одного сервера авторизации к другому будет изменяться. В Facebook в ней могут содержаться данные о пользователе и о графе его друзей, в GitHub — данные о пользователе GitHub и его обязательствах. Многие из этих конечных точек предоставляют как минимум JSON-атрибут по имени `name`.

Если мы настраиваем наш сервер авторизации на то, чтобы он выполнял еще и роль сервера ресурсов, добавляя к нему для этого аннотацию `@EnableResourceServer`, то среда Spring Security OAuth автоматически предоставляет `java.security.Principal` для любого обрабатывающего метода Spring MVC, разбирающего запрос, содержащий токен доступа. Среда Spring Security OAuth также выполнит преобразование `java.security.Principal` в JSON-структуру, которой сможет воспользоваться наш сервер ресурсов.

Внесем изменения в сервер авторизации и создадим конечную точку для обмена токенов на `Principal` (пример 8.29).

Пример 8.29. PrincipalRestController.java

```
package auth;

import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.bind.annotation.RestController;

import java.security.Principal;

@RestController
class PrincipalRestController {

 ❶
 @RequestMapping("/user")
 Principal principal(Principal p) {
  return p;
 }
}
```

❶ При поступлении запроса, несущего токен, запрашивающему возвращается специально обработанный объект `Principal`.

Наш сервер авторизации будет начинаться с пути к контексту (`context-path`) `/uaa` и станет запускаться на порте 9191 (пример 8.30).

Пример 8.30. bootstrap.properties

```
spring.application.name=auth-service
❶
server.context-path=/uaa
security.sessions=if_required
logging.level.org.springframework.security=DEBUG
spring.jpa.hibernate.ddl-auto=create
spring.jpa.generate-ddl=true
```

❶ Элемент `spring.application.name` определяет приложение в реестре сервисов.

Теперь, имея все это, мы получаем возможность создания токена доступа. Наш тест показывает, как создавать запрос к серверу авторизации, чтобы получить действующий токен доступа, задействуя предоставление пароля. В нем мы обмениваем имя пользователя и пароль на токен доступа. Это самый простой способ доказать работоспособность нашего сервера авторизации OAuth.

В примере 8.31 показывается, как с помощью команды `curl` протестировать работоспособность всех компонентов.

Пример 8.31. Получение токена доступа с использованием типа предоставления пароля и команды curl

```
curl \
  -X POST \
  -H"authorization: Basic aHRtbDU6cGFzc3dvcmQ=" \
  -F"password=spring" \
  -F"client_secret=password" \
  -F"client_id=html5" \
  -F"username=jlong" \
  -F"grant_type=password" \
  -F"scope=openid" \
  http://localhost:9191/uaa/oauth/token
```

При запуске данного вызова команды `curl` и перенаправления результатов по конвейеру на JSON pretty printer (`json_pp`) мы получаем следующий ответ (пример 8.32).

Пример 8.32. Ответ в формате JSON, возвращенный сервером авторизации OAuth2

```
{
  "scope" : "openid",
  "expires_in" : 40222,
  "token_type" : "bearer",
  "refresh_token" : "12164df0-12a6-43d9-b631-8418aec28612",
  "access_token" : "6815d559-784c-496a-b50e-b8c91eb17ffd"
}
```

Компонент `access_token` играет весьма важную роль. Располагая этим компонентом, мы можем попытаться вызвать защищенный сервер ресурсов. Закроем наш сервер ресурсов и станем вести обмен данными с ним, используя только что созданный токен доступа.

Если у нас есть возможность создать действующий токен доступа на основе идентификатора клиента, его секретного слова, имени пользователя и пароля, то мы получим все необходимое для создания клиента JavaScript, имеющего доступ к данным из пограничного сервиса, который, в свою очередь, будет вести обмен данными с сервисом приветствий `greetings-service`. Первым шагом станет подтверждение того, что мы закрыли пограничный сервис (`greetings-service`). Вся конфигурация кодовой базы пограничного сервиса показана в примере 8.33. Она будет работать только при запуске пограничного сервиса с активным профилем защиты.

Пример 8.33. OAuthResourceConfiguration.java

```
package greetings;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
//@formatter:off
import org.springframework.security.oauth2.config.annotation
        .web.configuration.EnableOAuth2Client;
import org.springframework.security.oauth2.config.annotation
        .web.configuration.EnableResourceServer;
//@formatter:on

@Configuration
❶
@Profile("secure")
❷
@EnableResourceServer
❸
@EnableOAuth2Client
class OAuthResourceConfiguration {
}
```

❶ Конфигурация активна, только если сервис запущен с профилем `secure`.

❷ Аннотация `@EnableResourceServer` будет приводить к отклонениям запросов, не прошедших аутентификацию.

❸ Аннотация `@EnableOAuth2Client` приводит к сохранению любого видимого в запросе токена, позволяя узлу действовать в качестве клиента по отношению к другому защищенному узлу.

Аутентифицированный объект `java.security.Principal` можно вставить в ваши методы контроллера Spring MVC. Далее показана переделка ранее созданной конечной точки приветствий для доработки результатов на выходе с помощью аутентифицированного субъекта (пример 8.34). Ее существование обусловлено исключительно фактом запуска `greetings-service` под Spring-профилем `secure`. Если `greetings-service` еще не запущен, то его следует перезапустить, добавив этот профиль.

Пример 8.34. SecureGreetingsRestController.java

```
package greetings;

import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.security.Principal;
import java.util.Collections;
import java.util.Map;

@Profile("secure")
@RestController
@RequestMapping(method = RequestMethod.GET, value = "/greet/{name}")
public class SecureGreetingsRestController {

 @RequestMapping
 Map<String, String> hi(@PathVariable String name, Principal p) {
  return Collections.singletonMap("greeting",
    "Hello, " + name + " from " + p.getName() + "!");
 }
}
```

Вы можете получить только что созданный токен доступа и отправить запрос сервису приветствий `greetings-service` (пример 8.35).

Пример 8.35. Получение защищенного результата с помощью команды curl

```
curl -H"authorization: bearer ..."
{"greeting":"Hello, Tammie from jlong!"}
```

После того как вы сделали запрос и увидели, что он работает, попробуйте отправить запрос еще раз, но на этот раз измените значение токена (можете удалить букву или изменить несколько символов) и убедитесь в том, что запрос будет отклонен. Затем попробуйте еще раз отправить точно такой же запрос, не указывая заголовок авторизации (удалите ключ `-H` и аргумент), чтобы подтвердить отклонение запроса.

Итак, мы убедились в возможности закрытия доступа к конкретному REST API и разрешения запросов с действующим токеном доступа. И все-таки откуда берется токен доступа? Вы же не думаете, что посетители сайта для получения этого токена станут извлекать на свет команду `curl`? Нам нужно предоставить пользователю способ создания токена.

Создание одностраничного приложения, защищенного OAuth

Вернемся к пограничному сервису. Мы уже подошли к серверу авторизации и защите нашего REST API сервиса приветствий `greetings-service`, отклоняя запросы, которые не содержат действующий токен доступа, используя для этого аннотацию `@EnableResourceServer`. Чтобы продемонстрировать защищенность в действии, мы сделали запрос к серверу авторизации для получения токена доступа в обмен на имя пользователя и пароль, задействуя предоставление пароля. Понятно, что наши пользователи для входа на сайт не собираются отправлять HTTP-запросы, применяя в командной строке сервера авторизации инструкцию `curl`. Нам необходимы подключение к потоку входа и UI на стороне клиента к Spring Security OAuth. В момент посещения веб-страницы с JavaScript на стороне клиента пользователь будет перенаправлен на сервис авторизации `auth-service` и ему предложат войти и дополнительно санкционировать отправку определенных запросов. После подтверждения у пограничного сервиса будет токен доступа. Его он сможет распространить во всех запросах ко всем нижестоящим сервисам, которые, если обладают защитой наподобие той, что есть у `greetings-service`, могут воспользоваться им для идентификации того, кто делает запрос и в отношении какого клиента сделан запрос. Это называется *единым входом* в Spring Cloud Security.

Рассмотрим простое приложение на стороне клиента, написанное с помощью Angular.js, которое отображает некое состояние из защищенного REST API. Мы будем обеспечивать аутентификацию с применением подразумеваемого предоставления, которое затем приводит к перенаправлению на сервер авторизации с выдачей приглашения на ввод пользовательского имени и пароля, и санкционирования доступа к запрошенным областям видимости. После этого мы будем возвращены к UI клиента. В данном примере мы увидим, как Spring Cloud Security превращает все описанное в слаженный процесс.

У нас уже есть пограничный сервис, имеющий несколько конечных точек, от которых хотелось бы, чтобы они были защищены, но мы также собираемся добавить

к каталогу `src/main/resources/static` файл `index.html` и некий код JavaScript. Следует обеспечить этим ресурсам повсеместную видимость. Нам нужно защитить только REST API в подкаталогах `/api/*` (установленные ранее и управляемые `RestTemplate` или Netflix Feign). Чтобы запросить действующий токен доступа путем добавления `@EnableResourceServer`, а затем сконфигурировать приложение, мы предоставим адаптер конфигурации сервера ресурсов `ResourceServerConfigurerAdapter` (пример 8.36).

Пример 8.36. SecureResourceConfiguration.java

```
package greetings;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
//@formatter:off
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.oauth2.config.annotation.web
        .configuration.EnableResourceServer;
import org.springframework.security.oauth2.config.annotation.web
        .configuration.ResourceServerConfigurerAdapter;
//@formatter:on

❶
@Profile("secure")
@Configuration
@EnableResourceServer
class SecureResourceConfiguration extends ResourceServerConfigurerAdapter {

 @Override
 public void configure(HttpSecurity http) throws Exception {
  http.antMatcher("/api/**").authorizeRequests() ❷
   .anyRequest().authenticated();
 }
}
```

❶ Эта конфигурация станет работать только при условии активности профиля `secure`.

❷ Защищенными будут только запросы, направляемые к `/api/*`. Все остальное останется без защиты.

Для клиента нужно настроить функциональные возможности единого входа и выхода. Приложение будет предоставлять конечную точку `/login`, фрагменты кода JavaScript и главную страницу `/index.html`, которая загрузится, если кто-нибудь перейдет по адресу `/`. В пограничном сервисе мы не станем просто отклонять запросы, не имеющие токена доступа, а собираемся инициировать получение токена доступа путем перенаправления на экран входа. Для настройки локальных конечных точек (наподобие `/login`), которые станут запускать «танцы» аутентификации между пограничным сервисом и сервером аутентификации, имеющим аннотацию

`@EnableOAuth2Sso`, мы можем установить фильтр аутентификации и точку входа аутентификации. Рассмотрим SSO-конфигурацию и укажем те конечные точки, которые должны инициировать поток единого входа (SSO-поток), используя адаптер `WebSecurityConfigurerAdapter` (пример 8.37).

Пример 8.37. SsoConfiguration.java

```
package greetings;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

//@formatter:off
import org.springframework.boot.autoconfigure
        .security.oauth2.client.EnableOAuth2Sso;
import org.springframework.security.config.annotation
        .web.builders.HttpSecurity;
import org.springframework.security.config.annotation
        .web.configuration.WebSecurityConfigurerAdapter;
//@formatter:on

import org.springframework.security.web.csrf.CookieCsrfTokenRepository;

❶
@Profile("sso")
@Configuration
❷
@EnableOAuth2Sso
class SsoConfiguration extends WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
  // @formatter:off
        http.antMatcher("/**").authorizeRequests()  ❸
                .antMatchers( "/", "/app.js", "/login**", "/webjars/**
                 ").permitAll().anyRequest()
                .authenticated().and().logout().logoutSuccessUrl("/").permitAll()
                    .and().csrf()
                .csrfTokenRepository(CookieCsrfTokenRepository.
                    withHttpOnlyFalse());
    // @formatter:on
 }
}
```

❶ Эта конфигурация будет работать только при условии активности профиля `sso`.

❷ Аннотация `@EnableOAuth2Sso` приводит к регистрации всех необходимых механизмов инициирования SSO-потока.

❸ Нам нужно защитить все ресурсы, *за исключением* следующих конкретных ресурсов.

Наш пограничный сервис является OAuth-клиентом. Он получит пользовательский контекст, заставив пользователя выполнить вход, но ему нужно пройти самоидентификацию в качестве конкретного клиента. Указать, каким клиентом он будет, можно, в частности, в конфигурации (пример 8.38).

Пример 8.38. bootstrap-sso.properties

```
security.oauth2.client.client-id=html5 ❶
security.oauth2.client.client-secret=password


security.oauth2.client.access-token-uri=http://localhost:9191/uaa/oauth/token ❷
security.oauth2.client.user-authorization-uri=\
  http://localhost:9191/uaa/oauth/authorize

security.basic.enabled=false
```

❶ Определение клиентского идентификатора и секретного слова, применяемых нами для получения пользовательского контекста.

❷ Направление нашего пограничного сервиса на сервер авторизации OAuth.

И это все, что нужно было сделать для получения SSO-потока, работающего на серверной стороне на пограничном сервисе. Напишем Angular.js-контроллер на стороне клиента, который попытается считать защищенную информацию и выдаст приглашение на аутентификацию, если мы ее еще не прошли. В примере 8.39 показан HTML-шаблон для Angular.js-приложения и его (и только его) контроллер `home`.

Пример 8.39. index.html

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8"/>
    <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
    <title>Edge Service</title>
    <meta name="description" content=""/>
    <meta name="viewport" content="width=device-width"/>
    <base href="/"/>
    <script type="text/javascript"
            src="/webjars/jquery/jquery.min.js"></script>
    <script type="text/javascript"
            src="/webjars/bootstrap/js/bootstrap.min.js"></script>
    <script type="text/javascript"
            src="/webjars/angularjs/angular.min.js"></script>
</head>

<body ng-app="app" ng-controller="home as home">

<div class="container" ng-show="!home.authenticated">
```

```
    <a href="/login">Login </a>
</div>

<div class="container" ng-show="home.authenticated">

    ❶
    Logged in as:
    <b><span ng-bind="home.user"></span></b> <br/>

    Token:
    <b><span ng-bind="home.token"></span> </b><br/>

    Greeting from Zuul Route: <b>
    <span ng-bind="home.greetingFromZuulRoute"></span></b> <br/>

    Greeting from Edge Service (Feign):
    <b><span ng-bind="home.greetingFromEdgeService"></span></b><br/>
</div>

❷
<script type="text/javascript" src="app.js"></script>
</body>
</html>
```

❶ Отображение аутентификационной информации и защищенных данных будет выполнено с помощью установки контакта с защищенной конечной точкой REST на том же самом узле.

❷ Логика JavaScript находится в другом файле по имени `app.js`.

В шаблоне, в свою очередь, содержится панель, выводящая информацию, полученную и связанную в логике JavaScript. Суть приложения JavaScript заключается в вызове конечной точки `/user`, находящейся в локальном узле. Если у запроса есть токен доступа, то он будет успешно выполнен, а при отсутствии такового он инициирует поток аутентификации.

Конечная точка `/user` просто выводит дамп информации о прошедшем аутентификацию пользователе, идентично тому, что мы сделали ранее в `auth-service` с конечной точкой `/uaa/user`. У пограничного сервиса `edge-service` имеется аннотация `@EnableResourceServer`, поэтому при обращении к конечной точке `/user` возвращается доверитель для применения в приложении JavaScript. Код для конечной точки `/user` в пограничном сервисе (но мы простим мысль о том, что этот код взят из `auth-service`!) показан в примере 8.40.

Пример 8.40. PrincipalRestController

```
package greetings;

import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
import java.security.Principal;

@Profile("secure")
@RestController
class PrincipalRestController {

 @RequestMapping("/user")
 public Principal user(Principal principal) {
  return principal;
 }
}
```

Посмотрим на логику нашего приложения Angular.js (JavaScript) (пример 8.41).

Пример 8.41. app.js

```
var app = angular.module("app", []);

❶
app.factory('oauth', function () {
    return {details: null, name: null, token: null};
});

app.run(['$http', '$rootScope', 'oauth', function ($http, $rootScope, oauth) {

    $http.get("/user").success(function (data) {

        oauth.details = data.userAuthentication.details;
        oauth.name = oauth.details.name;
        oauth.token = data.details.tokenValue;

        ❷
        $http.defaults.headers.common['Authorization'] = 'bearer ' + oauth.token;

        ❸
        $rootScope.$broadcast('auth-event', oauth.token);
    });
}]);

app.controller("home", function ($http, $rootScope, oauth) {

    var self = this;

    self.authenticated = false;

    ❹
    $rootScope.$on('auth-event', function (evt, ctx) {
        self.user = oauth.details.name;
        self.token = oauth.token;
        self.authenticated = true;
```

```
        var name = window.prompt('who would you like to greet?');

        ❺
        $http.get('/greetings-service/greet/' + name)
            .success(function (greetingData) {
                self.greetingFromZuulRoute = greetingData.greeting;
            })
            .error(function (e) {
                console.log('oops!' + JSON.stringify(e));
            });

        ❻
        $http.get('/lets/greet/' + name)
            .success(function (greetingData) {
                self.greetingFromEdgeService = greetingData.greeting;
            })
            .error(function (e) {
                console.log('oops!' + JSON.stringify(e));
            });
    });
});
```

❶ Чтобы это приложение заработало, понадобится синглтон-объект для хранения данных аутентификации.

❷ Код в `app.run` вызывается сразу после загрузки приложения; это делает данный файл идеальным местом для выполнения любых действий, касающихся инициализации. Чтобы считать информацию по аутентификации, нужно вызвать защищенную конечную точку `/user`. Неаутентифицированный запрос вызовет перенаправление на `auth-service`, где потребуется выполнить вход в приложение и согласиться с предоставлением разрешения. Сервис авторизации выполнит обратное перенаправление на прежнюю страницу, и этот вызов завершится успешно, позволяя установить исходное значение для заголовка `Authorization` для всех Ajax-вызовов, выполняемых через имеющегося в библиотеке Angular.js клиента `$http`.

❸ Публикация события для всех компонентов, заинтересованных новостями о получении нашей информации об аутентификации.

❹ В этом Angular.js-приложении имеется один контроллер `home`, связывающий четыре поля разметки в нашем шаблоне, представленном в файле `index.html`.

❺ Для подтверждения того, что все работает, как и ожидалось, Angular.js-клиент вызывает конечные точки в пограничном сервисе через прокси-сервер Zuul и с помощью...

❻ ...API, применяющего инструменты библиотеки Feign.

Если запустить `service-registry`, `auth-service`, `greetings-service` (с профилем `secure`) и `edge-service` (с профилями `secure` и `sso`), то появится возможность зайти

в пограничный сервис и быть перенаправленным на `auth-service`. Посмотрим, как выглядит эта последовательность.

1. Зайдем на пограничный сервис 8082, и тут же нас перенаправят на `auth-service` (рис. 8.3).

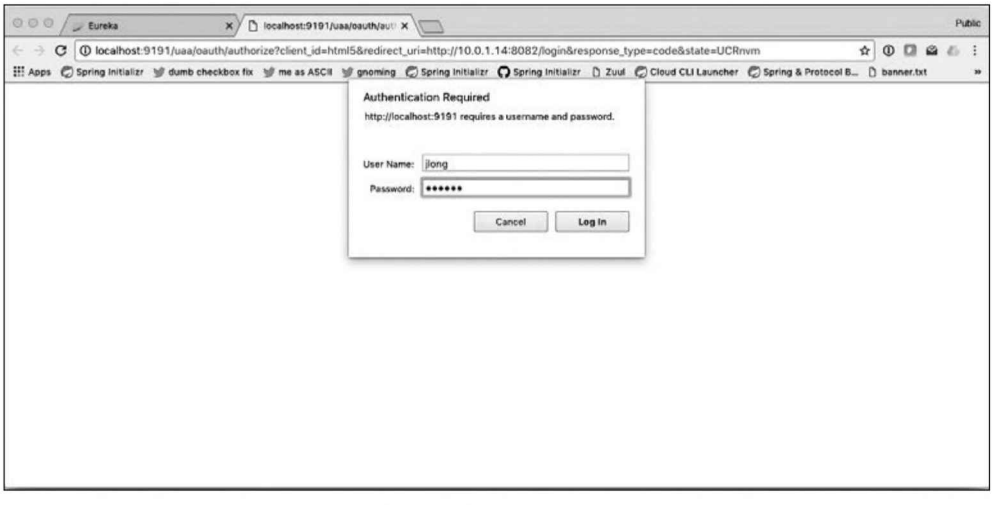


Рис. 8.3. Посещение пограничного сервиса

2. Здесь, возможно, придется разрешать запрашиваемые области видимости (рис. 8.4).

Рис. 8.4. Разрешение запрашиваемых областей видимости

3. Затем нас перенаправят обратно в приложение, где появится возможность совершить повторную попытку (рис. 8.5). Все получится, так как `edge-service`

будет иметь действующий токен доступа и предложит предоставить имя для отправки конечным точкам сервиса `edge-service` (который, в свою очередь, вызовет `greetings-service`).

Рис. 8.5. Маршруты Zuul

4. После чего, конечно же, будут выведены результаты, завершающие запрос, инициированный в `edge-service`, отправленный на `auth-service`, затем перенаправленный на `edge-service`, который, в свою очередь, вызывает `greetings-service` (рис. 8.6).

Рис. 8.6. Результаты

Поддержка технологии единого входа (SSO), включенная в среду Spring Cloud Security, одновременно защищает ресурсы, имеющиеся в пограничном сервисе, и устанавливает механизм, необходимый для настройки шаблона `OAuthRestTemplate`, способного выполнять безопасные вызовы в адрес нижестоящих сервисов (в данном случае в адрес `greetings-service`). Пограничный сервис `edge-service` одновременно выступает в роли как защищенного ресурса, так и клиента последнего. Пограничный сервис `edge-service`, как и `greetings-service`, содержит в путях к классам ранее рассмотренный нами модуль `edge-security-autoconfiguration`.

Резюме

В данной главе мы изучили ряд разносторонних понятий, необходимых для создания эффективных пограничных сервисов. Хотя многому из рассмотренного можно было бы посвятить отдельные главы и обсуждения, мы остановились на роли этих понятий, которую они играют для полнофункциональных клиентов (HTML5, iOS, Android и т. д.) в системах, основанных на применении микросервисов.

Мы рассмотрели вопросы эффективного и быстрого создания клиентов с целью упростить некоторые действия по обмену данными с нижестоящими сервисами с помощью шаблона `RestTemplate`, а также Netflix-библиотеки Feign. Кроме того, обсудили вопрос интеграции балансировки нагрузки на стороне клиента.

Мы разработали сервис аутентификации OAuth `auth-service`, делегирующий аутентификацию провайдеру `AuthenticationProvider`, поддерживаемому JPA. В ином варианте можно было бы всецело делегировать аутентификацию другому OAuth-совместимому сервису аутентификации. Ваш сервис авторизации может работать в качестве фасада для одного или нескольких других нижестоящих сервисов авторизации, таких как Facebook или GitHub. Именно этим и занимается приведенный в данной главе пример кода `social-auth-service`. Если захотите продолжить данный маршрут, то можете проявить активность и сделать этот пример более интересным.

Пограничные сервисы очень похожи (по крайней мере концептуально) на ряд других технологий и шаблонов; частично продублированные обязанности имеются у всех далее перечисленных средств: и у SOFEA (service-oriented front end applications), и у BFFs (backends-for-frontends), и у API-шлюзов. Как бы ни называлось все вышеперечисленное, эти сервисы должны приводить к снижению объема бизнес-логики и служить объединению большего количества клиентов с более широким количеством сервисов без снижения скорости передачи данных.

Часть III
Интеграция данных

9 Управление данными

Эту главу мы посвятим вопросам управления данными при создании масштабируемого приложения, предназначенного для работы в облачной среде. Мы рассмотрим известные методы моделирования данных предметной области, а также уделим внимание способам предоставления проектами Spring Data хранилищ для управления данными. Кроме этого, обсудим несколько примеров микросервисов, управляющих исключительным доступом к источнику данных с помощью проектов Spring Data.

Моделирование данных

Удачно сконструированные модели данных помогают эффективно выражать намерения предметной области в наших программных приложениях. Модели предметных областей, подобные той, что показана на рис. 9.1, могут быть построены для выражения их важнейших аспектов. Одна из наиболее успешных технологий для моделирования области была впервые предложена Эриком Эвансом (Eric Evans) в его фундаментальной книге *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley).

Эванс популяризировал концепции предметно-ориентированного проектирования, продвигая идею, что как специалисты по бизнесу, так и программисты в ИТ-организации должны иметь возможность эффективно общаться, используя однозначные понятия, описывающие объекты и модули в программном приложении.

Проблемой, которую предполагалось решить с помощью предметно-ориентированного проектирования, была сложность. Эванс снабдил свою книгу подзаголовком *Tackling Complexity in the Heart of Software* («Структуризация сложных программных систем»). Он имел в виду следующее: специалистам нужно сосредоточиться на том, как упростить создание и сопровождение ПО путем устранения сложностей во взятой за основу модели бизнеса. Сложности кроются в запутанности бизнес-процессов и функций, создаваемых компаниями для обслуживания клиентов; модели

предметных областей определяют эти сложности и предоставляют специалистам язык для написания более качественных программ для ведения бизнеса.

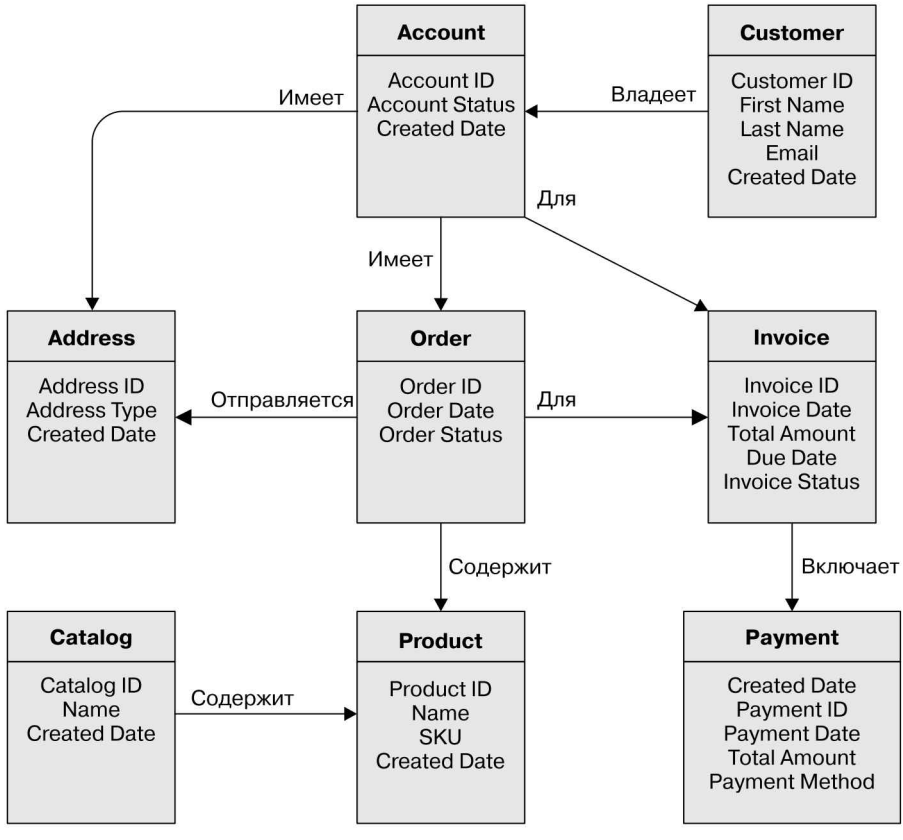


Рис. 9.1. Модель предметной области, отражающая взаимоотношения между ее классами

Рассмотрим такой пример. Если разработчик в программном модуле называет клиента (`Customer`) пользователем (`User`), то остальные специалисты затем вынуждены применять неоднозначные понятия, которые могут полностью дезориентировать специалистов-предметников. Вот описание требований заказчика для управления учетной записью клиента:

> *Мне как клиенту нужна возможность создания пользователей, которые могут управлять набором учетных записей.*

Это требование было написано специалистом по предметной области. В нем описан порядок управления учетной записью и выражено четкое различие между *клиентом* и *пользователем*, но не разъяснено, способен ли *пользователь* также быть *клиентом*.

Без модели предметной области, проводящей такое различие, программист может прийти к выводу, что будет проще всего закодировать один класс предметной области `User` (Пользователь). Позже, в отдельной версии, специалист-предметник может уточнить, что `Customer` и `User` — это не одно и то же. Впоследствии данное уточнение превратится в более затратное конструкторское решение с серьезными издержками по разработке новых функций, что приведет к срыву программистами исходных сроков выпуска ПО.

Большинство хранилищ данных задумываются как наиболее подходящие по типу при минимальном вкладе в удобство их применения. По нашему мнению, стоит все же следовать всему, что действительно сближает позиции специалистов-предметников и программистов. Хранилище данных не нужно рассматривать как место для хранения байтов, оно предназначается для хранения сущностей и взаимоотношений, описываемых вашей предметной областью. Наша неизменная цель — увеличение отдачи от наших вложений (как по времени, так и по средствам) в конкретную технологию при сведении к минимуму требований программного кода по работе с конкретным хранилищем данных, не особо способствующим продвижению наших бизнес-устремлений. И в этом, в частности, стремятся помочь среды Spring и Spring Data.

Системы управления реляционными базами данных (СУРБД)

Модель реляционной базы данных на протяжении многих десятилетий была основным элементом решений по созданию транзакционных хранилищ данных. По мере поступательного развития технологий в сторону облачных архитектур рынок начинает выдвигать требования по использованию других типов моделей данных, предоставляющих такие же транзакционные гарантии, что и СУРБД. Своеобразным мастером на все руки стала база данных SQL, но ей под силу не все. В настоящее время, по сравнению с технологиями поддержки постоянного хранения данных, она не в состоянии отвечать самым строгим требованиям, предъявляемым к проведению транзакций. Говоря в ироничном ключе, это далеко не лучший способ выражения взаимосвязей. База данных SQL поддерживает двоичные данные, но недостаточно хорошо и эффективно. У многих баз данных имеется геопространственная поддержка, однако обязательным характером такая поддержка обладает не всегда. Базы данных SQL способны справляться с большим количеством транзакций, связанных с чтением данных, но зачастую только после того, как проектировщик схемы данных прибегает к созданию нестандартной модели и идет на уступки в отношении согласованности для достижения высокой доступности данных. Короче говоря, хотя СУРБД не обладают всеми нужными свойствами, не исключено, что их некоторые специализированные альтернативы стоило бы изучить. Если вы используете СУРБД, то JPA, являющаяся надстройкой над JPA-реализациями наподобие Hibernate, может быть неплохо подогнана к желаемому результату при условии отображения записей на СУРБД-схему и наоборот (отображения СУРБД-схемы

на записи). JPA (и объектно-реляционные отображатели — ORM — в целом) позволяет разработчикам больше концентрироваться на модели предметной области и меньше думать о нюансах самой СУРБД. Для облегчения развития вам приходится жертвовать (в разумных пределах) возможностями управления.

NoSQL

Базы данных категории NoSQL предоставляют большое разнообразие моделей данных со специализированными характеристиками, оптимизированными под удовлетворение потребностей конкретных сценариев использования. Особенно это пригодится в микросервисах, так как позволит разложить на составляющие ограниченные контексты модели предметной области для применения специализированной характеристики базы данных NoSQL. Доступ к данным осуществляется через призму API микросервиса, поэтому о применяемой технологии клиенты ничего не знают.

Для описания архитектуры, задействующей несколько типов моделей баз данных, Мартин Фаулер ввел такое понятие, как *polyglot persistence* (приверженность к многоязычию). Приложения, рассчитанные на применение единой реляционной базы данных, могут быть разделены на составные части для использования микросервисов, управляющих собственными базами данных NoSQL, в зависимости от сценария применения конкретного фрагмента приложения. Это будет в особенности полезно для совершенно новых проектов, где бизнес требует более сложных и незамедлительно предлагаемых решений. Поскольку базы данных NoSQL предоставляют особые модели данных, оптимизированные под конкретные ситуации, то потребности в подстраивании модели реляционной базы данных для решения конкретной задачи снижаются.

Spring Data

Spring Data (http://projects.spring.io/spring-data/) — комплексный проект с открытым кодом, предоставляющий знакомую абстракцию для взаимодействия с хранилищем данных наряду с сохранением особых черт ее модели базы данных. Существует более десятка (официальных) проектов Spring Data, охватывающих широкий диапазон популярных баз данных, включая модели данных СУРБД и NoSQL. Имеются модули Spring Data, поддерживающие среди прочих такие системы, как JDBC, JPA, MongoDB, Neo4J, Redis, Elasticsearch, Solr, Gemfire, Cassandra и Couchbase.

JPA (и в более широком смысле Spring Data JPA) охватывает все популярные продукты поставщиков СУРБД, включая Oracle, MySQL, PostgreSQL, SQL Server, H2, HSQL, Derby и др.

Структура приложения Spring Data

В начале работы со Spring Data полезно будет разобраться в схемах, используемых для создания уровней доступа к данным. Начнем с понимания того, какие основные составляющие применяются для взаимодействия с хранилищем данных.

Класс предметной области

Первой составляющей станет класс объектов, представляющий объекты вашей модели предметной области. *Класс предметной области* — базовый, работающий в качестве модели для данных вашей предметной области. Такие классы состоят из набора закрытых полей и предоставляют содержимое, используя открытые геттеры и сеттеры, в зависимости от конструкции модели этой области (пример 9.1).

Пример 9.1. Основной класс предметной области, представляющий модель User

```
public class User {    ❶

    private Long id;
    private String firstName;  ❷
    private String lastName;
    private String email;
```

❶ Название класса, представляющего концепцию, существительное или объект, в модели предметной области.

❷ Закрытые поля отображены на свойства объектов данных в исходном хранилище данных.

Классы предметных областей представляют объекты, отображаемые на объекты данных (такие как таблицы) в хранилище данных приложения.

Хранилища

Первичный метод доступа к данным в приложении Spring Data — *хранилище*. Данное понятие было введено Эриком Эвансом в его книге о предметно-ориентированном проектировании (*Domain-Driven Design*). Среда Spring уже давно поддерживает стереотипную аннотацию `@Repository`. Последняя, помимо маркировки класса в качестве компонента с помощью аннотации `@Component`, сигнализирует среде Spring, что bean-компонент может выдать исключения, характерные для технологии сохранения, и их следует привести к иерархии, предоставляемой средой

Spring. Вследствие этого потребителям нужно лишь озаботиться о типе исключения для таких аспектов, как нарушение ограничений.

Spring Data расширяет области поддержки. Эта среда может обеспечить реализации предоставляемого пользователем определений интерфейса, чьи сигнатуры методов управления данными определены в соответствии с соглашениями самой среды. Вы можете описать методы определения собственного интерфейса или вместо этого расширить любой из нескольких удобных определений предоставленного интерфейса. Наиболее простой, но все же полезной реализацией этих интерфейсов является `CrudRepository`.

CRUD — акроним, включающий такие действия, как Create (создание), Read (чтение), Update (обновление) и Delete (удаление). Они описывают основные возможности управления данными в любой разновидности технологий постоянного хранения. Не случайно в HTTP-протоколе для управления такими ресурсами, как сетевые транзакции, используются немного другие описания действий, называемые Post, Get, Patch и Delete.

В `CrudRepository` поддерживаются CRUD-операции на основе двух частей информации: типа класса предметной области и его типа идентификатора. При необходимости создать хранилище для класса предметной области `User` можно использовать интерфейс `CrudRepository` (пример 9.2).

Пример 9.2. Хранилище Spring Data для класса предметной области User

```
public interface UserRepository extends CrudRepository<User, Long> {
}
```

Не нужно реализовывать `UserRepository`. Среда Spring Data предоставит bean-компонент, реализующий контракт, который при необходимости можно вставить в любой другой bean-компонент Spring.

Конструирование пакетов Java для данных предметной области

Стиль конструирования пакетов — более важный фактор при создании микросервисов. Мы рекомендуем конструировать классы и пакеты наиболее удобным для вас образом, но при этом предлагаем схему, которую полагаем подходящей для создания микросервисов с помощью Spring Data и Spring Boot.

Сначала нужно определить наличие ограниченных контекстов в нашей модели предметной области. Для этого упражнения рассмотрим модель, показанную на рис. 9.2.

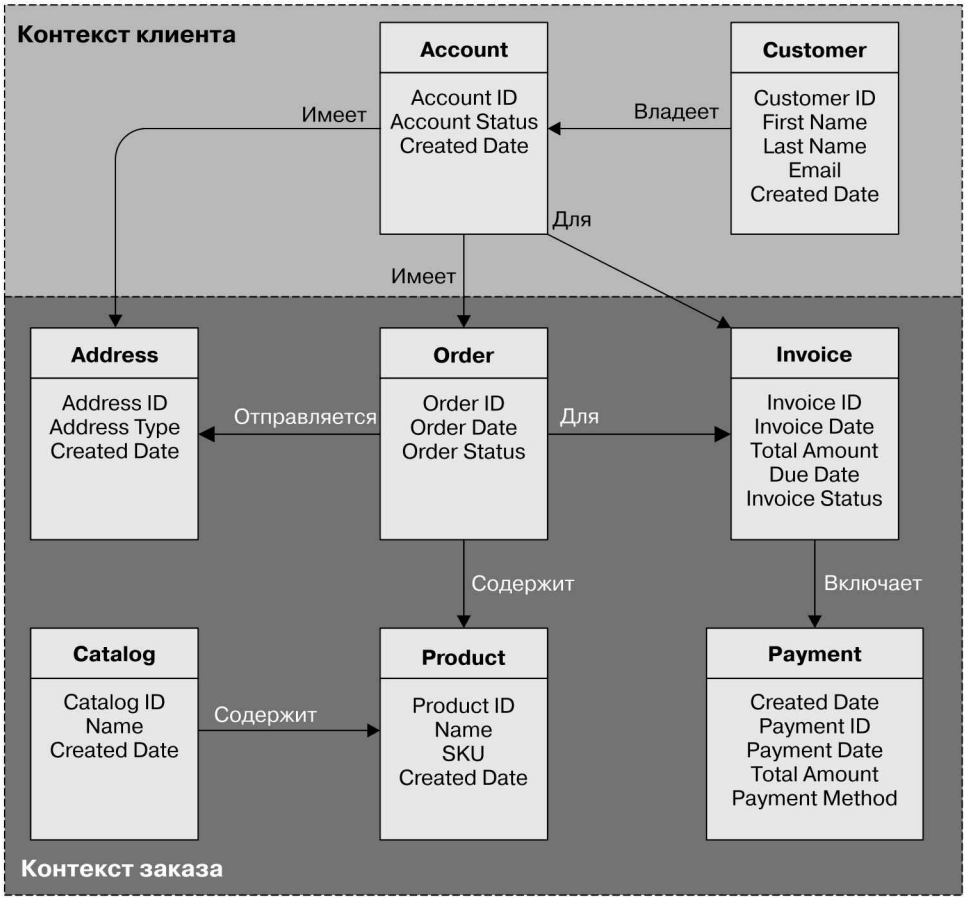


Рис. 9.2. Модель предметной области с двумя ограниченными контекстами

На схеме представлена модель предметной области, имеющая четкую границу между контекстами клиента и заказа. Предположим, что для охвата каждого из этих ограниченных контекстов мы хотим создать два микросервиса. Нам нужно будет сконструировать пакеты, чтобы потом можно было легко и просто перенести классы предметных областей и хранилища для концепции предметной области. Для улучшения подготовки к этому упражнению по реорганизации можно сгруппировать классы предметных областей и хранилища, которые обслуживают единый класс предметной области в один пакет. На рис. 9.3 и 9.4 показана структуризация кода в данной предметной области.

Поддерживаемые хранилища. Проекты Spring Data поддерживают несколько разновидностей хранилищ, расширяющих интерфейс `Repository`. Некоторые из них показаны в табл. 9.1.

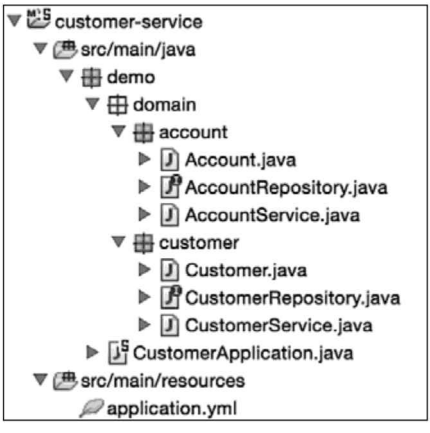


Рис. 9.3. Структура пакета для контекста Customer

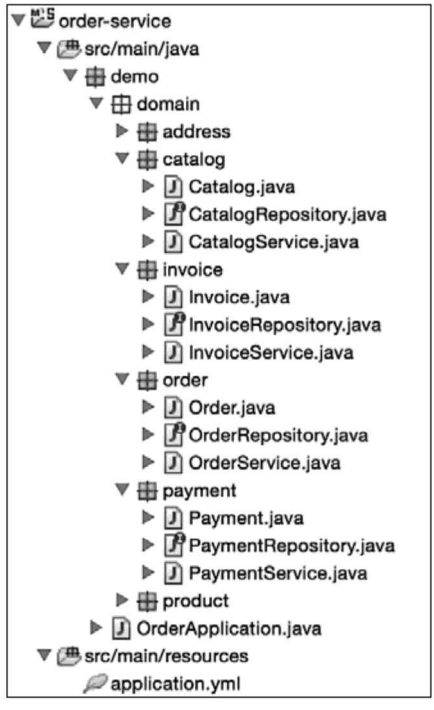


Рис. 9.4. Структура пакета для контекста Order

Таблица 9.1. Наиболее распространенные хранилища Spring Data

Тип хранилища	Проект Spring Data	Назначение
Repository	Spring Data Commons	Предоставляет основную абстракцию для хранилищ Spring Data
CrudRepository	Spring Data Commons	Расширяет Repository и добавляет утилиту для основных CRUD-операций
PagingAndSortingRepository	Spring Data Commons	Расширяет CrudRepository и добавляет утилиту для разбивки и сортировки записей
JpaRepository	Spring Data JPA	Расширяет PagingAndSortingRepository и добавляет утилиту для моделей баз данных JPA и СУРБД
MongoRepository	Spring Data MongoDB	Расширяет PagingAndSortingRepository и добавляет утилиту для управления документами MongoDB
CouchbaseRepository	Spring Data Couchbase	Расширяет CrudRepository и добавляет утилиту для управления документами Couchbase

Каждый интерфейс хранилища, поставляемого со Spring Data, включает абстракцию для конкретной утилиты, которая может понадобиться вашему приложению Spring Data. Показанная выше таблица описывает некоторые из множества интерфейсов хранилища в экосистеме проектов Spring Data. Оба интерфейса, и `CrudRepository`, и `PagingAndSortingRepository`, принадлежат библиотеке Spring Data Commons, предоставляя абстракции, используемые проектами Spring Data конкретных поставщиков.

Интерфейс `MongoRepository` — первичная абстракция хранилища, используемая исключительно в проекте Spring Data MongoDB. Это хранилище расширяет интерфейс `PagingAndSortingRepository`, предоставляя основные возможности управления данными для взаимодействия со страничными записями базы данных (рис. 9.5).

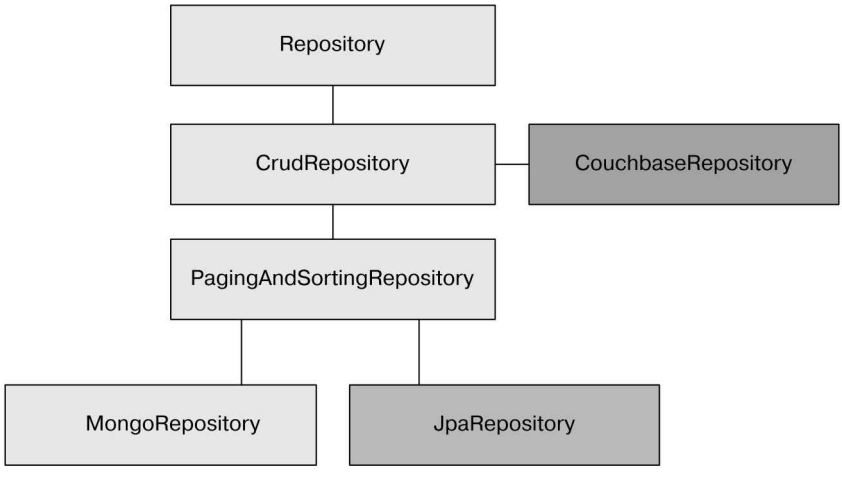


Рис. 9.5. Три основных хранилища в Spring Data Commons предоставляют базовый доступ к данным, находящимся в хранилищах данных конкретных поставщиков

Начало работы с доступом к данным СУРБД на JDBC

В среде Spring Boot используется автоконфигурирование для загрузочных зависимостей приложения с помощью набора исходных установок. Нам как минимум понадобится настройка SQL `DataSource`, JDBC и JPA. Среда Spring Boot произведет настройку подключения к локальному хранилищу данных, если сможет найти в пути к классам один из нескольких известных драйверов `DataSource`. В нашем примере мы применим MySQL, поэтому к пути к классам следует добавить зависимость `mysql : mysql-connector-java`. Если же к пути к классам мы добавим стартер `spring-boot-starter-data-jpa`, то будет также настроен и JPA-интерфейс. Стартер JPA автоматически берет автосконфигурированный источник данных `javax.sql.DataSource`,

указывающий на MySQL, и задействует его в качестве источника данных для нашей конфигурации JPA.

Вы можете подстроить `DataSource` с помощью ряда удобных свойств JDBC и JPA. В идеале служебную информацию для таких аспектов, как источники данных, желательно было бы содержать отдельно от кода самого приложения. Чтобы получить дополнительные сведения о конфигурации, обратитесь к нашему обсуждению данной темы в главе 3. Вполне обоснованно содержать менее конфиденциальные исходные установки конфигурации, относящиеся исключительно ко времени разработки в самой кодовой базе, под специфичным для разработки профилем. Соответствующее решение показано в примере 9.3.

Пример 9.3. Простой файл application.yml

```
spring:
  profiles:
    active: development
---
spring:
  profiles: development
  jpa:
    database: MYSQL
  datasource:  (1)
    url: jdbc:mysql://localhost/test
    username: dbuser
    password: dbpass
```

(1) В блоке свойства `spring.datasource` выполняется настройка деталей подключения, конкретизированных под СУРБД.

Изначально среда Spring Boot автоматически конфигурирует пул подключений для нашего источника данных JDBC DataSource. По умолчанию этот пул основан на Apache Tomcat DBCP, но, добавляя к пути к классам нужную библиотеку, можно настроиться на использование Commons DBCP или Hikari CP.

Поддержка имеющейся в Spring технологии JDBC

Один из классических механизмов доступа к данным — имеющийся в среде Spring Framework шаблон `JdbcTemplate`. Он предоставляет функции управления реляционными базами данных на основе SQL, поддерживающие спецификацию JDBC. В этом разделе мы поговорим о способах использования `JdbcTemplate`.

Благодаря своей доступности с ранних дней существования среды Spring шаблон JdbcTemplate вполне может претендовать на самую распространенную и широко известную реализацию шаблона проектирования. Проект Spring Data включает ряд других реализаций шаблонов, поддерживающих иные хранилища данных.

Теперь можно выполнить SQL-запросы к базе данных MySQL, сконфигурированной в качестве источника данных с помощью файла `application.yml` для разработочного профиля. Создадим таблицу `User` и наладим взаимодействие с ее записями, используя шаблон `JdbcTemplate` (пример 9.4).

Пример 9.4. Работа с нашим источником данных с применением простого языка SQL и шаблона JdbcTemplate

```
package demo;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Component;

import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

@Component
class JdbcCommandLineRunner implements CommandLineRunner {

 private final Logger log = LoggerFactory.getLogger(getClass());

 private final JdbcTemplate jdbcTemplate;

 @Autowired
 JdbcCommandLineRunner(JdbcTemplate jdbcTemplate) {
  this.jdbcTemplate = jdbcTemplate;
 }

 @Override
 public void run(String... strings) throws Exception {
  ❶
  jdbcTemplate.execute("DROP TABLE user IF EXISTS");
  jdbcTemplate
   .execute("CREATE TABLE user( id serial, first_name " +
      " VARCHAR(255), last_name VARCHAR(255), email VARCHAR(255))");

  ❷
  List<Object[]> userRecords = Stream
   .of("Michael Hunger michael.hunger@jexp.de",
    "Bridget Kromhout bridget@outlook.com", "Kenny Bastani kbastani@yahoo.com",
    "Josh Long jlong@hotmail.com").map(name -> name.split(" "))
   .collect(Collectors.toList());

  jdbcTemplate
```

```
    .batchUpdate(
     "INSERT INTO user(first_name, last_name, email) VALUES (?,?,?)",
     userRecords);

  ❸
  RowMapper<User> userRowMapper = (rs, rowNum) -> new User(rs.getLong("id"),
   rs.getString("first_name"), rs.getString("last_name"), rs.getString("email"));

  List<User> users = jdbcTemplate.query(
   "SELECT id, first_name, last_name, email FROM user WHERE first_name = ?",
   userRowMapper, "Michael");

  users.forEach(user -> log.info(user.toString()));
 }
}
```

❶ Создание схемы.

❷ Создание нескольких надуманных примеров записей и запись их в базу данных с помощью удобной функции `batchUpdate` в `JdbcTemplate`.

❸ Обход каждой записи путем *отображения* записей на объекты с последующей регистрацией результатов в виде этих объектов.

Надеемся, этот код достаточно краток, чтобы в нем можно было разобраться с первого взгляда. Допустим, вам когда-либо приходилось использовать обычный JDBC API. Тогда вы знаете, что основная часть кода связана с созданием `DataSources`, `Connections` и `Statements`, проходом через наборы результатов, управлением транзакциями, работой с исключениями, которые, вероятнее всего, никогда не выдаются и откатами, если что-то не заладится. Шаблон `JdbcTemplate` берет все на себя, позволяя сосредоточиться на сути решаемой проблемы. При выдаче запроса он автоматически создает инструкцию, запускает запрос, обходит каждую запись и дает предоставленной нами реализации `RowMapper<T>` функцию обратного вызова для преобразования результата.

Запустите пример, и на экране вы увидите результат: `Michael Hunger` (пример 9.5).

Пример 9.5. Взаимодействие с источником данных с помощью простого SQL и JdbcTemplate

```
User(id=1, firstName=Michael, lastName=Hunger, email=michael.hunger@jexp.de)
```

Этот довольно простой пример был разработан для выделения одной из ключевых концепций, имеющейся в Spring, а именно объекта шаблона. Нам еще попадется данная разновидность поддержки в Spring Data. Чтобы ничего не усложнять, для создания схемы и таблицы в коде Java мы запустили инструкции SQL. Но есть и более подходящая альтернатива. По соглашению среда Spring при запуске приложения начнет обрабатывать два файла: `src/main/resources/schema.sql` и `src/main/resources/data.sql`. Достаточно будет поместить любую схему на языке

определения данных DDL (data definition language), например инструкции для создания нашего пользователя и таблицы, в файл `schema.sql`. Мы могли бы поместить свои образцовые записи для `Bridget`, `Michael` и т. д. в файл `data.sql`. Затем нашему коду понадобилось бы всего несколько последних строк для работы с запросами и отображением записей из источника данных.

Примеры Spring Data

Ранее мы изучали некоторые основные понятия Spring Data: классы предметных областей и хранилища. А теперь углубимся в образцы проектов, использующих разные технологии. Мы рассмотрим примеры сервисов данных для следующих проектов Spring Data:

- Spring Data JPA (MySQL);
- Spring Data MongoDB;
- Spring Data Neo4j;
- Spring Data Redis.

Следует иметь в виду, что мы также применим Project Lombok (https://projectlombok.org/), процессор аннотаций, работающий в ходе компиляции, предназначенный для сокращения объема шаблонного кода в нашей кодовой базе. Аннотация @Data предписывает процессору Lombok создание акцессоров и мутаторов, метода toString и метода equals/hashCode. Аннотация @NoArgsConstructor предписывает процессору Lombok создание конструктора без аргументов. Аннотация @AllArgsConstructor предписывает процессору Lombok создание конструктора, принимающего каждое поле, имеющееся в классе. Кроме того, можно создавать собственные конструкторы или же переопределять любой из акцессоров или мутаторов даже при наличии этих аннотаций.

В качестве примера предметной области, используемой для этого набора примеров данных, станет фигурировать интернет-магазин. Мы применим модель, в которой дается описание приложения планирования ресурсов предприятия — enterprise resource planning (ERP). Каждый из наших сервисов будет распространяться на определенный ограниченный контекст предметной области.

Приложение интернет-магазина, которое мы используем для образцов проектов, будет принадлежать фиктивной компании *Monolithic Ltd.*, представляющей торговую марку одежды и желающей переместить разработку приложения для своего интернет-магазина в облачную среду.

У Monolithic Ltd. возникли проблемы с возможностью развертывания командой разработчиков новых функций в уже существующем интернет-магазине. Вдохнов-

ленная переходом на более современные методы создания ПО, стартап-компания решила провести ребрендинг своих предложений в *Cloud Native Clothing*.

Посмотрим на модель предметной области, которую *Cloud Native Clothing* будет применять для создания нового интернет-магазина (рис. 9.6).

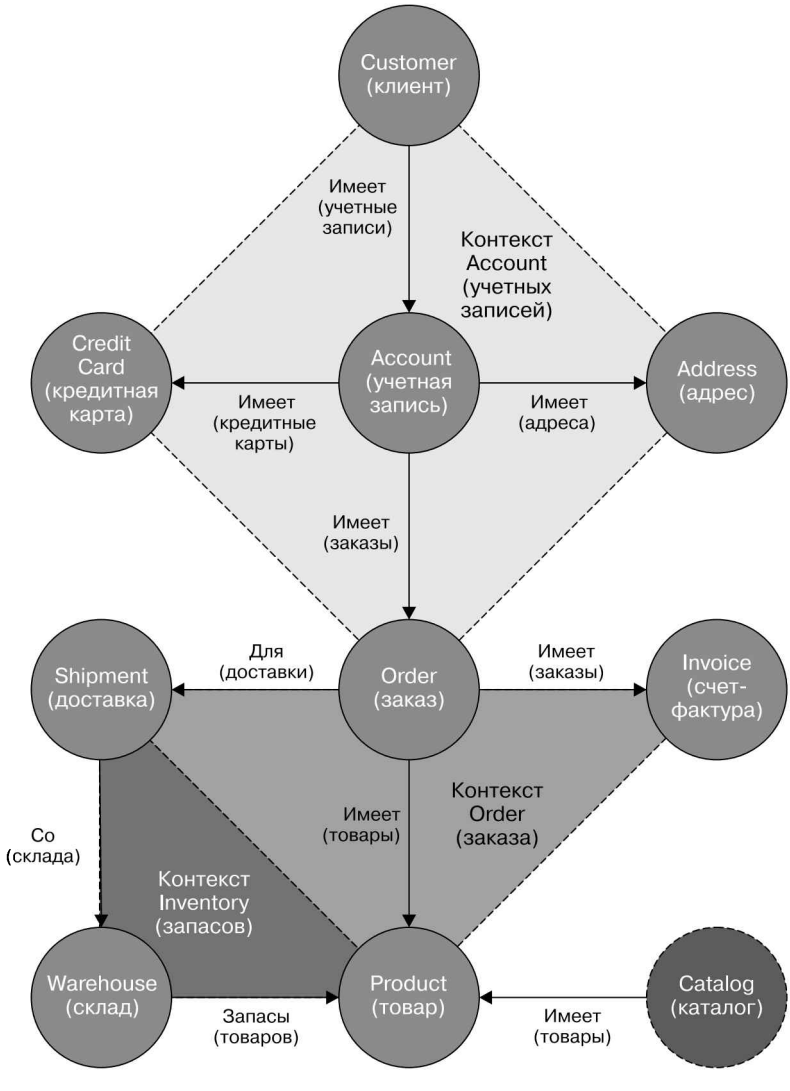


Рис. 9.6. Модель предметной области, используемая Cloud Native Clothing

Из модели предметной области *Cloud Native Clothing* можно увидеть наличие нескольких ограниченных контекстов. Каждый такой контекст на схеме помечен

соответствующей надписью. Согласно принципам, изложенным в DDD, ограниченные контексты представляют собой способ применения модульного проектирования в отношении части предметной бизнес-области с ее разбиением на отдельные унифицированные модели. Каждый ограниченный контекст может иметь набор не связанных между собой понятий при одновременном выявлении связей между общими понятиями. Мы решили создать модель, способную сформулировать понятия предметной бизнес-области, а также ясно изложить проект приложения с помощью хранилищ Spring Data.

Каждый показанный на схеме класс предметной области представлен в форме окружности. Каждая окружность представляет также хранилище Spring Data. Хранилища, в контексте данной схемы, требуются для любого класса области, которому нужно будет предоставить возможности создания запросов внутри ограниченного контекста. Поэтому при использовании данного типа модели мы всегда будем рассматривать окружность и как *класс предметной области*, и как *хранилище*. Взаимосвязи между хранилищами указывают на некоторый уровень связанности в каждом классе области.

Стрелки между понятиями показывают *явные связи* между классами предметных областей. Пунктирная линия показывает подразумеваемую связанность через *соединения хранилища*. Пунктирная линия свидетельствует, что должен быть запрос к хранилищу, работающий в качестве связующего моста между понятиями предметной области. Такой мост допускается только внутри контекста общих понятий класса области, существующего между двумя хранилищами. Например, `Account` образует связь между `Order` и `Address` (рис. 9.7).

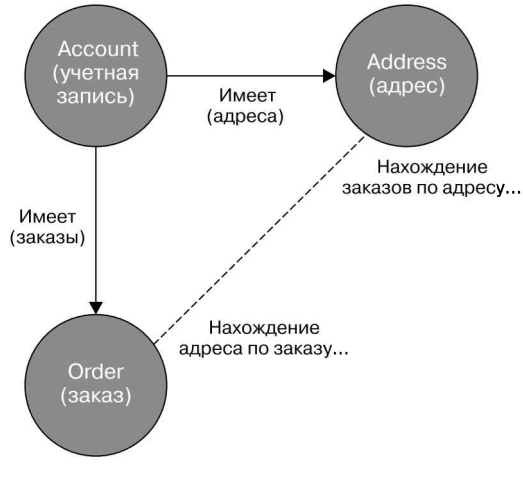


Рис. 9.7. Подразумеваемые отношения могут прослеживаться через общий класс предметной области

Как показано в примере 9.6, в SQL это понятие просто называется JOIN-запросом между таблицами.

Пример 9.6. Поиск заказов для адреса путем присоединения к таблице учетных записей

```
SELECT o.* FROM orders o
INNER JOIN account ac ON o.accountId = ac.Id
INNER JOIN address ad ON ad.accountId = ac.Id
WHERE ad.id = 0
```

Эти ограниченные контексты мы станем применять для создания трех отдельных сервисов данных, использующих конкретный проект Spring Data (табл. 9.2).

Таблица 9.2. Микросервисы для нашего приложения Cloud Native Clothing

Сервис	Источник данных	Проект Spring Data
Сервис учетных записей Account	MySQL	Spring Data JPA
Сервис заказов Order	MongoDB	Spring Data MongoDB
Сервис запасов Inventory	Neo4j	Spring Data Neo4j

Spring Data JPA

Проект Spring Data JPA предлагает поддержку управления данными для СУРБД и SQL. Следует заметить, что этот проект относится исключительно к проектам Spring Data, предоставляющим абстракции хранилища для СУРБД. Но существует также поддерживаемый сообществом проект сторонних производителей под названием Spring Data JBDC (https://github.com/nurkiewicz/spring-data-jdbc-repository), который неплохо справляется с непосредственной поддержкой JDBC.

JPA означает Java Persistence API, представляя собой спецификацию, описание которой впервые было дано и представлено в качестве части JSR 220, принадлежащей спецификации JCP (Java Community Process). JPA предоставляет абстракции и реализации для ORM-технологий конкретных производителей, таких как Hibernate и DataNucleus. Это довольно мощный аспект проекта Spring Data, позволяющий виртуально использовать любую СУРБД-технологию, предоставляющую драйвер, поддерживающий спецификацию JPA.

В приложении Spring Boot для поддержки SQL ORM не нужно применять JPA. Кроме всего прочего, существует весьма успешная интеграция Spring Boot для таких сторонних проектов, как MyBatis и JOOQ. Можете проверить их на деле! Мы рассматриваем JPA из-за высокой распространенности этой спецификации, а не из-за того, что у нас нет выбора.

Сервис учетных записей Account

Для нашего сервиса учетных записей `Account` мы задействуем Spring Data JPA. Сервис `Account` охватывает контекст учетных записей модели предметной области *Cloud Native Clothing*. Spring Data JPA представляет в экосистеме Spring Data особый проект. Он предоставляет единую библиотеку, прекрасно работающую с несколькими (JDBC-совместимыми) источниками данных, в то время как другие проекты Spring Data взаимодействуют только с одной разновидностью источника данных. Одно из косвенных преимуществ заключается в возможности использования встроенного источника данных для тестирования с последующим переключением на применение более полноценной базы данных в других режимах работы. Чтобы получить четкое разделение, мы воспользуемся имеющимися в среде Spring *профилями*.

В этом подразделе мы собираемся рассмотреть все основные вопросы настройки приложения Spring Data JPA для сервиса `Account`, принадлежащего компании *Cloud Native Clothing*.

Использование профилей для различных источников данных

Чтобы включить предметную область JPA, мы добавим к нашей сборке Maven три зависимости:

- `org.springframework.boot : spring-boot-starter-data-jpa`;
- `com.h2database : h2`;
- `mysql : mysql-connector-java`.

Теперь, заполучив заготовку для нашего проекта Spring Data JPA с драйверами баз данных MySQL и H2 в путях к классам, нужно сконфигурировать свое приложение под использование двух баз данных в правильном контексте. База данных MySQL запускается вне нашего приложения; мы будем подключаться к ней удаленно при старте приложения. Запуск встроенной базы данных H2 понадобится лишь в момент начала наших комплексных тестов. Среда Spring предоставляет способ конфигурирования приложения для запуска с отдельным набором настроек в зависимости от того, какой из профилей активен на момент старта. В файле `application.yml` можно создать профили: для режима разработки и для тестирования (пример 9.7).

Пример 9.7. Конфигурирование вашего приложения для разработки и тестирования

```
spring:
  profiles:
    active: development  (1)
---
spring:
  profiles: development  (2)
```

```
  jpa:
    show_sql: false
    database: MYSQL
    generate-ddl: true
  datasource:
    url: jdbc:mysql://192.168.99.100:3306/dev
    username: root
    password: dbpass
---
spring:
  profiles: test  ❸
  jpa:
    show_sql: false
    database: H2
  datasource:
    url: jdbc:h2:mem:testdb;DB_CLOSE_ON_EXIT=FALSE
```

❶ Это по умолчанию активный профиль при запуске приложения.

❷ Это конфигурация для профиля разработки `development`.

❸ Это конфигурация для профиля тестирования `test`.

Теперь, имея два отдельных профиля, определенных для конфигураций разработки и тестирования, нужно сконфигурировать наши комплексные тесты для применения профиля `test`. Поскольку исходным активным профилем в свойствах приложения установлен `development`, то комплексные тесты будут пытаться использовать профиль `development`, что нежелательно. Решить данную проблему поможет переопределение активного профиля, но только при нахождении в контексте наших комплексных тестов.

С такой настройкой понадобится лишь указать в комплексных тестах среды Spring аннотацию `@ActiveProfiles`, указывающую на профиль `test`. При запуске комплексных тестов для сервиса учетных записей `AccountService` воспользуемся встроенной в память базой данных H2. С ее помощью можно запустить тесты в любой среде сборки, не переживая за наличие подключения к внешней зависимости.

Описание предметной области сервиса учетных записей Account с помощью JPA

Для классов предметной области сервиса `Account` мы создадим объекты JPA. Класс JPA является аннотированным, он будет отображаться на таблицу реляционной базы данных (либо MySQL, либо, как в данном случае, H2).

В модели предметной области для нашей компании *Cloud Native Clothing* имеются четыре ограниченных контекста. В контексте `Account` можно увидеть наличие пяти хранилищ: `Account`, `Address`, `Order`, `CreditCard` и `Customer` (рис. 9.8).

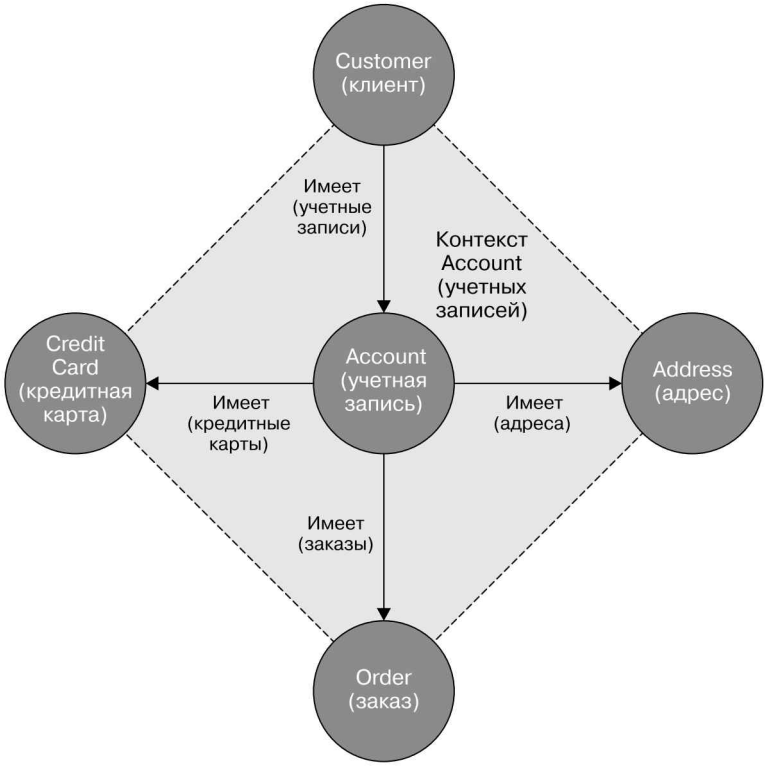


Рис. 9.8. Контекст Account для сервиса Account

Следующим шагом по созданию сервиса `Account` станет конструирование классов предметных областей. В каждом из проектов Spring Data для этого предусмотрены немного разные процессы. Что касается Spring Data JPA, у нас в арсенале имеется ряд инструментов, которыми можно воспользоваться для аннотирования частей классов предметных областей, чтобы добиться соответствия модели предметной области в контексте `Account`.

Теперь создадим каждый из этих классов предметных областей для сервиса `Account` в виде классов объектов JPA с помощью аннотаций JPA, описание которых было дано в последней таблице. Нужно создать классы предметных областей для каждого из понятий в контексте `Account`, включающем такие объекты (показанные на рис. 9.9):

- `Account`;
- `Customer`;
- `CreditCard`;
- `Address`.

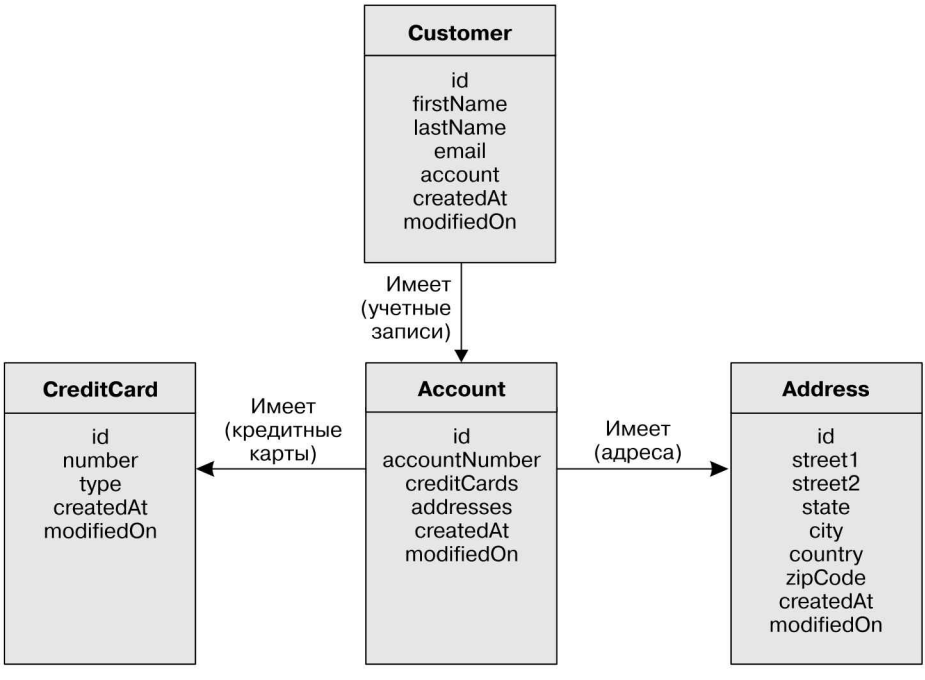


Рис. 9.9. Схема взаимоотношений объектов для контекста Account

Мы создадим класс предметной области со свойствами, перечисленными в показанной выше схеме. Каждый класс будет иметь аннотированные свойства для взаимосвязей с классом `Account`.

Рассмотрим несколько типов в нашей предметной области и порядок их отображения с помощью аннотаций JPA в примерах 9.8–9.11, а также на рис. 9.9.

Пример 9.8. Класс предметной области Account в качестве объекта JPA

```
package demo.account;

import demo.address.Address;
import demo.creditcard.CreditCard;
import demo.customer.Customer;
import demo.data.BaseEntity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;
import java.util.HashSet;
import java.util.Set;

@Entity
@Data
```

```
@NoArgsConstructor
@AllArgsConstructor
public class Account extends BaseEntity {

 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private Long id;    (1)

 private String accountNumber;

 (2)
 @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
 private Set<CreditCard> creditCards = new HashSet<>();

 @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
 private Set<Address> addresses = new HashSet<>();

 public Account(String accountNumber, Set<Address> addresses) {
  this.accountNumber = accountNumber;
  this.addresses.addAll(addresses);
 }

 public Account(String accountNumber) {
  this.accountNumber = accountNumber;
 }
}
```

(1) Аннотация `@GeneratedValue` вызовет единичное приращение уникального идентификатора для поля `@Id`. В данном случае мы (избыточно и в явном виде) указали на то, что первичный ключ должен использовать значение с автоприращением.

(2) Аннотация `@OneToMany` дает JPA-объекту описание связи по внешнему ключу (FK-связи).

Пример 9.9. Класс предметной области Customer, выраженный в виде JPA-объекта

```
package demo.customer;

import demo.account.Account;
import demo.data.BaseEntity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Customer extends BaseEntity {
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
```

```
 private Long id;

 private String firstName;

 private String lastName;

 private String email;

 @OneToOne(cascade = CascadeType.ALL)
 private Account account;

 public Customer(String firstName, String lastName, String email,
  Account account) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.email = email;
  this.account = account;
 }
}
```

Пример 9.10. Класс предметной области Address, выраженный в виде JPA-объекта

```
package demo.address;

import demo.data.BaseEntity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Address extends BaseEntity {

 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private Long id;

 private String street1;

 private String street2;

 private String state;

 private String city;

 private String country;

 private Integer zipCode;

 @Enumerated(EnumType.STRING)
```

```
 private AddressType addressType;

 public Address(String street1, String street2, String state, String city,
  String country, AddressType addressType, Integer zipCode) {
  this.street1 = street1;
  this.street2 = street2;
  this.state = state;
  this.city = city;
  this.country = country;
  this.addressType = addressType;
  this.zipCode = zipCode;
 }

}
```

Пример 9.11. Класс предметной области CreditCard, выраженный в виде JPA-объекта

```
package demo.creditcard;

import demo.data.BaseEntity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString;

import javax.persistence.*;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class CreditCard extends BaseEntity {

 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private Long id;

 private String number;

 @Enumerated(EnumType.STRING)
 private CreditCardType type;

 public CreditCard(String number, CreditCardType type) {
  this.number = number;
  this.type = type;
 }
}
```

Обратите внимание: в различных классах аннотациям `@OneToMany` мы предоставляли некоторые параметры (пример 9.12).

Пример 9.12. Отображение, задаваемое аннотацией @OneToMany, можно подстраивать

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
private Set<Address> address = new HashSet<>();
```

- `CascadeType.ALL` указывает на то, что фиксирование транзакции приводит к каскадной пересылке ко всем связанным объектам JPA.
- `FetchType.EAGER` указывает автоматическое заполнение связей.

Аннотация `@OneToMany` описывает связь по внешнему ключу между таблицами, управляемыми классами объектов JPA. В случае со связью `Account` с `Address` мы располагаем множеством элементов `1..*` или отношением *«один ко многим»*.

Проведение аудита с помощью JPA

Для фиксации даты создания записи, а также времени ее последнего изменения мы можем применить аудит для JPA-классов. За это отвечает класс `BaseEntity`, наследуемый каждым из объектов JPA. Посмотрим, как он выглядит (пример 9.13).

Пример 9.13. Класс BaseEntity предоставляет возможность JPA-аудита

```
package demo.data;

import lombok.Data;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedDate;
import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import javax.persistence.EntityListeners;
import javax.persistence.MappedSuperclass;

@Data
@MappedSuperclass
❶
@EntityListeners(AuditingEntityListener.class)
❷
public class BaseEntity {

 @CreatedDate
 private Long createdAt;  ❸

 @LastModifiedDate
 private Long lastModified; ❹
}
```

❶ Эта аннотация определяет родительский класс, из которого наследуются объекты JPA.

❷ Данный код определяет слушатель функции обратного вызова, занимающейся аудитом, который будет наблюдать за жизненным циклом операций JPA.

❸ Эта аннотация при создании записи приведет к сохранению метки времени.

❹ Данная аннотация повлечет применение обновленной метки времени при обновлении записи.

Чтобы включить аудит JPA для нашего приложения Spring Boot, нужно выполнить еще одно действие. К классу конфигурации нашего приложения следует применить аннотацию `@EnableJpaAuditing`.

Разобравшись с созданием классов объектов JPA из отображенных на них родительских классов, можно проделать то же самое для других классов предметных областей в контексте `Account`. Создадим хранилища, чтобы получить возможность управления данными для сервиса `Account`.

`Account` и `Customer` являются агрегированными объектами — все остальные объекты возникают и исчезают при их возникновении и исчезновении. Поэтому для управления ими мы создадим хранилища (примеры 9.14 и 9.15).

Пример 9.14. Определение AccountRepository

```
package demo.account;

import org.springframework.data.repository.PagingAndSortingRepository;

public interface AccountRepository extends
 PagingAndSortingRepository<Account, Long> {

}
```

Пример 9.15. Определение CustomerRepository

```
package demo.customer;

import org.springframework.data.repository.PagingAndSortingRepository;

import java.util.Optional;

public interface CustomerRepository extends
 PagingAndSortingRepository<Customer, Long> {

 ❶
 Optional<Customer> findByEmailContaining(String email);
}
```

❶ По соглашению в этом хранилище мы определяем настраиваемый метод поиска, возвращающий `Optional<Customer>`, содержащий либо `Customer`, либо ничего. Закулисно данный метод поиска превращается в запрос с использованием JPA-QL. Определенный запрос можно переопределить с помощью аннотации `@Query`.

Комплексные тесты

Располагая полученными объектами, можно убедиться в том, что все работает ожидаемым образом (пример 9.16).

Пример 9.16. Определение AccountApplicationTests

```
package service;

import com.mysql.jdbc.AssertionFailedException;
import demo.AccountApplication;
import demo.account.Account;
import demo.address.Address;
import demo.address.AddressType;
import demo.creditcard.CreditCard;
import demo.creditcard.CreditCardType;
import demo.customer.Customer;
import demo.customer.CustomerRepository;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.Collection;
import java.util.Optional;

import static demo.creditcard.CreditCardType.VISA;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = AccountApplication.class)
@ActiveProfiles(profiles = "test")
public class AccountApplicationTests {

 @Autowired
 private CustomerRepository customerRepository;

 @Test
 public void customerTest() {
  Account account = new Account("12345");
  Customer customer = new Customer("Jane", "Doe", "jane.doe@gmail.com",
    account);
  CreditCard creditCard = new CreditCard("1234567890", VISA);
  customer.getAccount().getCreditCards().add(creditCard);

  String street1 = "1600 Pennsylvania Ave NW";
  Address address = new Address(street1, null, "DC", "Washington",
   "United States", AddressType.SHIPPING, 20500);
```

```
  customer.getAccount().getAddresses().add(address);

  customer = customerRepository.save(customer);
  Customer persistedResult = customerRepository.findOne(customer.getId());
  Assert.assertNotNull(persistedResult.getAccount());    ❶
  Assert.assertNotNull(persistedResult.getCreatedAt());
  Assert.assertNotNull(persistedResult.getLastModified());  ❷

  Assert.assertTrue(persistedResult.getAccount().getAddresses().stream()
   .anyMatch(add -> add.getStreet1().equalsIgnoreCase(street1)));  ❸

  customerRepository.findByEmailContaining(customer.getEmail())  ❹
   .orElseThrow(
    () -> new AssertionFailedException(new RuntimeException(
     "there's supposed to be a matching record!")));
 }
}
```

❶ Мы успешно сохранили запись и отношение «один к одному».

❷ Мы успешно сохранили запись при сработавшем механизме аудита.

❸ Мы успешно сохранили запись и связь.

❹ Здесь при поиске всех объектов `Customer` с конкретным адресом электронной почты мы полагаемся на настраиваемый метод поиска.

Spring Data MongoDB

Проект *Spring Data MongoDB* предоставляет механизм управления данными на основе хранилища, предназначенный для MongoDB.

MongoDB — база данных NoSQL, хорошо известная простотой использования и упрощенной моделью данных. Она относится к категории *документоориентированных* баз данных, то есть записи в ней хранятся в виде иерархии JSON-*подобных* документов. К объекту MongoDB можно довольно легко подключиться, если добавить к пути к классам приложения зависимость `org.springframework.boot : spring-boot-starter-data-mongodb`. Указать на подключение к конкретному экземпляру позволят свойства `spring.data.mongodb.host`, `spring.data.mongodb.port` и `spring.data.mongodb.database`. В противном случае среда Spring Boot настроится на подключение к локальному экземпляру.

Сервис заказов Order

Spring Data MongoDB мы собираемся использовать для сервиса заказов `Order`, действующего в качестве сервиса, который работает на серверной стороне для контекста заказов в модели предметной области *Cloud Native Clothing*. Как и в предыдущем подразделе, нам предстоит создать приложение *Spring Data MongoDB*.

В этой работе будет встречаться многое из изученного при создании сервиса учетных записей `Account`.

Запуск datasource в виде встроенной базы данных, находящейся в оперативной памяти, осуществляемый при работе со Spring Data JPA и H2 или с HSQLDB, поддерживается не всеми проектами Spring Data. Поскольку код MongoDB написан на языке C++, то он не может быть встроен в JVM-приложение в виде процесса. При работе с кодом, который приводится в данном разделе, придется загрузить и установить MongoDB (https://www.mongodb.com/).

Создайте новый проект, воспользовавшись Spring Initializr, и не забудьте выбрать из зависимостей `org.springframework.boot : spring-boot-starter-data-mongodb`.

Для разработки сервиса `Order` вернемся к модели предметной области *Cloud Native Clothing*, чтобы рассмотреть контекст заказов `Order` (рис. 9.10).

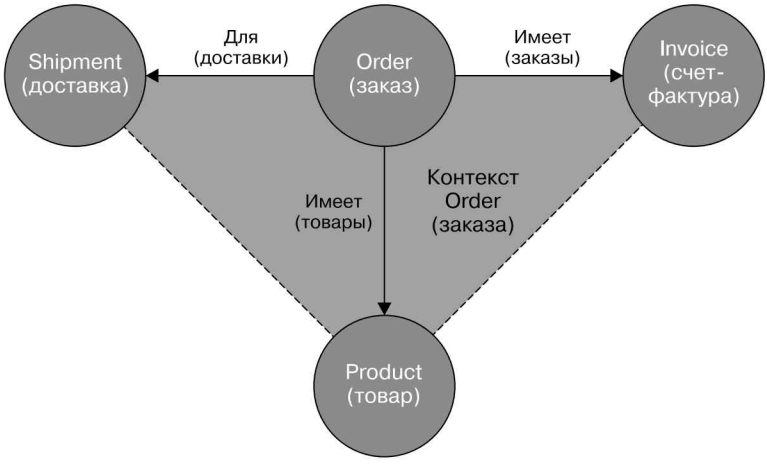


Рис. 9.10. Контекст заказов Order для сервиса Order

В контексте `Order` видны четыре класса предметных областей: `Invoice`, `Order`, `Product` и `Shipment`. Чтобы разобраться с созданием этих классов, посмотрим на основные варианты использования сервиса `Order`, показанные в табл. 9.3.

Таблица 9.3. Основные варианты использования сервиса Order

Классы предметных областей	Действие
Order, Product	Оформление заказа на набор товаров с указанием количества
Shipment, Address, Order	Доставка заказа по адресу
Invoice, Order, Account	Создание счета-фактуры для набора заказов, относящихся к учетной записи

В таблице представлен упрощенный набор вариантов использования по сравнению с имеющимся в реальном приложении. Этими тремя вариантами выражаются наиболее характерные действия сервиса `Order`. Обратите внимание: классы предметных областей взаимодействуют друг с другом. В предметно-ориентированном проектировании важно разбить варианты применения на описания действий субъектов в плане их взаимодействия с объектами предметных областей.

Классы документов, применяемые при использовании MongoDB

Ранее в данной главе мы уже говорили, что каждый класс предметной области в сервисе `Account` был снабжен JPA-аннотацией `@Entity`. Схема, используемая для аннотации классов предметных областей других проектов Spring Data, остается неизменной. Семантика, используемая для аннотаций каждого проекта Spring Data, отражает специфику модели базы данных. Применительно к Spring Data MongoDB для класса можно задействовать аннотацию `@Document`, указывающую на представление документа в MongoDB.

Первым создаваемым классом документа станет `Invoice`. В представленных основных вариантах использования имеется выражение, которое можно применить для описания возможных способов создания класса предметной области `Invoice`: «счет-фактура создается для набора заказов, относящихся к учетной записи».

Задача заключается в создании класса модели, поддерживающей действия варианта использования без введения излишних ограничений. При создании нового счета-фактуры в сервисе `Order` нужно, чтобы его можно было найти по номеру учетной записи, который, в свою очередь, будет ссылаться на класс предметной области `Account`, управляемый сервисом `Account`. Кроме того, для каждого счета-фактуры будет предоставлено поле не просто для отдельного заказа, а для набора заказов. Для `Invoice` потребуется и адрес выставления счета, по которому должен быть отправлен счет-фактура (примеры 9.17 и 9.18).

Причина использования документальной базы данных для сервиса заказов заключается в следующем: она является хранилищем составных данных. Ее практичность выражается в том, что позиции в заказе после его отправки пользователю не должны изменяться. Товар для каждой позиции нужно представить в виде состояния товара на время оформления заказа.

Пример 9.17. Класс предметной области Invoice, созданный в виде документа MongoDB

```
package demo.invoice;

import demo.address.Address;
import demo.address.AddressType;
```

```
import demo.data.BaseEntity;
import demo.order.Order;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

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

@Data
@AllArgsConstructor
@NoArgsConstructor
@Document
public class Invoice extends BaseEntity {

 @Id
 private String invoiceId;

 private String customerId;

 private List<Order> orders = new ArrayList<Order>();

 private Address billingAddress;

 private InvoiceStatus invoiceStatus;

 public Invoice(String customerId, Address billingAddress) {
  this.customerId = customerId;
  this.billingAddress = billingAddress;
  this.billingAddress.setAddressType(AddressType.BILLING);
  this.invoiceStatus = InvoiceStatus.CREATED;
 }

 public void addOrder(Order order) {
  order.setAccountNumber(this.customerId);
  orders.add(order);
 }
}
```

Пример 9.18. Перечень состояний для текущего статуса счета-фактуры Invoice

```
package demo.invoice;

public enum InvoiceStatus {
 CREATED, SENT, PAID
}
```

Чтобы упростить управление экземплярами документа, понадобится хранилище Spring Data MongoDB (пример 9.19).

Пример 9.19. Хранилище InvoiceRepository в комплекте с настраиваемым методом поиска

```
package demo.invoice;

import demo.address.Address;
import org.springframework.data.repository.PagingAndSortingRepository;

public interface InvoiceRepository extends
 PagingAndSortingRepository<Invoice, String> {

 Invoice findByBillingAddress(Address address);
}
```

Из класса предметной области `Invoice` можно увидеть, что в нем есть ссылка на набор классов `Order`. Вернемся также к основным вариантам использования, предоставляющим дополнительные сведения об `Order`:

- заказ оформляется для набора товаров и их количества;
- доставка выполняется по адресу заказа.

Эти два выражения описывают несколько взаимодействий классов предметных областей. Чтобы выполнить требования первого выражения, воспользуемся новым классом объектов `LineItem` (позиция) с целью описать заказанный товар `Product` и его количество. Каждая позиция `LineItem` будет определять удаленную ссылку на товар `Product`, хранящийся благодаря сервису `Inventory`, создание которого мы рассмотрим позже в данной главе. Второе выражение варианта применения — описание создания доставки `Shipment`, выполняемой по адресу `Address` заказа `Order`. Чтобы выполнить требования этого выражения, понадобится указывать номер учетной записи и адрес доставки в конструкторе класса `Order`. И наконец, нужно будет предоставить объект состояния, содержащий описание объекта `Order` по мере его перемещения от исходного статуса к финальному (примеры 9.20 и 9.21).

Пример 9.20. Класс предметной области Order

```
package demo.order;

import demo.address.Address;
import demo.address.AddressType;
import demo.data.BaseEntity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

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

@Data
@NoArgsConstructor
@AllArgsConstructor
@Document
```

```
public class Order extends BaseEntity {

 @Id
 private String orderId;

 private String accountNumber;

 private OrderStatus orderStatus;

 private List<LineItem> lineItems = new ArrayList<>();

 private Address shippingAddress;

 public Order(String accountNumber, Address shippingAddress) {
  this.accountNumber = accountNumber;
  this.shippingAddress = shippingAddress;
  this.shippingAddress.setAddressType(AddressType.SHIPPING);
  this.orderStatus = OrderStatus.PENDING;
 }

 public void addLineItem(LineItem lineItem) {
  this.lineItems.add(lineItem);
 }
}
```

Пример 9.21. Перечень состояний для текущего статуса заказа Order

```
package demo.order;

public enum OrderStatus {
 PENDING, CONFIRMED, SHIPPED, DELIVERED
}
```

А теперь создадим класс позиции `LineItem`, дающий описание ссылке на товар `Product` в каталоге и состоянию товара на время оформления заказа (пример 9.22).

Пример 9.22. Класс LineItem

```
package demo.order;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class LineItem {

 private String name, productId;

 private Integer quantity;

 private Double price, tax;
}
```

Аудит с использованием MongoDB

Аудит можно поддерживать с помощью MongoDB. В первую очередь для этого понадобится базовый объект, от которого другие объекты смогут унаследовать общие поля, применяемые для аудита, как показано в следующем листинге (пример 9.23).

Пример 9.23. Исходный базовый класс аудита

```
package demo.data;

import lombok.Data;
import org.joda.time.DateTime;

@Data
public class BaseEntity {

 private DateTime lastModified, createdAt;
}
```

Затем, как показано в примере 9.24, нужно сконфигурировать слушатель событий `AbstractMongoEventListener`, чтобы внести эти значения в изменения жизненного цикла объекта.

Пример 9.24. Для аудита транзакций MongoDB можно воспользоваться слушателем

```
package demo.data;

import org.joda.time.DateTime;
import
   org.springframework.data.mongodb.core.mapping.event.
AbstractMongoEventListener;
import org.springframework.data.mongodb.core.mapping.event.BeforeSaveEvent;
import org.springframework.stereotype.Component;

@Component
class BeforeSaveListener extends AbstractMongoEventListener<BaseEntity> {

 @Override
 public void onBeforeSave(BeforeSaveEvent<BaseEntity> event) {

  DateTime timestamp = new DateTime();

  if (event.getSource().getCreatedAt() == null)
   event.getSource().setCreatedAt(timestamp);

  event.getSource().setLastModified(timestamp);

  super.onBeforeSave(event);
 }
}
```

Комплексные тесты

После создания для сервиса `Order` уровня данных, перейдем к циклу создания комплексного теста (пример 9.25). В нем предполагается выполнение следующих действий:

- создание нового заказа `Order`;
- добавление к новому `Order` позиции `LineItem`;
- сохранение `Order` с использованием хранилища `OrderRepository`.

Пример 9.25. Комплексный тест для компонентов MongoDB

```
package orders;

import demo.OrderApplication;
import demo.address.Address;
import demo.invoice.Invoice;
import demo.invoice.InvoiceRepository;
import demo.order.LineItem;
import demo.order.Order;
import demo.order.OrderRepository;
import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = OrderApplication.class)
public class OrderApplicationTest {

  @Autowired
  private OrderRepository orderRepository;

 @Autowired
 private InvoiceRepository invoiceRepository;

 @Before
  @After
 public void reset() {
  orderRepository.deleteAll();
  invoiceRepository.deleteAll();
 }

 @Test
 public void orderTest() {

  Address shippingAddress = new Address("1600 Pennsylvania Ave NW", null, "DC",
```

```
    "Washington", "United States", 20500);
  Order order = new Order("12345", shippingAddress);
  order.addLineItem(new LineItem("Best. Cloud. Ever. (T-Shirt, Men's Large)",
   "SKU-24642", 1, 21.99, .06));
  order.addLineItem(new LineItem("Like a BOSH (T-Shirt, Women's Medium)",
   "SKU-34563", 3, 14.99, .06));
  order.addLineItem(new LineItem(
   "We're gonna need a bigger VM (T-Shirt, Women's Small)", "SKU-12464", 4,
   13.99, .06));
  order.addLineItem(new LineItem("cf push awesome (Hoodie, Men's Medium)",
   "SKU-64233", 2, 21.99, .06));
  order = orderRepository.save(order);
  ❶
  Assert.assertNotNull(order.getOrderId());
  Assert.assertEquals(order.getLineItems().size(), 4);

  ❷
  Assert.assertEquals(order.getLastModified(), order.getCreatedAt());
  order = orderRepository.save(order);
  Assert.assertNotEquals(order.getLastModified(), order.getCreatedAt());

  ❸
  Address billingAddress = new Address("875 Howard St", null, "CA",
   "San Francisco", "United States", 94103);
  String accountNumber = "918273465";

  Invoice invoice = new Invoice(accountNumber, billingAddress);
  invoice.addOrder(order);
  invoice = invoiceRepository.save(invoice);
  Assert.assertEquals(invoice.getOrders().size(), 1);

  ❹
  Assert.assertEquals(invoiceRepository.findByBillingAddress(billingAddress),
   invoice);
 }
}
```

❶ Сначала убеждаемся в том, что данные записаны в постоянное хранилище...

❷ ...затем убеждаемся в работоспособности аудита...

❸ ...после этого убеждаемся в работоспособности вложенных ассоциаций.

❹ Здесь используем настраиваемый метод поиска, который Spring Data MongoDB превращает в BSON-запрос, отправляемый в адрес документа MongoDBInvoice.

Spring Data Neo4j

Проект *Spring Data Neo4j* позволяет управлять данными на основе хранилища для Neo4j, популярной *графовой базы* данных NoSQL. Объекты в ней представлены в виде связных графов узлов и отношений. Последние могут содержать как простые, так и сложные типы значений. Графовая база данных основывается на прин-

ципах теории графов. Математическая база теории графов позволяет графовой базе легко обходить миллионы отношений в секунду и моделировать большие взаимосвязанные наборы данных. На первый взгляд может показаться: то же самое предлагает и *реляционная* база данных. Но это не так: попробуйте смоделировать граф друзей в Facebook в СУРБД, а затем выполнить левое внешнее соединение! В графе отношения между узлами так же важны, как и сами узлы. Отношение может иметь направление, быть однонаправленным или двунаправленным; специалист может рассматриваться в качестве «сотрудника» другого специалиста.

Польза от этой модели отношений проявляется в сочетании с Cypher, декларативным языком запросов Neo4j, похожим на SQL. В отличие от последнего Cypher позволяет выполнять запросы и управлять данными с помощью шаблонов, использующих строгую схему, которая должна быть определена до запроса данных. Кроме того, Neo4j предоставляет возможность определять обходы графа, изживая тем самым SQL-соединения, затрудняющие написание и чтение сложных запросов.

В данном разделе для создания сервиса запасов `Inventory` в рамках серверной поддержки интернет-магазина нашей условной компании *Cloud Native Clothing* будет использоваться Spring Data Neo4j. Для этого нужно достичь следующего:

- предоставить обзор требований *Cloud Native Clothing* для управления запасами товаров;
- ознакомиться с порядком управления объектами базы данных в виде связного графа в Neo4j;
- спроектировать модель данных графа для управления запасами;
- реализовать классы предметных областей Spring Data в виде узлов Neo4j;
- создать хранилища для управления классами предметных областей для узлов Neo4j.

Сервис Inventory

Создание сервиса `Inventory` мы собираемся разбить на две части: проектирование и реализацию. Нужно проработать способы создания модели данных графа, описывающей контекст запасов для интернет-магазина *Cloud Native Clothing*.

Рассмотрим такой пример. Магазину одежды следует иметь сезонный каталог, пригодный для изменения набора товаров, представленного на сайте в начале каждого квартала. Изменения не обязательно должны касаться всех товаров. В магазине могут быть товары и круглогодичной продажи. Поэтому каталоги могут иметь иерархию: допустим, основной каталог товаров, предлагаемый круглогодично, и сезонный, в котором линейка товаров меняется каждый квартал.

Нужно будет также подключать товары к запасам по коллекции складов, разбросанных по всему свету. Товар из каталога может иметься в каждом из этих складов. Можно отслеживать глобальный запас товаров и воспользоваться данной

информацией для создания доставок по кратчайшему маршруту. Короче говоря, это не простое дело. Для решения описанной задачи нужно справиться со сложностью модели связности, не затрачивая слишком много времени на ее разработку.

Neo4j позволяет гибко моделировать данные интуитивным образом; можно отобразить их на приложение Spring Data с помощью абстракций, уже рассмотренных в предыдущих разделах данной главы.

Контекст запасов, показанный на рис. 9.11, демонстрирует упрощенный взгляд на действия, необходимые для создания приложения. Чтобы оно удовлетворяло ранее рассмотренным требованиям, имея функции доставки и управления запасами, нужно будет создать графовую модель, позволяющую спроектировать классы предметных областей и хранилища. Начнем с обзора способов использования свойств графовой модели данных Neo4j для управления взаимосвязанными данными с помощью Spring Data Neo4j.

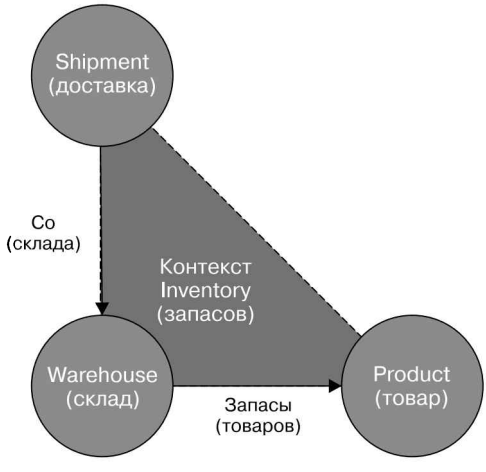


Рис. 9.11. Контекст для сервиса Inventory

Конфигурирование Neo4j

Среда Spring Boot позволяет проекту Spring Data Neo4j выполнять автоматическую конфигурацию. При этом можно воспользоваться набором функций, подстроенных под специфику графовой базы данных. Чтобы создать приложение Neo4J, в сборке Maven нужно указать зависимость `org.springframework.boot : spring-boot-starter-data-neo4j`. Приложению можно указать конкретный экземпляр Neo4J, задействовав любое свойство из `spring.data.neo4j.uri`, `spring.data.neo4j.username`, `spring.data.neo4j.password` и т. д. После того как создаваемый проект Spring Boot настроен на применение Spring Data Neo4j, появляется возможность создать приложение с помощью уже известных абстракций, которые были использованы в предыдущих примерах для управления взаимосвязанными данными в базе данных Neo4j.

Моделирование графа данных с помощью Neo4j

Моделирование графа данных носит куда более интуитивный характер, нежели моделирование данных предметных областей для СУРБД. Здесь мы рассмотрим порядок возможного применения свойств графовой модели данных Neo4j для создания схемы узлов и отношений, используемых для проектирования хранилищ и классов предметных областей Spring Data. Начнем с порядка управления объектами данных Neo4j как сущностями модели графа.

В Neo4j имеются две собственные сущности данных: *узел* и *отношение*. Узлы используются для хранения свойств в виде отображения «ключ — значение». Отношения позволяют хранить отображения свойств. Простые типы значений, например *цвет* или *пол*, могут храниться в узле в виде свойства. Сложные типы значений, такие как *адрес*, должны быть представлены в виде отдельного узла. Отношения служат для связи узлов друг с другом, скажем *адресного* узла с узлом *доставки*.

На рис. 9.12 показан узел доставки, связанный с двумя адресными. Тип отношения `HAS_ADDRESS` может быть связан с несколькими адресами.

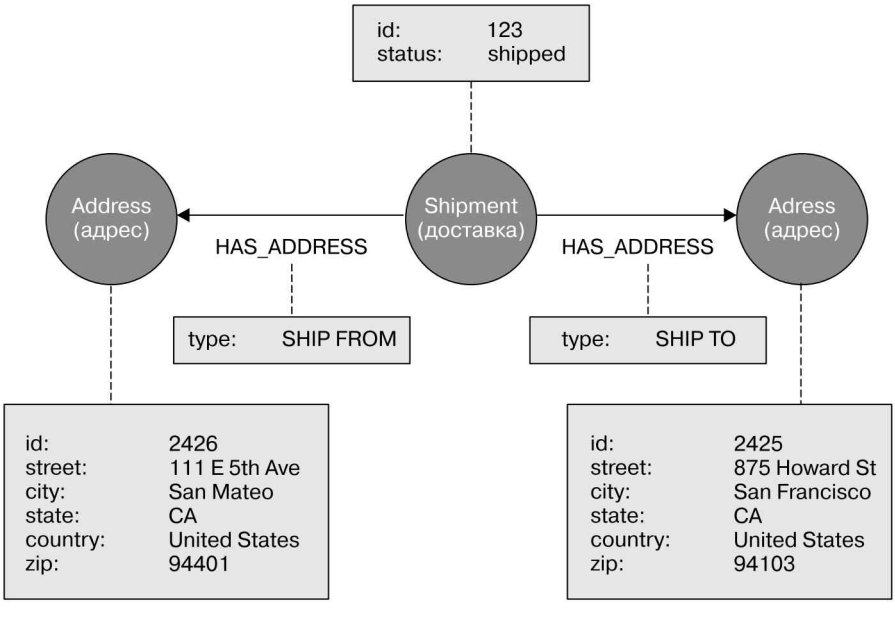


Рис. 9.12. Свойства в отношениях используются для классификации сложных типов значений узла

Используя Cypher, можно выразить запрос для получения каждого адреса данной доставки:

```
MATCH (shipment:Shipment)-[r:HAS_ADDRESS]->(address)
WHERE shipment.id = 123
RETURN address
```

В данном запросе сначала находится соответствие всем узлам доставки, которым дано описание (`shipment:Shipment`). В Cypher для узлов и отношений используется ASCII-графика, позволяющая описывать сущности графа в виде схем. Узел заключается в круглые скобки, а двоеточие отделяет переменную, в которой Cypher должен сохранить результаты (допустим, (`shipment`)), от названия, дающего описание той группы, к которой принадлежит узел (например, (`:Shipment`)), где указывается, что последний имеет название `Shipment` (Доставка).

Если бы мы воспользовались вышеприведенным запросом в отношении данных, показанных на рис. 9.12, то получили бы следующие результаты.

```
Detailed Query Results

Query Results

+-------------------------------------------------------------------------------------------------------------------+
| address                                                                                                           |
+-------------------------------------------------------------------------------------------------------------------+
| Node[7]{street:"875 Howard St",country:"United States",id:2425,city:"San Francisco",zip:"94103",state:"CA"} |
| Node[6]{street:"111 E 5th Ave",country:"United States",id:2426,city:"San Mateo",zip:"94401",state:"CA"}     |
+-------------------------------------------------------------------------------------------------------------------+
2 rows
54 ms
```

Как видите, для конкретной доставки с идентификатором 123 получены оба адреса. А что делать, если нужно запросить лишь тот адрес, с которого была произведена доставка? Cypher-запрос можно заменить следующим:

```
MATCH (shipment:Shipment)-[r:HAS_ADDRESS { type: "SHIP_FROM" }]->(address)
WHERE shipment.id = 123
RETURN address
```

Теперь, как показано ниже, из двух адресов будет получен только один.

```
Detailed Query Results

Query Results

+-------------------------------------------------------------------------------------------------------------------+
| address                                                                                                           |
+-------------------------------------------------------------------------------------------------------------------+
| Node[6]{street:"111 E 5th Ave",country:"United States",id:2426,city:"San Mateo",zip:"94401",state:"CA"} |
+-------------------------------------------------------------------------------------------------------------------+
1 row
71 ms
```

Изменив первый Cypher-запрос путем включения в него указания на тип свойства отношения `HAS_ADDRESS`, `()-[r:HAS_ADDRESS { type: "SHIP_FROM" }]->()`, который должен быть эквивалентен `SHIP_FROM`, мы получили возможность извлечь адресную информацию, касающуюся места, откуда пришла доставка.

Узнав, как Neo4j моделирует данные в виде графов, можно создать модель графа, дающую описание узлов и отношений микросервиса запасов.

На рис. 9.13 показаны каждый узел с его названием и отношения между узлами, что позволяет получить примерное представление о том, как в приложении Spring Boot спроектировать классы предметных областей и хранилища с помощью Spring Data Neo4j.

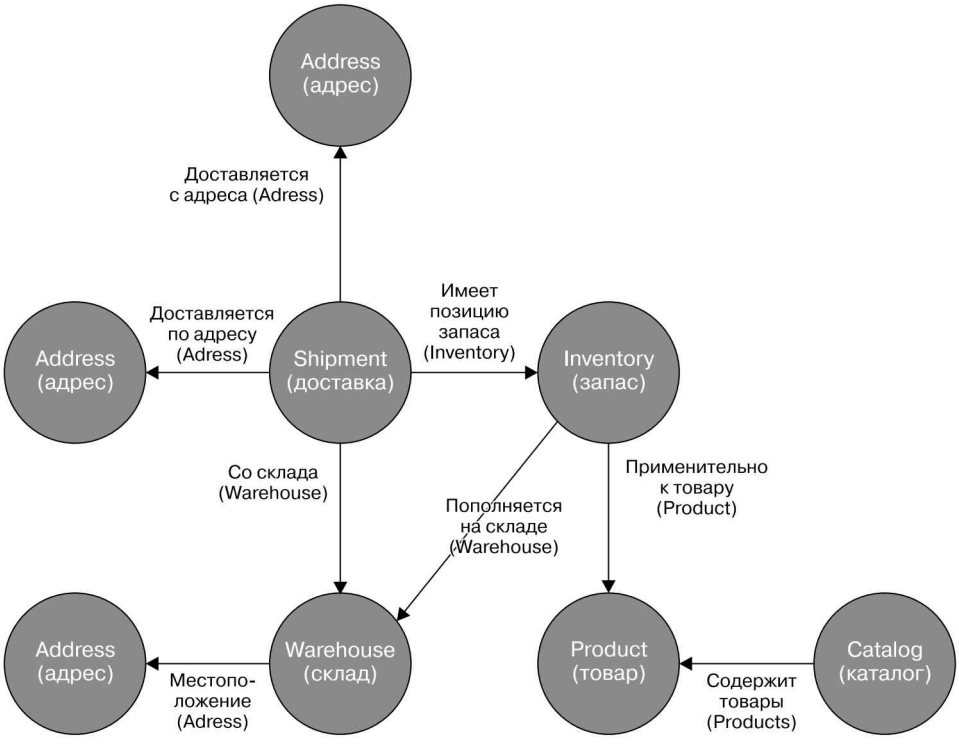


Рис. 9.13. Модель графа для ограниченного контекста запасов

В Spring Data Neo4j названия в модели графа Neo4j, подобные названиям узлов на рис. 9.13, станут именами наших хранилищ Spring Data. У нас будут хранилища для каждого из следующих названий, показанных на модели графа данных:

- Address;
- Shipment;
- Inventory;
- Product;
- Catalog;
- Warehouse.

Нужно создать классы предметных областей в их собственных соответствующих пакетах для каждого названия узла, представленного на рис. 9.13. В примере 9.26 показан первый класс, предназначенный для моделирования адреса `Address`.

Пример 9.26. Класс предметной области Address на основе названия, используемого в Neo4j

```
package demo.address;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;

@Data
@NoArgsConstructor
@AllArgsConstructor
@NodeEntity
public class Address {

 @GraphId
 private Long id;

 private String street1, street2, state, city, country;

 private Integer zipCode;

 public Address(String street1, String street2, String state, String city,
  String country, Integer zipCode) {
  this.street1 = street1;
  this.street2 = street2;
  this.state = state;
  this.city = city;
  this.country = country;
  this.zipCode = zipCode;
 }
}
```

Класс `Product`, показанный в примере 9.27, моделирует то, что можно продать, цену за единицу товара и его идентификатор (в некоторых частях мира встречается название SKU).

Пример 9.27. Класс предметной области Product на основе названия, используемого в Neo4j

```
package demo.product;

import demo.catalog.Catalog;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;

import java.util.HashSet;
```

```
import java.util.Set;

@Data
@NoArgsConstructor
@AllArgsConstructor
@NodeEntity

public class Product {

 @GraphId
 private Long id;

 private String name, productId;

 private Double unitPrice;

 public Product(String name, String productId, Double unitPrice) {
  this.name = name;
  this.productId = productId;
  this.unitPrice = unitPrice;
 }

}
```

Класс `Warehouse`, показанный в примере 9.28, моделирует логическое название для структуры, хранящей физические товары, и ее расположение.

Пример 9.28. Класс предметной области Warehouse на основе названия, используемого в Neo4j

```
package demo.warehouse;

import demo.address.Address;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;

@NoArgsConstructor
@AllArgsConstructor
@Data
@NodeEntity
public class Warehouse {

 @GraphId
 private Long id;

 private String name;

 @Relationship(type = "HAS_ADDRESS")
```

```
 private Address address;

 public Warehouse(String n, Address a) {
  this.name = n;
  this.address = a;
 }

 public Warehouse(String name) {
  this.name = name;
 }
}
```

Класс `Inventory`, показанный в примере 9.29, описывает количество заданного товара, имеющегося на заданном складе (или запаса).

Пример 9.29. Класс предметной области Inventory на основе названия, используемого в Neo4j

```
package demo.inventory;

import demo.product.Product;
import demo.warehouse.Warehouse;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;

@Data
@NoArgsConstructor
@AllArgsConstructor
@NodeEntity
public class Inventory {

 @GraphId
 private Long id;

 private String inventoryNumber;

 @Relationship(type = "PRODUCT_TYPE", direction = "OUTGOING")
 private Product product;

 @Relationship(type = "STOCKED_IN", direction = "OUTGOING")
 private Warehouse warehouse;

 private InventoryStatus status;

 public Inventory(String inventoryNumber, Product product, Warehouse warehouse,
  InventoryStatus status) {
  this.inventoryNumber = inventoryNumber;
  this.product = product;
  this.warehouse = warehouse;
```

```
  this.status = status;
 }

}
```

Класс `Shipment`, показанный в примере 9.30, моделирует информацию из набора товарных запасов на доставку по адресу `Address`.

Пример 9.30. Класс предметной области Shipment на основе названия, используемого в Neo4j

```
package demo.shipment;

import demo.address.Address;
import demo.inventory.Inventory;
import demo.warehouse.Warehouse;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;

import java.util.HashSet;
import java.util.Set;

@Data
@NoArgsConstructor
@AllArgsConstructor
@NodeEntity
public class Shipment {

 @GraphId
 private Long id;

 @Relationship(type = "CONTAINS_PRODUCT")
 private Set<Inventory> inventories = new HashSet<>();

 @Relationship(type = "SHIP_TO")
 private Address deliveryAddress;

 @Relationship(type = "SHIP_FROM")
 private Warehouse fromWarehouse;

 private ShipmentStatus shipmentStatus;

 public Shipment(Set<Inventory> inventories, Address deliveryAddress,
  Warehouse fromWarehouse, ShipmentStatus shipmentStatus) {
  this.inventories = inventories;
  this.deliveryAddress = deliveryAddress;
  this.fromWarehouse = fromWarehouse;
  this.shipmentStatus = shipmentStatus;
 }
}
```

Класс `Catalog`, показанный в примере 9.31, является перечислением экземпляров класса `Product`, доступных для группировки под каким-либо (возможно, произвольным) именем группы.

Пример 9.31. Класс предметной области Catalog на основе названия, используемого в Neo4j

```
package demo.catalog;

import demo.product.Product;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;

import java.util.Collection;
import java.util.HashSet;
import java.util.Set;
import java.util.stream.Collector;

@Data
@NoArgsConstructor
@AllArgsConstructor
@NodeEntity
public class Catalog {

 @GraphId
 private Long id;

 @Relationship(type = "HAS_PRODUCT", direction = Relationship.OUTGOING)
 private Set<Product> products = new HashSet<>();

 private String name;

 public Catalog(String n, Collection<Product> p) {
  this.name = n;
  this.products.addAll(p);
 }

 public Catalog(String name) {
  this.name = name;
 }

}
```

Для хранилищ в Spring Data Neo4j служат те же абстракции, которые мы применяли в предыдущих примерах данной главы. Как показано в примере 9.32, в сервисе запасов мы опять используем абстракцию `PagingAndSortingRepository`, чтобы указать на желание получить от Spring Data хранилище со встроенным разбиением на страницы и функцией сортировки.

Пример 9.32. Хранилище с разбиением на страницы и сортировкой для названия Address в Neo4j

```
package demo.address;

import org.springframework.data.neo4j.repository.GraphRepository;

public interface AddressRepository extends GraphRepository<Address> {
}
```

В каждом пакете, содержащем класс предметной области из модели графа, создастся хранилище, позволяющее управлять данными этой области. Так, фрагмент кода из примера 9.32 — определение хранилища, которое будет находиться в принадлежащем проекту пакете `address`. Хранилище `AddressRepository` позволит управлять данными для узлов в Neo4j с названием `Address`.

Теперь структура пакетов для проекта Inventory Service (сервис запасов) должна приобрести вид, показанный на рис. 9.14.

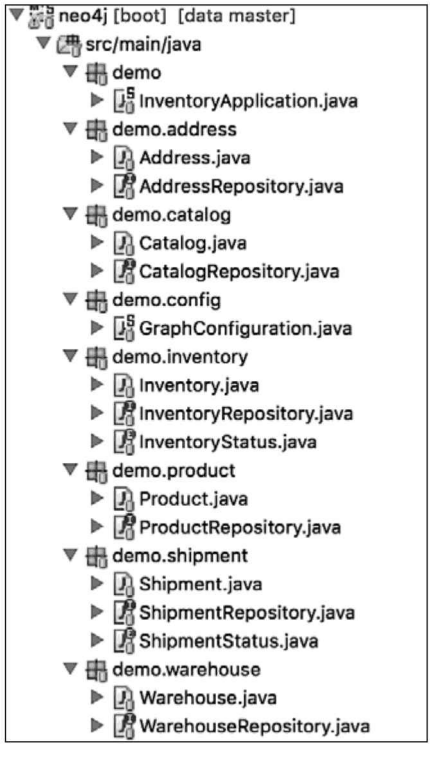


Рис. 9.14. Структура пакетов сервиса запасов

Комплексные тесты

Создав уровень данных для сервиса запасов, займемся разработкой простого сквозного комплексного теста, в котором предпримем действия, показанные в примере 9.33, в результате чего будут созданы:

- склад;
- перечень товаров;
- каталог;
- адреса доставки;
- запасы товаров;
- доставка товаров для пополнения запасов.

Пример 9.33. Комплексный тест для графа Neo4j

```
package demo;

import demo.address.Address;
import demo.address.AddressRepository;
import demo.catalog.Catalog;
import demo.catalog.CatalogRepository;
import demo.inventory.Inventory;
import demo.inventory.InventoryRepository;
import demo.product.Product;
```

```
import demo.product.ProductRepository;
import demo.shipment.Shipment;
import demo.shipment.ShipmentRepository;
import demo.shipment.ShipmentStatus;
import demo.warehouse.Warehouse;
import demo.warehouse.WarehouseRepository;
import org.apache.commons.lang3.exception.ExceptionUtils;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.neo4j.ogm.session.Session;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.*;
import java.util.stream.Collectors;
import java.util.stream.Stream;

import static demo.inventory.InventoryStatus.*;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = InventoryApplication.class)
public class InventoryApplicationTests {

 @Autowired
 private ProductRepository products;

 @Autowired
 private ShipmentRepository shipments;

 @Autowired
 private WarehouseRepository warehouses;

 @Autowired
 private AddressRepository addresses;

 @Autowired
 private CatalogRepository catalogs;

 @Autowired
 private InventoryRepository inventories;

 @Autowired
 private Session session;

 @Before
 public void setup() {
  try {
   this.session.query("MATCH (n) OPTIONAL MATCH (n)-[r]-() DELETE n, r;",
    Collections.emptyMap()).queryResults();
```

```
  }
  catch (Exception e) {
   Assert.fail("can't connect to Neo4j! " + ExceptionUtils.getMessage(e));
  }
 }

 @Test
 public void inventoryTest() {

  ❶
  List<Product> products = Stream
   .of(
    new Product("Best. Cloud. Ever. (T-Shirt, Men's Large)", "SKU-24642", 21.99),
    new Product("Like a BOSH (T-Shirt, Women's Medium)", "SKU-34563", 14.99),
    new Product("We're gonna need a bigger VM (T-Shirt, Women's Small)",
     "SKU-12464", 13.99),
    new Product("cf push awesome (Hoodie, Men's Medium)", "SKU-64233", 21.99))
   .map(p -> this.products.save(p)).collect(Collectors.toList());

  Product sample = products.get(0);
  Assert.assertEquals(this.products.findOne(sample.getId()).getUnitPrice(),
   sample.getUnitPrice());

  ❷
  this.catalogs.save(new Catalog("Spring Catalog", products));

  ❸
  Address warehouseAddress = this.addresses.save(new Address("875 Howard St",
   null, "CA", "San Francisco", "United States", 94103));
  Address shipToAddress = this.addresses.save(new Address(
   "1600 Amphitheatre Parkway", null, "CA", "Mountain View", "United States",
   94043));

  ❹
  Warehouse warehouse = this.warehouses.save(new Warehouse("Pivotal SF",
   warehouseAddress));
  Set<Inventory> inventories = products
   .stream()
   .map(
    p -> this.inventories.save(new Inventory(UUID.randomUUID().toString(), p,
     warehouse, IN_STOCK))).collect(Collectors.toSet());
  Shipment shipment = shipments.save(new Shipment(inventories, shipToAddress,
   warehouse, ShipmentStatus.SHIPPED));
  Assert.assertEquals(shipment.getInventories().size(), inventories.size());
 }
}
```

❶ Сохранение ряда товаров...

❷ ...создание каталога...

❸ ...создание ряда адресов доставки...

❹ ...и создание доставки.

Spring Data Redis

Проект Spring Data Redis позволяет среде Spring интегрировать компонент для Redis-хранилища на основе использования пар «ключ — значение». Redis — база данных NoSQL с открытым кодом и одно из наиболее популярных на сегодняшний день хранилищ типа «ключ — значение». Хотя Redis относят именно к этому типу, лучше всего ее суть передает понятие *хранилища структуры данных*, организованного в оперативной памяти. Redis отличается от большинства других хранилищ данных тем, что позволяет хранить в качестве значений различные типы сложных структур данных. Наибольшую привлекательность Redis придает ее возможность предоставления различных наборов операций для каждой поддерживаемой разновидности структуры данных.

Redis поддерживает такие структуры данных:

- строки;
- списки;
- наборы;
- хеши;
- отсортированные наборы;
- битовые массивы и HyperLogLogs.

Преимущество распределенного хранилища данных, созданного для операций над сложными структурами данных, состоит в следующем: несколько процессов, приложений и серверов получают возможность параллельно взаимодействовать со значениями с одним и тем же ключом. Как правило, подобная работа из нескольких приложений, прежде чем можно будет оперировать значениями хранилища, потребует определенной формы десериализации в структуры данных, поддерживаемые языком (например, в список или набор).

В Redis данная проблема решается путем предоставления программного доступа к выполнению операций над поддерживаемыми структурами данных в виде API. Это значит, что такие операции могут применяться в атомарном режиме с более высокой степенью транзакционной детализации, нежели при использовании сериализации.

Redis зачастую также используется для обмена данными и сообщениями между процессами. Помимо специализированной поддержки операций над структурами данных, Redis может выступать в роли брокера сообщений, поддерживающего публикацию сообщений и подписку на них.

Кэширование. Это самый популярный вариант применения Redis. В компоненте Spring Data Redis реализуется принадлежащая среде Spring абстракция `CacheManager`, которая делает его весьма подходящим выбором для централизованного кэширования записей в архитектуре микросервисов. В данном пункте мы намерены исследовать приложение Spring Boot, задействующее Spring Data Redis для управления кэшированием пользовательских записей.

Как показано на рис. 9.15, сервис `User` (User Service) — приложение Spring Boot, отвечающее за управление ресурсами для класса предметной области `User`. Там же показано веб-приложение `User` (User Web), которое является приложением Spring Boot, использует сервис `User` и играет роль веб-приложения, зависящего от ресурса `User`, управляемого сервисом `User`.

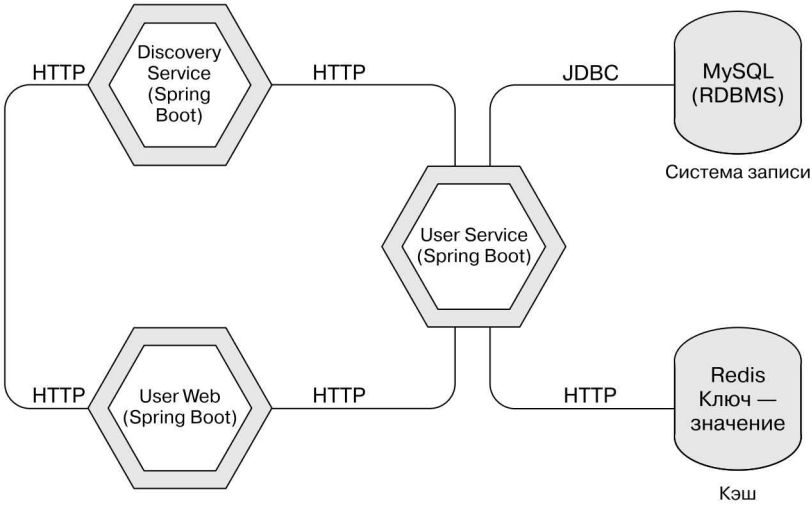


Рис. 9.15. Схема архитектуры сервиса User и веб-приложений User

Сервис User предоставляет REST API, позволяющий потребителям осуществлять удаленное управление ресурсами `User` через HTTP. Приложение User Web возьмет на себя роль клиентского приложения, содержащего одностраничное приложение для управления пользователями. Приложение Spring Boot будет размещать статический код HTML/CSS и код JavaScript и применять REST API, предоставляемый сервисом `User`.

На рис. 9.15 слева приведены сервисы без сохранения состояния, не имеющие зависимости от подключенной базы данных. В центре показан сервис, имеющий одно или несколько подключений к поставщику базы данных и управляющий только сохранением набора ресурсов, в данном случае ресурса User.

При добавлении к приложениям Spring Boot уровня кэширования следует рассматривать множество разных сценариев возможного использования данных. Кэширование — важная часть любой архитектуры микросервисов в связи с большой востребованностью взаимодействия с ресурсами для каждого сервиса, располагающего REST API.

Из рис. 9.15 следует, что у нас два хранилища данных: представленное базой данных MySQL и еще одно, являющееся сервером Redis. Записи станут сохраняться

и помещаться для постоянного хранения на диск с сервиса `User` в подключенную к нему базу данных MySQL. Когда запись запрашивается из другого сервиса, ответ будет кэширован путем репликации записи в сервере Redis, где она будет доступна в оперативной памяти и станет поставляться потребителям в течение предписанного срока годности, прежде чем пройдет повторное кэширование.

Цель кэширования — повышение производительности и разгрузка базы данных за счет сокращения обращений к ней и уменьшения объема использования пула подключений к первичному хранилищу данных с помощью вторичного высокодоступного хранилища. Рассмотрим сценарий лимитированного количества подключений к базе данных MySQL. Если превысить их допустимое количество, то база может зависнуть или стать недоступной, пока не произойдет переподключение и высвобождение пула. Для решения данной проблемы можно применить вторичное хранилище, записи в котором хранятся в памяти. Это позволяет обслуживать основную часть трафика за счет записей, находящихся в кэш-памяти, экономя на количестве подключений к базе данных и обращений к первичному хранилищу данных. Это особенно важно в архитектуре микросервисов, где ресурсы должны не только эффективно предоставляться потребителям, но и подвергаться длительному хранению на диске.

Среда Spring Boot также сработает в наших интересах и возьмет на себя автоматическое конфигурирование `RedisTemplate` на основе значений конфигурации (`spring.redis.host`, `spring.redis.port` и т. д.), указанных в среде окружения `Environment` при наличии в пути к классам зависимости `org.springframework.boot : spring-boot-starter-data-redis`. Если же добавить к классу `@Configuration` аннотацию `@EnableCaching`, то среда пойдет еще дальше и выполнит конфигурирование диспетчера `CacheManager`, имеющегося в проекте Spring Cache.

Подберите подходящее время истечения срока актуальности записей при конфигурировании ресурсов в архитектуре микросервисов. Когда в архитектуре довольно много приложений Spring Boot, могут складываться ситуации, при которых записи редко обновляются, но часто запрашиваются. В таком случае для подобных объектов срок истечения актуальности нужно увеличить. А для часто запрашиваемых ресурсов, подвергаемых активному обновлению, срок истечения актуальности нужно сократить, особенно если записи изменяются без удаления из кэша.

Наш пример представлен весьма типичным сервисом на основе применения Spring Data JPA, остро нуждающимся в кэшировании данных. Для выборочного кэширования или *запоминания* можно воспользоваться конкретными вызовами кэш-абстракции на основе Spring Data Redis.

В примере 9.34 мы уделим внимание классу `UserService`, в котором реализуется логика для каждого метода REST API, обслуживающего ресурс `User` из

UserController. В этом классе будут также содержаться наши аннотации кэширования для управления вставкой, извлечением и удалением записей User, направляемых из MySQL в Redis.

Пример 9.34. UserService управляет записями User с помощью операций с кэшем, управляемых аннотациями

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

@Service
public class UserService {

 private final UserRepository userRepository;

 @Autowired
 public UserService(UserRepository userRepository) {
  this.userRepository = userRepository;
 }

 @CacheEvict(value = "user", key = "#user.getId()")
 ❶
 public User createUser(User user) {

  User result = null;

  if (!userRepository.exists(user.getId())) {
   result = this.userRepository.save(user);
  }

  return result;
 }

 @Cacheable(value = "user")
 ❷
 public User getUser(String id) {
  return this.userRepository.findOne(id);
 }

 @CachePut(value = "user", key = "#id")
 ❸
 public User updateUser(String id, User user) {

  User result = null;

  if (userRepository.exists(user.getId())) {
```

```
    result = this.userRepository.save(user);
   }

   return result;
  }

  @CacheEvict(value = "user", key = "#id")
  ❹
  public boolean deleteUser(String id) {

   boolean deleted = false;

   if (userRepository.exists(id)) {
    this.userRepository.delete(id);
    deleted = true;
   }
   return deleted;
  }
}
```

❶ Извлечение из кэша Redis записи `User` с предоставленным идентификатором.

❷ Получение кэшированной записи из Redis или помещение записи `User` из MySQL в кэш Redis.

❸ Извлечение записи `User` и замена ее только что обновленной записью `User`.

❹ Извлечение из кэша Redis записи `User` с предоставленным идентификатором.

В примере 9.34 мы располагаем классом `UserService`, ответственным за выполнение операций с кэш-памятью, а также за управление записями базы данных для объекта класса `User` в MySQL. Этот сервис для управления жизненным циклом реплицированной записи в Redis использует кэширование, управляемое аннотациями. В данном классе имеются четыре весьма простые CRUD-операции. Первичный ключ записи `User` служит в качестве ключа, созданного для диспетчера кэша, предоставленного для Redis. Для доступа к ключу из параметров методов в аннотациях кэширования применяется язык Spring Expression Language (SpEL).

При совместном использовании сервера Redis в качестве общей кэш-памяти для различных сервисов важно помнить: создание ключа не будет охватывать пространство имен, содержащее имя сервиса. То есть наличие одного и того же идентификатора для различных понятий предметных областей, являющихся ресурсами, управляемыми разными сервисами, вызовет конфликтную ситуацию, приводящую к замене одного значимого ресурса другим. В таком случае на первый план может выйти сервис, возвращающий запись Account, притом что запрос делался на получение записи User. При совместном использовании кэшей всегда нужно проявлять осмотрительность. Следует создавать ключи, применяющие уникальное пространство имен для сервиса, владеющего кэшированной записью.

Redis может служить основой для кэширования для любого количества сервисов. В данном примере рассматривался вопрос кэширования, но Redis будет уместен и для HTTP-сессий, рассмотренных в главе 5 (подраздел «HTTP-сессии со Spring Session» раздела «Незначительная реструктуризация для перемещения вашего приложения» главы 5), где речь шла о миграции устаревших приложений в облачную среду путем изменения порядка использования ими HTTP-сессий для применения Spring Session.

Резюме

В данной главе мы рассмотрели приемы создания приложений Spring Boot, использующих различные проекты Spring Data. Мы создали полноценную серверную поддержку для интернет-магазина *Cloud Native Clothing*, применяя различные проекты Spring Data для каждого из следующих приложений:

- сервиса `Account` (JPA — MySQL);
- сервиса `Order` (MongoDB);
- сервиса `Inventory` (Neo4j).

Кроме того, мы обсудили возможности кэширования результатов, получаемых от сервисов на основе одного из этих источников данных, с помощью Spring Data Redis.

10 Рассылка сообщений

Рассылка сообщений поддерживает подключение публикации событий приложения к сервисам между процессами и сетями. Она применяется довольно широко, и многие варианты использования упомянуты в публикации в блоге Мартина Фаулера What do you mean by «Event-Driven»? («Что следует понимать под термином “управляемый событиями”?») (https://martinfowler.com/articles/201701-event-driven.html).

- **Уведомления о событиях:** одна система отправляет сообщения о событиях, чтобы уведомить другие системы об изменениях в предметной области. При этом не ожидается создания какого-либо ответа от получателя сообщения. Исходная система его не ожидает и не использует. Уведомление о событии неизменно, а значит, содержимое сообщения не должно включать данные, измененные после выдачи события.
- **Передача состояния за счет переноса события:** в этих событиях не содержатся данные, требующие от получателя обратного вызова в адрес исходной системы. Данная форма сообщений будет включать все, что необходимо получателю для обработки события.
- **Порождение событий:** включает сохранение журнала событий предметных областей, вызывающих со временем изменение состояния системы. В данном сценарии события могут воспроизводиться с любого момента времени для воссоздания текущего состояния системы.

Такие поставщики сообщений, как Apache Kafka, RabbitMQ, ActiveMQ или MQSeries, действуют в качестве хранилища и концентратора сообщений. Источник и потребитель подключаются к поставщику сообщений, а не друг к другу. Обычно традиционные поставщики весьма статичны, но предложения, подобные RabbitMQ и Apache Kafka, могут удовлетворять потребностям по масштабированию. Системы рассылки сообщений не допускают привязку источника к потребителю. Источнику или потребителю не нужно совместно использовать одно и то же место, находиться в одном и том же месте или процессе или даже быть доступными в одно и то же время. Важность этих качеств *возрастает* в облачной среде, где сервисы характеризуются размытостью и эфемерностью. Облачные платформы поддерживают

адаптируемость — они могут разрастаться и ужиматься по мере необходимости. Системы рассылки сообщений предоставляют идеальный способ для усечения нагрузки при снижении активности систем и для балансировки нагрузки, когда требуется провести масштабирование.

При всей важности использования таких технологий, как RabbitMQ и Apache Kafka, имейте в виду: они повышают рабочие издержки. В идеале решением этих задач должна заниматься платформа, причем в автоматическом режиме. Если применяется платформа Cloud Foundry, то в ее каталоге уже присутствуют сервисы, подобные RabbitMQ.

Архитектуры, управляемые событиями со Spring Integration

До сих пор мы рассматривали обработку данных, инициированную нами и по нашему плану. Но происходящее в этом мире не поддается планированию: нас окружают события, управляющие всеми нашими действиями. При всей ценности логического распределения данных по окнам некоторые данные просто не могут рассматриваться в понятиях окон. Отдельные данные имеют непрерывный характер и связаны с событиями в реальном мире. В данном разделе мы обсудим приемы работы с данными, управляемыми событиями.

Наверное, большинство из нас, представляя события, думают о технологиях рассылки сообщений, таких как JMS, RabbitMQ, Apache Kafka, Tibco Rendezvous или IBM MQSeries. Они позволяют создавать подключения различных автономных клиентов через отправку сообщений к централизованному посреднику, во многом похожие на то, как электронная почта дает людям возможность контактировать друг с другом. Поставщик сообщений сохраняет доставленные сообщения до тех пор, пока клиент не сможет воспользоваться ими и отреагировать на них (что во многом похоже на роль ящика электронной почты).

В большинство из этих технологий входит API и полезный клиент, которым можно воспользоваться. В Spring всегда имелась хорошая низкоуровневая поддержка для технологий рассылки сообщений, а также совершенная низкоуровневая поддержка для JMS API, AMQP (и поставщиков наподобие RabbitMQ), Redis и Apache Geode.

Разумеется, в мире множество типов событий. Одним из примеров является получение электронной почты. Еще одним — получение сообщения в «Твиттере». Новый файл в каталоге? Это тоже событие. Сообщение в чате с XMPP-поддержкой? Событие. Микроволновая печь с поддержкой MQTT-протокола отправляет обновление состояния? И это тоже событие.

Поразительное разнообразие! Если попытаться осмыслить пространство различных источников событий, то вы, надеемся, сможете увидеть массу возможностей *и* трудностей. Отчасти сложность обусловливается самой интеграцией. Как создается система, которая зависит от событий, поступающих от всего многообразия других систем? Интеграцию можно представить в понятиях двухточечной связи между различными источниками событий... Но в конечном итоге это приведет к *архитектуре типа спагетти*. Данная идея ущербна и с математической точки зрения, поскольку каждая точка интеграции нуждается в подключении к другой такой точке. Это биномальный коэффициент: $n(n - 1) / 2$. Получается, что для шести сервисов понадобятся 15 различных двухточечных связей!

Вместо этого мы обратимся к более структурированному интеграционному подходу с помощью Spring Integration. В основу этого фреймворка заложены имеющиеся в среде Spring типы `MessageChannel` и `Message<T>`. Объект `Message<T>` обладает информационным наполнением и набором заголовков, предоставляющим метаданные о самом информационном наполнении сообщения. `MessageChannel` похож на `java.util.Queue`, а объекты `Message<T>` проходят через экземпляры `MessageChannel`.

Spring Integration поддерживает интеграцию сервисов и данных между несколькими в иных условиях несовместимыми системами. Концептуально составление интеграционного потока похоже на составление потока из каналов и фильтров в операционной системе UNIX с помощью `stdin` и `stdout` (пример 10.1).

Пример 10.1. Использование модели каналов и фильтров для соединения утилит командной строки, изначально сконцентрированных на одной задаче

```
cat input.txt | grep ERROR | wc -l > output.txt
```

Здесь данные берутся из источника (файла `input.txt`), проходят по каналу к команде `grep` для фильтрации результатов и сохранения только строк с пометкой `ERROR`. Затем они передаются по каналу утилите `wc` для подсчета количества строк. Далее получившееся количество записывается в выходной файл `output.txt`. Используемые компоненты — `cat`, `grep` и `wc` — ничего друг о друге не знают. Они создавались автономно. Им известно, только как считывать данные из стандартного устройства ввода `stdin` и записывать их в стандартное устройство вывода `stdout`. Такая нормализация данных существенно упрощает составление сложных решений из простых элементов. В данном примере команда `cat` превращает данные файла в данные, доступные для считывания любому процессу, разбирающемуся в `stdin`. Тем самым происходит *приспособление* входящих данных к нормализованному формату: строкам из строковых величин в `stdout`. В конце оператор перенаправления (`>`) превращает нормализованные данные, строки из строковых величин, в данные файловой системы. Он *приспосабливает* их. Знак канала (`|`) используется, чтобы показать переход выхода одного компонента во вход другого компонента.

Поток Spring Integration работает аналогичным образом: данные проходят нормализацию в экземпляры `Message<T>`. У каждого объекта `Message<T>` имеется

информационное наполнение и заголовки — метаданные об информационном наполнении в `Map<K,V>`, — ввод и вывод различных компонентов рассылки сообщений. Как правило, эти компоненты предоставляет Spring Integration, но можно без особого труда создавать и использовать собственные. Существуют различные компоненты рассылки сообщений, поддерживающие все схемы Enterprise Application Integration (http://www.enterpriseintegrationpatterns.com/) (фильтры, маршрутизаторы, преобразователи, адаптеры, шлюзы и т. д.). Принадлежащий среде Spring канал `MessageChannel` является поименованным, через него объекты `Message<T>` проходят между компонентами рассылки сообщений. Это каналы, и в исходном состоянии они работают наподобие `java.util.Queue`. Данные входят и выходят.

Конечные точки рассылки сообщений

Объекты `MessageChannel` соединяются с помощью конечных точек рассылки сообщений, то есть через Java-объекты, проделывающие с сообщениями различные действия. Spring Integration будет все делать правильно, если предоставить ему `Message<T>`, или просто `T` для различных компонентов. Фреймворк обеспечивает доступную для использования модель компонентов, или Java DSL. Каждая конечная точка рассылки сообщений в потоке Spring Integraton может создавать значение на выходе, которое затем отправляется вниз по маршруту, или `null`, что приводит к прекращению обработки.

Шлюзы на входе принимают входящие запросы от внешних систем, обрабатывают их как `Message<T>` и отправляют ответ. Шлюзы на выходе принимают объект `Message<T>`, переправляют его в адрес внешней системы и ждут ответ от этой системы. Они поддерживают взаимодействие запросов и ответов.

Адаптер на входе представляет собой компонент, принимающий сообщение из внешнего мира и превращающий их в Spring `Message<T>`. *Адаптер на выходе* выполняет то же самое, но в обратном порядке: он получает Spring `Message<T>` и поставляет его в виде типа сообщения, ожидаемого нижестоящей системой. Spring Integration снабжается набором различных адаптеров для технологий и протоколов, включая MQTT, Twitter, email, (S)FTP(S), XMPP, TCP/UDP и т. д.

Существует два вида адаптеров на входе: либо *опрашивающий адаптер*, либо *адаптер, управляемый событиями*. Первый на входе настраивается на автоматическое извлечение данных через определенный интервал или со скоростью, определяемой источником сообщений, находящимся выше по потоку.

Шлюз — это компонент, обрабатывающий как запросы, так и ответы. Например, шлюз на входе будет принимать сообщение из внешнего мира, доставлять его к Spring Integration, а затем отправлять ответное сообщение обратно во внешний мир, на что может быть похож обычный HTTP-поток. *Шлюз на выходе* станет принимать сообщение от Spring Integration и доставлять его во внешний мир, затем принимать ответ и доставлять его обратно во фреймворк. Эти действия можно

наблюдать при использовании поставщика RabbitMQ, если указать ему пункт назначения ответа.

Фильтр представляет собой компонент, применяющий некие условия для определения характера обработки поступающего сообщения. Фильтр следует воспринимать как проверку `if (...)` в Java.

Маршрутизатор принимает входящее сообщение и применяет проверку (или, если нужно, несколько) для определения, куда именно следует отправить это сообщение ниже по потоку. Воспринимайте маршрутизатор как инструкцию `switch` для сообщений.

Преобразователь принимает сообщение и производит над ним некие действия, дополняя или изменяя его, а затем отправляет сообщение дальше.

Разделитель (сплиттер) принимает сообщение, а затем, на основе неких свойств сообщения, делит его на несколько меньших по объему сообщений, которые после перенаправляются вниз по потоку. Например, может быть принято входящее сообщение для заказа, а затем отправлено сообщение по каждой товарной позиции в этом заказе какому-нибудь из потоков выполнения.

Агрегатор принимает несколько сообщений, сопоставляемых по какому-либо уникальному свойству, и составляет сообщение, отправляемое вниз по потоку.

Интеграционный поток осведомлен о системе, но компоненты, привлеченные к интеграции, не нуждаются в этом. Данное обстоятельство упрощает составление сложных решений из небольших, в иных случаях обособленных сервисов. Лишь создание потока Spring Integration полезно само по себе. Оно заставляет проводить логическую декомпозицию сервисов, при которой у них должна быть возможность вести обмен данными в понятиях сообщений, содержащих информационное наполнение. Схемой информационного наполнения является контракт. Это свойство крайне полезно в распределенных системах.

От простых компонентов к сложным системам

Spring Integration поддерживает *архитектуры, управляемые событиями*, поскольку может поспособствовать обнаружению событий во внешнем мире и реагированию на них. Например, он пригоден для опроса файловой системы каждые десять секунд и публикации `Message<T>` при появлении нового файла. Данный фреймворк можно использовать в качестве слушателя сообщений, доставленных в тему Apache Kafka. Адаптер обрабатывает ответ на внешнее событие, освобождая вас от излишних забот об источнике происхождения сообщения и позволяя сконцентрироваться на обработке сообщения после его поступления. Это интеграционный эквивалент внедрения зависимостей.

Внедрение зависимостей освобождает код компонента от забот об инициализации и получении ресурсов и позволяет сосредоточиться на написании кода с этими

зависимостями. Откуда взялось поле `javax.sql.DataSource`? Какая разница! Подключением к источнику данных занимается Spring. Возможно, данная среда получила его от имитатора в программе тестирования, от JNDI в классическом сервере приложения или от сконфигурированного Spring Boot bean-компонента. Код компонента лишен этих подробностей. Внедрение зависимостей можно объяснить «принципом Голливуда»: «Не звоните мне, я сам позвоню».

Не приходится инициализировать объект зависимости или искать ресурс, объекты зависимостей предоставляются нуждающемуся в них объекту. Такой же принцип применим и к Spring Integration: код написан таким образом, что он не знает, откуда идут сообщения. Это существенно упрощает процесс разработки.

Приступим к несложному примеру, который реагирует на появление в каталоге новых файлов, регистрирует отслеживаемый файл, после чего на основе простой проверки динамически направляет информационное наполнение в один из двух возможных потоков, используя маршрутизатор.

Воспользуемся Spring Integration Java DSL, который хорошо справляется с лямбда-выражениями в Java 8. Каждый интеграционный поток `IntegrationFlow` выстраивает цепочку компонентов. Это действие можно провести явным образом, предоставив, как продемонстрировано в примере 10.2, связующие ссылки `MessageChannel`.

Пример 10.2. Выстраивание цепочки компонентов

```
package eda;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.channel.MessageChannels;
import org.springframework.integration.dsl.file.Files;
import org.springframework.messaging.MessageChannel;

import java.io.File;

@Configuration
public class IntegrationConfiguration {

 private final Log log = LogFactory.getLog(getClass());

 @Bean
 IntegrationFlow etlFlow(
  @Value("${input-directory:${HOME}/Desktop/in}") File dir) {

  return IntegrationFlows
```

```
  ❶
   .from(Files.inboundAdapter(dir).autoCreateDirectory(true),
    consumer -> consumer.poller(spec -> spec.fixedRate(1000)))
   ❷
   .handle(File.class, (file, headers) -> {
    log.info("we noticed a new file, " + file);
    return file;
})
❸
.routeToRecipients(
 spec -> spec.recipient(csv(), msg -> hasExt(msg.getPayload(), ".csv"))
  .recipient(txt(), msg -> hasExt(msg.getPayload(), ".txt"))).get();
}

private boolean hasExt(Object f, String ext) {
 File file = File.class.cast(f);
 return file.getName().toLowerCase().endsWith(ext.toLowerCase());
}

❹
@Bean
MessageChannel txt() {
 return MessageChannels.direct().get();
}

❺
@Bean
MessageChannel csv() {
 return MessageChannels.direct().get();
}

❻
@Bean
IntegrationFlow txtFlow() {
 return IntegrationFlows.from(txt()).handle(File.class, (f, h) -> {
  log.info("file is .txt!");
  return null;
 }).get();
}

❼
@Bean
IntegrationFlow csvFlow() {
 return IntegrationFlows.from(csv()).handle(File.class, (f, h) -> {
  log.info("file is .csv!");
  return null;
 }).get();
}
}
```

❶ Конфигурирование в Spring Integration адаптера на входе по имени `File`. Кроме того, нужно настроить способ использования адаптером входящих сообщений и указать интервал опроса каталога в миллисекундах.

❷ Этот метод объявляет о получении файла, а затем перенаправляет дальше информационное наполнение.

❸ Направление запроса на основе расширения файла в один из двух возможных интеграционных потоков через известные экземпляры `MessageChannel`.

❹ Канал, по которому будут направлены все файлы с расширением `.txt`.

❺ Канал, по которому будут направлены все файлы с расширением `.csv`.

❻ `IntegrationFlow` для обработки файлов с расширением `.txt`.

❼ `IntegrationFlow` для обработки файлов с расширением `.csv`.

Канал — это логическая развязка; если есть указатель на него, то что именно находится на любом конце канала, не имеет значения. Сегодня потребитель, взявшийся из канала, может быть, как и в данном примере, простым зарегистрированным `MessageHandler<T>`, но завтра это может быть компонент, записывающий сообщение в Apache Kafka. Можно также начать поток с момента его прибытия в канал. Как именно он туда прибывает, не имеет значения. Можно принимать запросы от REST API, или адаптировать сообщения, поступающие из Apache Kafka, или отслеживать состояние каталога. Это не имеет значения, поскольку входящие сообщения тем или иным образом адаптируются в `java.io.File`, отправляемый в нужный канал.

Предположим, что имеется пакетный процесс, работающий с файлами. По традиции такие задачи будут запускаться в конкретное время, вероятно, в планировщике, подобном cron. Между запусками образуется время простоя, и он задерживает получение искомых результатов. Для запуска пакетного задания при каждом появлении нового объекта `java.io.File` можно воспользоваться Spring Integration. Этот фреймворк предоставляет файловый адаптер, работающий на входе. Мы станем отслеживать сообщения, поступающие от входного файлового адаптера, выполняя их преобразование в сообщение, чьей информационной нагрузкой будет запрос на запуск задания `JobLaunchRequest`. В запросе указывается, какое именно задание `Job` запустить, а также параметры `JobParameters` для него. И наконец, запрос `JobLaunchRequest` перенаправляется далее к шлюзу запуска задания `JobLaunchingGateway`, который затем выдает на выходе объект выполнения задания `JobExecution`, подвергаемый проверке с целью принять решение о маршрутизации выполнения. При успешном выполнении задания входящий файл будет направлен в каталог выполненных заданий, в противном случае — в каталог ошибок.

У нас будет один основной поток, направляющий ход выполнения кода по одному из ответвлений выполнения, представленному двумя каналами: `invalid` и `completed` (пример 10.3).

Пример 10.3. Два экземпляра MessageChannel

```
package edabatch;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.integration.dsl.channel.MessageChannels;
import org.springframework.messaging.MessageChannel;

@Configuration
class BatchChannels {

 @Bean
 MessageChannel invalid() {
  return MessageChannels.direct().get();
 }

 @Bean
 MessageChannel completed() {
  return MessageChannels.direct().get();
 }

}
```

Основной поток, названный `etlFlow`, отслеживает состояние каталога (`directory`) через постоянные интервалы времени и превращает каждое событие в запрос на запуск задания `JobLaunchRequest` (пример 10.4).

Пример 10.4. Два экземпляра EtlFlowConfiguration

```
package edabatch;

import org.springframework.batch.core.*;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.batch.integration.launch.JobLaunchRequest;
import org.springframework.batch.integration.launch.JobLaunchingGateway;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.file.Files;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.messaging.Message;

import java.io.File;

import static org.springframework.integration.file.FileHeaders.ORIGINAL_FILE;

@Configuration
class EtlFlowConfiguration {

 ❶
 @Bean
 IntegrationFlow etlFlow(
  @Value("${input-directory:${HOME}/Desktop/in}") File directory,
```

```
  BatchChannels c, JobLauncher launcher, Job job) {

  return IntegrationFlows
   .from(Files.inboundAdapter(directory).autoCreateDirectory(true),
    cs -> cs.poller(p -> p.fixedRate(1000)))
   .handle(
    File.class,
    (file, headers) -> {

     String absolutePath = file.getAbsolutePath();
           ❷
     JobParameters params = new JobParametersBuilder().addString("file",
      absolutePath).toJobParameters();

     return MessageBuilder.withPayload(new JobLaunchRequest(job, params))
      .setHeader(ORIGINAL_FILE, absolutePath)
      .copyHeadersIfAbsent(headers).build();
   })
         ❸
  .handle(new JobLaunchingGateway(launcher))
         ❹
  .routeToRecipients(
   spec -> spec.recipient(c.invalid(), this::notFinished).recipient(
    c.completed(), this::finished)).get();
 }

 private boolean finished(Message<?> msg) {
  Object payload = msg.getPayload();
  return JobExecution.class.cast(payload).getExitStatus()
   .equals(ExitStatus.COMPLETED);
 }

 private boolean notFinished(Message<?> msg) {
  return !this.finished(msg);
 }
}
```

❶ Эта конфигурация `EtlFlowConfiguration` начинается так же, как и в предыдущем примере.

❷ Установка `JobParameters` для задания Spring Batch с помощью `JobParametersBuilder`.

❸ Отправка задания и связанных с ним параметров шлюзу `JobLaunchingGateway`.

❹ Проверка с целью убедиться в успешном выходе из задания путем изучения значения объекта `JobExecution`.

Последний компонент в потоке Spring Integration — маршрутизатор, проверяющий значение объекта `ExitStatus` из `Job` и определяющий, в какой именно каталог должен быть перемещен входящий файл. При успешном завершении обработки файл

будет направлен в канал `completed`. Если произошла ошибка или же по каким-либо причинам задание было прервано преждевременно, то он будет направлен в канал `invalid`.

Конфигурация `FinishedFileFlowConfiguration` отслеживает входящие сообщения в канале `completed`, перемещая поступающее в них информационное наполнение (`java.io.File`) в каталог `completed`, а затем запрашивает таблицу, в которую были записаны данные (пример 10.5).

Пример 10.5. Экземпляры FinishedFileFlowConfiguration

```
package edabatch;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.batch.core.JobExecution;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.file.FileHeaders;
import org.springframework.jdbc.core.JdbcTemplate;
import java.io.File;

import java.util.List;
import static edabatch.Utils.mv;

@Configuration
class FinishedFileFlowConfiguration {

 private Log log = LogFactory.getLog(getClass());

 @Bean
 IntegrationFlow finishedJobsFlow(BatchChannels channels,
  @Value("${completed-directory:${HOME}/Desktop/completed}") File finished,
  JdbcTemplate jdbcTemplate) {
  return IntegrationFlows
    .from(channels.completed())
    .handle(JobExecution.class,
      (je, headers) -> {
       String ogFileName = String.class.cast(headers
         .get(FileHeaders.ORIGINAL_FILE));
       File file = new File(ogFileName);
       mv(file, finished);
       List<Contact> contacts = jdbcTemplate.query(
         "select * from CONTACT",
         (rs, i) -> new Contact(
           rs.getBoolean("valid_email"),
           rs.getString("full_name"),
           rs.getString("email"),
```

```
            rs.getLong("id")));
        contacts.forEach(log::info);
        return null;
       }).get();
 }

}
```

Конфигурация `InvalidFileFlowConfiguration` отслеживает входящие сообщения в канале `invalid`, перемещая поступающее в них информационное наполнение (`java.io.File`) в каталог `errors`, а затем прерывает работу потока (пример 10.6).

Пример 10.6. The InvalidFileFlowConfiguration instances

```
package edabatch;

import org.springframework.batch.core.JobExecution;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.file.FileHeaders;

import java.io.File;

import static edabatch.Utils.mv;

@Configuration
class InvalidFileFlowConfiguration {

 @Bean
 IntegrationFlow invalidFileFlow(BatchChannels channels,
  @Value("${error-directory:${HOME}/Desktop/errors}") File errors) {
  return IntegrationFlows
    .from(channels.invalid())
    .handle(JobExecution.class,
      (je, headers) -> {
       String ogFileName = String.class.cast(headers
         .get(FileHeaders.ORIGINAL_FILE));
       File file = new File(ogFileName);
       mv(file, errors);
       return null;
      }).get();
 }

}
```

Определения `MessageChannel` служат для развязки различных потоков интеграции. Общие функции можно использовать повторно и составлять системы высшего порядка, применяя в качестве связи только канал. Интерфейсом является `MessageChannel`.

Поставщики сообщений, наведение мостов, шаблон конкурирующих потребителей и порождение событий

Последний пример был несложным. Это простой поток, работающий в понятиях сообщений (событий). К обработке стоит приступать при первой же возможности, не перегружая систему. Поскольку интеграция выстраивается в понятиях каналов и каждый компонент может с помощью канала потреблять или создавать сообщения, то во введении поставщика сообщений между файловым адаптером на входе и узлом, фактически запускающим задание Spring Batch, не будет ничего необычного. Это позволит провести горизонтальное масштабирование работы по всему кластеру, как, например, на базе Cloud Foundry.

Тем не менее вы, скорее всего, не станете интегрировать файловые системы в архитектуру, ориентированную на облачные вычисления. У вас, наверное, найдутся другие, не связанные с транзакциями источники событий (производители сообщений) и приемники (потребители сообщений), с которыми нужно интегрироваться.

Можно воспользоваться паттерном саги (saga pattern) и спроектировать для каждого привлекаемого к интеграции сервиса и для любых условий сбоя транзакцию компенсации. Но в случае использования поставщика сообщений подойдет более легкое решение. Концептуально поставщики сообщений довольно просты: как только сообщения доставляются поставщику, они сохраняются и отправляются подключенным потребителям. Если таковые отсутствуют, то поставщик сохранит сообщения и отправит их при подключении потребителя.

Как правило, поставщики сообщений предлагают два типа распространения (представьте это в виде почтовых ящиков): *публикация-подписка* (publish-subscribe) и *от точки к точке* (point-to-point).

Распространение типа «публикация-подписка»

При распространении данного типа одно сообщение отправляется всем подключенным потребителям. Это напоминает публичное объявление через мегафон в заполненном зале.

Рассылка сообщений по типу «публикация-подписка» поддерживает событийное сотрудничество, при котором несколько систем сохраняют собственное представление или перспективу состояния мира. При поступлении новых событий от различных компонентов системы с обновлением состояния объектов предметной области каждая система будет реагировать обновлением перспективного взгляда на состояние приложения.

Предположим, что имеется каталог товаров. При добавлении к сервису товаров `ProductService` новых записей он может опубликовать событие с описанием изменения. Сервис поиска `SearchEngine` может воспользоваться сообще-

ниями, после чего обновить индекс в локально связанном экземпляре сервиса `ElasticSearch`. Сервис запасов `InventoryService` может обновить данные, содержащиеся в локально связанном экземпляре сервиса `RDBMS`. Сервис рекомендаций `RecommendationService` может принять решение об обновлении данных в локально связанном экземпляре сервиса `Neo4j`. Этим системам больше не нужно о чем-либо *спрашивать* сервис товаров `ProductService`.

При записи каждого события предметной области в упорядоченном журнале появляется возможность делать запросы по времени, воссоздавая состояние системы на любой предыдущий момент времени. При сбое какого-нибудь сервиса его локальное состояния можно полностью воссоздать из журнала. Такой прием называется *порождением событий* (Event Sourcing) и является весьма эффективным средством при создании распределенных систем.

Распространение от точки к точке

При таком распространении одно сообщение доставляется одному потребителю, даже если таковых несколько. Это как поделиться секретом с одним человеком из группы. Представим, что подключены несколько потребителей и все они работают с максимальной для себя скоростью на отбор сообщений из канала распространения от точки к точке. Тогда происходит нечто, напоминающее балансировку нагрузки: работа делится на количество подключенных потребителей. Такой подход, названный *шаблоном конкурирующих потребителей* (competing consumers), упрощает деятельность по балансировке нагрузки между несколькими потребителями. Это идеальный способ использования эластичности облачной вычислительной среды наподобие Cloud Foundry, где горизонтальная емкость является эластичной и (практически) бесконечной.

У поставщиков сообщений также имеется собственное понятие транзакции, локальное по отношению к ресурсу. Поставщик может доставить сообщение, а затем при необходимости отозвать его, выполнив фактически откат сообщения. Потребитель может принять доставку сообщения, попытаться его обработать, после чего подтвердить доставку или же, если что-то пойдет не так, возвратить сообщение поставщику, фактически выполнив откат доставки. *В конечном итоге* обе стороны договорятся о состоянии. Описанный процесс отличается от распределенной транзакции тем, что поставщик сообщений вводит переменную времени или развязку по времени. Тем самым упрощается интеграция сервисов. Это свойство облегчает обоснование состояния в распределенной системе. Можно гарантировать, что два в иных случаях не применяющих транзакций ресурса со временем договорятся насчет состояния. В результате поставщик сообщений *наводит мосты* (bridges) между двумя ресурсами, в иных условиях не применяющими транзакций.

Поставщик сообщений усложняет архитектуру за счет добавления к системе еще одной движущейся части. У поставщиков сообщений имеются хорошо знакомые рецепты по аварийному восстановлению, резервному копированию и масштабированию. В организации должны знать, как выполнить данные действия, после

чего она сможет повторно задействовать их во всех сервисах. В противном случае для каждого сервиса придется изобретать все заново, или же, что менее желательно, обходиться без этих качеств. При использовании таких платформ, как Cloud Foundry, которые уже справляются с ролью поставщика сообщений, применение такого поставщика становится весьма простым решением.

Рассуждая логически, в использовании поставщиков сообщений есть особый резон, поскольку они обеспечивают способ связи между различными сервисами. Фреймворк Spring Integration обеспечивает в этом смысле вполне достаточную поддержку, предоставляя адаптеры, создающие и потребляющие сообщения от различных поставщиков.

Spring Cloud Stream

Spring Integration создает с помощью поставщиков сообщений прочную основу для решения задач обмена данными от сервиса к сервису, но для микросервисов этот фреймворк может показаться слишком громоздким. Здесь нужна поддержка рассылки сообщений не сложнее того, что представляется в виде операций взаимодействий на основе использования REST в Spring. Мы не собираемся соединять сервисы с помощью «Твиттера» или электронной почты. По всей вероятности, будет применен RabbitMQ, или Apache Kafka, или подобные им поставщики сообщений с известными режимами взаимодействия. Конечно же, можно сконфигурировать явным образом входные и выходные адаптеры RabbitMQ или Apache Kafka. Но вместо этого поднимемся немного выше, чтобы упростить задачу и устранить когнитивный диссонанс конфигурирования входных и выходных адаптеров при каждой потребности в работе с другим сервисом.

Spring Integration хорошо справляется с развязкой компонентов применительно к объектам `MessageChannel`. Эти объекты — неплохой уровень обхода связности. С точки зрения логики приложения канал представляет собой логический проводник к какому-то нисходящему сервису, направление к которому будет дано через поставщика сообщений.

В данном разделе мы рассмотрим среду Spring Cloud Stream — надстройку над Spring Integration, в основе модели взаимодействия которой лежит канал. Здесь подразумевается использование соглашений и имеется поддержка легкой экстернализации конфигурации. Взамен приобретается четкость и лаконичность общих случаев соединения сервисов с поставщиками сообщений.

Рассмотрим простой пример. Сконструируем производитель и потребитель. Первый предоставит конечную точку REST API, при вызове которой будет публиковаться сообщение в два канала: один для *широковещательной* (или выполняемой в стиле «публикация-подписка») рассылки сообщений, а другой — для рассылки сообщений *от точки к точке*. Затем возьмемся за потребителя для приема этих входящих сообщений.

Spring Cloud Stream упрощает определение каналов, которые затем подключаются к технологиям рассылки сообщений. Для подключения к поставщику можно по

соглашению применить реализации *привязчика*. В данном примере будет использоваться RabbitMQ, популярный поставщик сообщений, понимающий спецификацию расширенного протокола организации очереди сообщений AMQP. Привязчиком для поддержки в Spring Cloud Stream поставщика RabbitMQ является `org.springframework.cloud : spring-cloud-starter-stream-rabbit`.

Существуют клиенты и привязки на десятках языков, что позволяет RabbitMQ (и спецификации AMQP в целом) прекрасно вписываться в осуществление интеграции на разных языках и платформах. Spring Cloud Stream основывается на фреймворке Spring Integration (предоставляющем необходимые входные и выходные адаптеры), а тот, в свою очередь, основывается на среде Spring AMQP (предоставляющей низкоуровневые компоненты `AmqpOperations`, `RabbitTemplate`, RabbitMQ-реализацию компонента `ConnectionFactory` и т. д.). Среда Spring Boot выполняет автоматическое конфигурирование `ConnectionFactory` на основе исходных установок или свойств. На локальной машине с чистым экземпляром RabbitMQ это приложение будет работать в исходном состоянии.

Производитель потока

Центральный элемент в Spring Cloud Stream — *привязка*. В ней определяются логические ссылки на другие сервисы с помощью экземпляров `MessageChannel`, которые мы оставим на попечение самой среде, чтобы она выполнила подключение. С точки зрения нашей бизнес-логики это вышестоящие или нижестоящие сервисы, работа которых основана на рассылке сообщений. На данный момент беспокоиться о том, как происходит подключение, не нужно. Определим два канала: один для передачи приветствия всем потребителям и второй — для однократной отправки приветствия от точки к точке любому первому попавшемуся потребителю (пример 10.7).

Пример 10.7. Простой производитель приветствий, ориентированный на использование объектов MessageChannel

```
package stream.producer;

import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;

public interface ProducerChannels {

 ❶
 String DIRECT = "directGreetings";

 String BROADCAST = "broadcastGreetings";

 @Output(DIRECT)
 ❷
 MessageChannel directGreetings();

 @Output(BROADCAST)
 MessageChannel broadcastGreetings();
}
```

❶ В исходном состоянии имя потока (который будет работать с другими частями системы) основано на самом методе `MessageChannel`. Будет полезно предоставить интерфейсу константу типа `String`, чтобы можно было ссылаться без каких-либо магических строк.

❷ Среда Spring Cloud Stream предоставляет две аннотации: `@Output` и `@Input`. Первая сообщает среде, что сообщения, помещаемые в канал, будут отправляться далее (обычно и в конечном счете через выходной адаптер канала в Spring Integration).

Среде Spring Cloud Stream нужно подать идею о том, что делать с данными, отправленными в эти каналы. Здесь помогут некоторые правильно расположенные свойства в среде окружения.

В примере 10.8 показано, как выглядят свойства приложения нашего поставщика `application.properties`.

Пример 10.8. Простой производитель приветствий, ориентированный на использование объектов MessageChannel

```
spring.cloud.stream.bindings.broadcastGreetings.destination=greetings-pub-sub ❶
spring.cloud.stream.bindings.directGreetings.destination=greetings-p2p

spring.rabbitmq.addresses=localhost ❷
```

❶ В этих двух строках фрагменты, следующие сразу же за `spring.cloud.stream.bindings.` и находящиеся непосредственно перед `.destination`, должны соответствовать имени Java `MessageChannel`. Это локальный взгляд приложения на вызываемый им сервис. Та небольшая часть, которая находится после знака `=`, является согласованной точкой рандеву для производителя и потребителя. Здесь обе стороны нуждаются в указании конкретного имени. Это имя пункта назначения в любом сконфигурированном нами поставщике.

❷ Для создания RabbitMQ-фабрики `ConnectionFactory`, от которой будет зависеть Spring Cloud Stream, используется автоконфигурация Spring Boot.

В примере 10.9 показан простой производитель, поддерживающий REST API, который затем публикует сообщения, отслеживаемые потребителем.

Пример 10.9. Простой производитель приветствий, ориентированный на использование объектов MessageChannel

```
package stream.producer.channels;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.http.ResponseEntity;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.web.bind.annotation.PathVariable;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import stream.producer.ProducerChannels;

@SpringBootApplication
@EnableBinding(ProducerChannels.class)
❶
public class StreamProducer {

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

@RestController
class GreetingProducer {

 private final MessageChannel broadcast, direct;

 ❷
 @Autowired
 GreetingProducer(ProducerChannels channels) {
  this.broadcast = channels.broadcastGreetings();
  this.direct = channels.directGreetings();
 }

 @RequestMapping("/hi/{name}")
 ResponseEntity<String> hi(@PathVariable String name) {
  String message = "Hello, " + name + "!";

  ❸
  this.direct.send(MessageBuilder.withPayload("Direct: " + message).build());

  this.broadcast.send(MessageBuilder.withPayload("Broadcast: " + message)
   .build());
  return ResponseEntity.ok(message);
 }
}
```

❶ Аннотация `@EnableBinding` приводит к активации Spring Cloud Stream.

❷ Мы внедряем специально обработанный объект `ProducerChannels`, а затем разыменовываем требуемые каналы в конструкторе, чтобы можно было отправить сообщения, когда кто-то делает HTTP-запрос по адресу `/hi/{имя}`.

❸ Это обычный канал среды Spring, поэтому использовать `MessageBuilder` API для создания `Message<T>` довольно просто.

Разумеется, стиль имеет значение, так что, несмотря на снижение затрат на работу с нижестоящими сервисами за счет ее сведения к использованию интерфейса (то есть к написанию нескольких строк кода с объявлением свойств и нескольких строк кода на операции с каналом, ориентированным на рассылку сообщений),

можно было бы добиться более качественного результата, воспользовавшись имеющейся во фреймворке Spring Integration поддержкой шлюза сообщений. Такой шлюз, являющийся паттерном проектирования, предназначен для того, чтобы скрыть клиент от логики рассылки сообщений за сервисом. С точки зрения клиента шлюз может показаться обычным объектом. И это обстоятельство, вероятно, будет весьма удобным. Сегодня можно определить интерфейс и синхронный сервис, завтра извлечь его в качестве основной реализации шлюза рассылки сообщений, и никто этого не заметит. Вернемся к производителю и вместо отправки сообщений непосредственно с помощью Spring Integration отправим их с применением шлюза рассылки сообщений (пример 10.10).

Пример 10.10. Реализация производителя шлюзов рассылки сообщений

```
package stream.producer.gateway;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.http.ResponseEntity;
import org.springframework.integration.annotation.Gateway;
import org.springframework.integration.annotation.IntegrationComponentScan;
import org.springframework.integration.annotation.MessagingGateway;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import stream.producer.ProducerChannels;


@SpringBootApplication
@EnableBinding(ProducerChannels.class)
❶
@IntegrationComponentScan
❷
public class StreamProducer {

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

❸
@MessagingGateway
interface GreetingGateway {

 @Gateway(requestChannel = ProducerChannels.BROADCAST)
 void broadcastGreet(String msg);

 @Gateway(requestChannel = ProducerChannels.DIRECT)
```

```
 void directGreet(String msg);
}

@RestController
class GreetingProducer {

 private final GreetingGateway gateway;

 ❹
 @Autowired
 GreetingProducer(GreetingGateway gateway) {
  this.gateway = gateway;
 }

 @RequestMapping(method = RequestMethod.GET, value = "/hi/{name}")
 ResponseEntity<?> hi(@PathVariable String name) {
  String message = "Hello, " + name + "!";
  this.gateway.directGreet("Direct: " + message);
  this.gateway.broadcastGreet("Broadcast: " + message);
  return ResponseEntity.ok(message);
 }
}
```

❶ Аннотация `@EnableBinding`, как и прежде, приводит к активации Spring Cloud Stream.

❷ Многие Spring-среды регистрируют стереотипные аннотации для настраиваемых компонентов, но в Spring Integration можно также превратить определения интерфейсов в bean-компоненты. Поэтому для активации компонента Spring Integration, проводящего сканирование с целью найти декларативный шлюз рассылки сообщений, основанный на использовании интерфейса, нужна настроечная аннотация.

❸ Аннотация `@MessagingGateway` — одна из многих конечных точек рассылки сообщений, поддерживаемых фреймворком Spring Integration (в качестве альтернативы используемому до сих пор Java DSL). Каждый метод в шлюзе имеет аннотацию `@Gateway`, где указывается, к какому каналу сообщений должны пойти аргументы метода. В данном случае все будет так, будто бы мы отправили сообщение на канал и вызвали `.send(Message<String>)` с аргументом в качестве информационного наполнения.

❹ Что касается остальной части нашей бизнес-логики, то `GreetingGateway` является обычным bean-компонентом.

Потребитель потока

С другой стороны, нужно принять доставку сообщений и провести их регистрацию. Каналы будут созданы с использованием интерфейса. Стоит повторить: имена этих каналов не обязаны совпадать с именами на стороне производителя, они должны совпадать лишь с именами получателей у поставщиков (пример 10.11).

Пример 10.11. Каналы для входящих приветствий

```
package stream.consumer;

import org.springframework.cloud.stream.annotation.Input;
import org.springframework.messaging.SubscribableChannel;

public interface ConsumerChannels {

 String DIRECTED = "directed";

 String BROADCASTS = "broadcasts";

 ❶
 @Input(DIRECTED)
 SubscribableChannel directed();

 @Input(BROADCASTS)
 SubscribableChannel broadcasts();

}
```

❶ Единственное, что здесь стоит отметить: данные каналы имеют аннотацию `@Input` (это вполне естественно!) и возвращают подтип `MessageChannel`, поддерживающий *подписку* на входящие сообщения `SubscribableChannel`.

Следует помнить: все привязки в Spring Cloud Stream изначально являются публикациями-подписками. Мы можем добиться эффекта исключительности и прямого присоединения к группе потребителей. Например: если, скажем, в группе с названием `foo` имеется десять экземпляров потребителей, то только один из них увидит доставленное ему сообщение. В распределенной системе нельзя быть уверенными, что сервис всегда будет запущен, поэтому мы воспользуемся *продолжительными подписками* (durable subscriptions), чтобы гарантировать повторную доставку сообщений, как только потребитель подключится к поставщику (пример 10.12).

Пример 10.12. application.properties для нашего потребителя

```
spring.cloud.stream.bindings.broadcasts.destination=greetings-pub-sub  ❶

spring.cloud.stream.bindings.directed.destination=greetings-p2p  ❷
spring.cloud.stream.bindings.directed.group=greetings-p2p-group
spring.cloud.stream.bindings.directed.durableSubscription=true
server.port=0
spring.rabbitmq.addresses=localhost
```

❶ Учитывая, что мы сейчас рассмотрели производитель, это должно выглядеть вполне знакомо.

❷ Здесь, как и прежде, выполняется конфигурирование получателя, но, кроме этого, дается непосредственный потребитель в исключительной группе таковых. Входящее сообщение увидит только один узел из всех активных

потребителей в группе `greetings-p2p-group`. Указав, что в привязке имеется `durableSubscription`, мы гарантируем: поставщик сохранит и проведет повторную доставку недоставленных сообщений, как только потребитель выполнит переподключение.

И наконец, в примере 10.13 показан потребитель Spring Integration, реализованный на основе Java DSL. Вам все это должно быть весьма знакомо.

Пример 10.13. Потребитель, функционирующий за счет применения Spring Integration Java DSL

```
package stream.consumer.integration;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.context.annotation.Bean;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.messaging.SubscribableChannel;
import stream.consumer.ConsumerChannels;

@SpringBootApplication
@EnableBinding(ConsumerChannels.class)
(1)
public class StreamConsumer {

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

 (2)
 private IntegrationFlow incomingMessageFlow(SubscribableChannel incoming,
  String prefix) {

  Log log = LogFactory.getLog(getClass());

  return IntegrationFlows
   .from(incoming)
   .transform(String.class, String::toUpperCase)
   .handle(
    String.class,
    (greeting, headers) -> {
     log.info("greeting received in IntegrationFlow (" + prefix + "): "
      + greeting);
     return null;
    }).get();
 }

 @Bean
```

```
 IntegrationFlow direct(ConsumerChannels channels) {
  return incomingMessageFlow(channels.directed(), "directed");
 }

 @Bean
 IntegrationFlow broadcast(ConsumerChannels channels) {
  return incomingMessageFlow(channels.broadcasts(), "broadcast");
 }
}
```

1. Как и прежде, мы видим аннотацию `@EnableBinding`, которая активирует каналы потребителей.
2. В классе `@Configuration` определяются два потока `IntegrationFlow`, которые делают в основном одно и то же: принимают входящее сообщение, преобразуют его, заменяя строчные буквы прописными, а затем регистрируют его. Один поток отслеживает транслируемые приветствия, а другой предназначен для непосредственного приветствия, направляемого от точки к точке. Обработка прекращается путем возвращения значения `null` в финальном компоненте цепочки.

Попробуйте это в работе: запустите один экземпляр одного из узлов производителя (все равно, какой именно) и три экземпляра потребителя. Зайдите на страницу http://localhost:8080/hi/World и зафиксируйте в журналах потребителей, что у всех трех имеется сообщение, отправленное по каналу трансляции, а у одного (хотя здесь не говорится, у какого именно, поэтому проверьте все консоли) будет сообщение, отправленное по прямому каналу. Повысьте ставки, отключив все узлы потребителей, после чего зайдите на страницу http://localhost:8080/hi/Again. Потребители бездействуют, но мы указали, что подключение от точки к точке носит продолжительный характер; вследствие этого, как только один из потребителей будет перезапущен, вы увидите поступление сообщения по прямому каналу и его регистрацию в консоли.

Резюме

Рассылка сообщений предоставляет канал связи для сложных распределенных взаимодействий. В следующих главах мы увидим, что в отношении основы для рассылки сообщений можно подключать решения пакетной обработки, механизмы последовательно выполняемых действий и сервисы.

11 Пакетные процессы и задачи

Облачная среда предоставляет беспрецедентные возможности масштабирования. Практически ничего не стоит развернуть новые экземпляры приложений для удовлетворения спроса и, как только ажиотаж спадет, свернуть ненужное. Это значит, что, пока текущая работа поддается распараллеливанию, ее эффективность повышается за счет масштабирования. Одни задачи легко *поддаются распараллеливанию*, не требуя согласования между узлами, в отличие от других. Оба типа рабочей нагрузки идеально подходят для среды облачных вычислений, а вот другие типы носят изначально последовательный характер. Для работы, распараллелить которую нелегко, среда облачных вычислений идеально подходит с точки зрения горизонтального масштабирования вычислений по нескольким узлам. В данной главе мы рассмотрим различные способы, как старые, так и новые, для приема и обработки данных с помощью микросервисов.

Пакетные рабочие нагрузки

Пакетная обработка имеет давнюю историю. Под ней подразумевается идея одновременной программной обработки *пакетов* входных данных. Традиционно пакетная обработка — более эффективный способ применения вычислительных ресурсов. Он амортизирует стоимость линейки машин путем предоставления приоритета периодам интерактивной работы, когда машины задействуют операторы, над периодами неинтерактивной работы в вечерние часы, когда машины простаивали бы. Сегодня, в эпоху облачных вычислений с практически бесконечной и эфемерной вычислительной емкостью, эффективное использование машин не является особо веской причиной применения пакетной обработки.

Пакетная обработка также привлекательна, когда работа ведется с большими наборами данных. Последовательные данные — в частности, SQL-данные, `.csv`-файлы и т. д. — допускают такую обработку. Ценные ресурсы, подобные файлам,

табличным курсорам SQL и транзакциям, могут сохраняться во фрагменте данных, позволяя ускорять обработку.

В пакетной обработке поддерживается логическое понятие *окна* — верхней и нижней границ, с помощью которых один набор данных отделяется от другого. Возможно, окно будет иметь отношение к какому-то интервалу времени: все записи за последние 60 минут или все регистрационные записи за последние 24 часа. Возможно, в окно будет вкладываться логический смысл: первые 1000 записей или все записи с конкретным свойством.

Если обрабатываемый набор данных слишком велик, чтобы поместиться в памяти, то его можно обработать небольшими блоками. Блок — это весьма эффективное (хотя и ресурсоориентированное) разделение пакета данных. Предположим, нужно обратиться к каждой записи в базе данных продаж товаров, превышающей 20 миллионов строк. Без разбиения на страницы выполнение инструкции `SELECT * FROM PRODUCT_SALES` способно привести к загрузке в память всего набора данных, что может быстро вызвать перегрузку системы. Намного эффективнее будет разбить большой набор данных на страницы, одновременно загружая лишь 1000 (или 10 000!) записей. Последовательная или параллельная обработка блоков с перемещением далее, пока не будут просмотрены все записи объемного запроса, позволяет обойтись без одномоментной загрузки в память всего объема данных.

Если ваша система терпима к устаревшим данным, то пакет обеспечивает эффективность обработки. Под эту категорию подпадают многие системы: например, еженедельный отчет о продажах не потребует вычисления продаж за последнюю неделю, пока она не закончится.

Spring Batch

Spring Batch — это среда, созданная для поддержки обработки больших объемов записей. В нее включены регистрационно-трассировочные инструменты, управление транзакциями, статистика обработки заданий, перезапуск и пропуск заданий, а также управление ресурсами. Данная среда стала промышленным стандартом пакетной обработки в JVM.

В основе Spring Batch лежит понятие задания, у которого, в свою очередь, может быть несколько этапов. Затем каждый из них предоставляет контекст для дополнительного компонента считывания элемента (`ItemReader`), обработки элемента (`ItemProcessor`) и записи элемента (`ItemWriter`) (рис. 11.1).

Пакетные задания выполняются в несколько этапов. Под этапом подразумевается проведение подготовительных действий, прежде чем данные будут отправлены к следующему этапу. Поток данных может направляться от одного этапа к другому с помощью логики маршрутизации: условий, согласований и учета базового цик-

ла. Этапы также могут определять общую бизнес-функциональность в *тасклете* (tasklet). В данном случае для управления *последовательностью* выполнения используется среда Spring Batch. Для более конкретизированной обработки этапы могут расширять реализации `ItemReader`, `ItemProcessor` и `ItemWriter`.

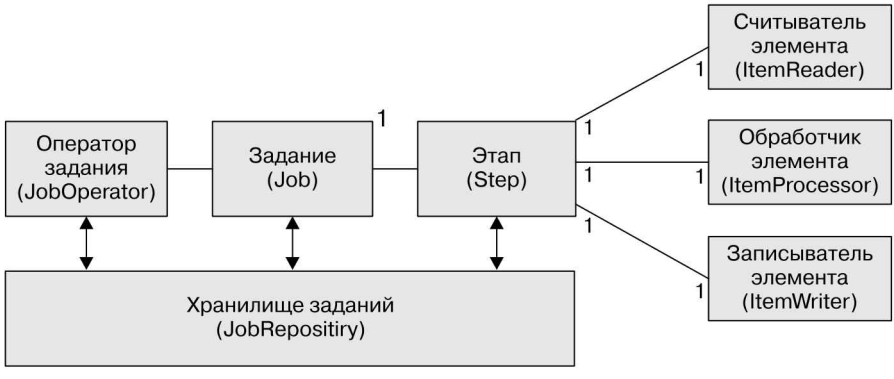


Рис. 11.1. Предметная область задания Spring Batch

Компонент `ItemReader` принимает ввод из внешнего мира (`.csv`- или XML-документы, базы данных SQL, каталоги и т. д.) и адаптирует его к чему-то, что поддается логической обработке в задании: к *элементу* (item). Последний может быть чем угодно: записью из БД или абзацем текстового файла, записью из файла `.csv` или строкой XML-документа. Среда Spring Batch предоставляет полезную, готовую к применению реализацию компонента `ItemReader`, позволяющую выполнять поэлементное считывание, но аккумулировать получаемые элементы в буфере, соответствующем определенному размеру *блока*.

Если определен компонент `ItemProcessor`, то ему от `ItemReader` будет передан каждый элемент. Компонент `ItemProcessor` предназначен для обработки и преобразования: данные поступают в него и выходят из него. `ItemProcessor` — идеальное место для любой логики размещения, или проверки, или бизнес-логики общего назначения, не связанной с вводом и выводом.

Компоненту `ItemWriter` указывается, когда он получит *накопленные* (а не отдельно взятые) элементы из `ItemProcessor` (если он присутствует в конфигурации) или из `ItemReader`. Компонент `ItemWriter` записывает накопленный блок элементов в ресурс наподобие базы данных или файла. `ItemWriter` ведет за одну запись максимально возможный объем данных, не превышающий указанный размер блока.

Именно это подразумевается, когда мы говорим о *пакетной* обработке. Для большинства типов средств ввода-вывода запись пакетов элементов позволяет существенно повысить производительность. Намного эффективнее вести запись пакетов строк в буфер и еще более эффективно сводить записи в пакет, помещаемый в базу данных.

Наше первое пакетное задание. Сначала рассмотрим высокоуровневое задание `Job` и поток сконфигурированных экземпляров этапов `Step` (пример 11.1).

Пример 11.1. Обычное задание Job с тремя экземплярами этапов Step

```
package processing;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration.annotation.
JobBuilderFactory;
import org.springframework.batch.core.configuration.annotation.
StepBuilderFactory;
import org.springframework.batch.core.launch.support.RunIdIncrementer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.HttpStatusCodeException;
import processing.email.InvalidEmailException;
import java.util.Map;

@Configuration
class BatchConfiguration {

 ❶
 @Bean
 Job etl(JobBuilderFactory jbf, StepBuilderFactory sbf,
  Step1Configuration step1,  ❷
  Step2Configuration step2, Step3Configuration step3) throws Exception {

  Step setup = sbf.get("clean-contact-table").tasklet(step1.tasklet(null)) ❸
   .build();

  Step s2 = sbf.get("file-db").<Person, Person>chunk(1000)
   ❹
   .faultTolerant()
   ❺
   .skip(InvalidEmailException.class).retry(HttpStatusCodeException.class)
   .retryLimit(2).reader(step2.fileReader(null))  ❻
   .processor(step2.emailValidatingProcessor(null)) ❼
   .writer(step2.jdbcWriter(null)) ❽
   .build();

  ❾
  Step s3 = sbf.get("db-file")
   .<Map<Integer, Integer>, Map<Integer, Integer>>chunk(100)
   .reader(step3.jdbcReader(null)).writer(step3.fileWriter(null)).build();

  return jbf.get("etl").incrementer(new RunIdIncrementer())  ❿
   .start(setup)  ⓫
   .next(s2).next(s3).build();  ⓬
  }
}
```

❶ Определение задания Spring Batch `Job` с помощью фабрик `JobBuilderFactory` и `StepBuilderFactory`.

❷ Для каждого этапа нужны и bean-компоненты, поэтому они были определены в классах конфигурации и внедрены, чтобы быть легко разыменованными.

❸ На первом этапе `Step` будет использоваться `Tasklet` — разновидность произвольной функции обратного вызова, в которой вам предоставляется полная свобода действий. Включая ее в этап, вы конкретизируете место ее выполнения относительно всего остального задания.

❹ Для сконфигурированного компонента `ItemWriter` нужно за одну операцию охватить десять записей.

❺ Требуется обработать возможные сбои, поэтому для задания следует настроить политики пропуска (при каких исключениях запись должна быть пропущена) и повторных попыток (при каких исключениях этап для данного элемента должен быть пройден повторно и сколько раз). Среда Spring Batch предоставляет множество механизмов управления, поэтому не нужно отменять все задание из-за одной неправильной записи.

❻ Здесь разыменовывается `FlatFileItemReader` из `Step1Configuration`. Компонент `ItemReader` считывает данные из `.csv`-файла и превращает их в простой объект языка Java (POJO) `Person`. Объявление `@Bean` предусматривает ожидание параметров, но их обеспечит среда Spring. Метод вполне можно вызвать вообще без аргументов, поскольку bean-компонент уже создан (с параметрами, предоставленными в контейнере), а возвращаемое значение метода будет кэшированным предварительно созданным экземпляром.

❼ Компонент `ItemProcessor` — идеальное место для вставки вызова веб-сервиса, проверяющего допустимость электронного адреса в записи `Person`.

❽ Разыменование `ItemWriter` для записи в источник данных JDBC `DataSource`.

❾ Здесь делается запрос на получение только что сохраненных данных из первого этапа, вычисляется частота записей с определенным возрастом, а затем эти вычисления записываются в выходной `.csv`-файл.

❿ Пакетное задание среды Spring Batch Job параметризировано. Параметры помогают *идентифицировать* задание. Эта идентификация сохраняется для вашей БД в таблицах метаданных. В нашем примере эти таблицы сохраняются в MySQL. Если попытаться запустить то же самое задание `Job` с теми же самыми параметрами дважды, то в запуске будет отказано, поскольку задание уже *встречалось* ранее, о чем имеется соответствующая запись в таблицах метаданных. Здесь выполняется конфигурация компонента `RunIdIncrementer`, получающего новый ключ путем увеличения на единицу значения существующего ключа (который называется параметром `run.id`), если таковой уже имеется.

⓫ Теперь поток этапов в задании объединяется, начиная с этапа настройки и продолжая этапом `s1` и этапом `s3`.

⓬ И наконец, сборка задания `Job`.

Предположим, после запуска задания что-то пошло не так: считывание дало сбой и задание было отменено. Среда Spring Batch пройдет через сконфигурированные политики повторения и пропуска и предпримет попытку продолжить выполнение. Если задание ждет неудачное завершение, все пройденное будет записано в таблицах метаданных, а затем оператор может вмешаться и, вероятно, перезапустить задание (вместо запуска другого, дублирующего экземпляра). Задание должно продолжить выполнение с места остановки. Такая возможность имеется благодаря тому, что некоторые реализации `ItemReader` способны считывать ресурсы, сохраняющие состояние, и записывать пройденные этапы их обработки в виде файла.

Это задание не отличается особой эффективностью. Согласно конфигурации компонент `ItemReader` выполнит последовательное считывание в одном потоке. Для задания `Job` можно сконфигурировать компонент исполнителя `TaskExecutor`, и среда Spring Batch произведет параллельное считывание, эффективно разбив время считывания на столько потоков, сколько будет поддерживать сконфигурированный компонент `TaskExecutor`.

Повторные попытки запуска сбойных заданий могут быть предприняты с записанных смещений или же считывания можно распараллелить с помощью TaskExecutor, но совмещать одно с другим нельзя! Дело в том, что смещение хранится в локальной области потока и, следовательно, приведет к искажению значений, видимых в других потоках.

Среда Spring Batch работает с сохранением состояния. В ней ведутся таблицы метаданных для всех заданий, запущенных в базе. Последняя, кроме всего прочего, записывает, насколько продвинулось выполнение задания, каков его код выхода, были ли пропущены конкретные строки (и привело ли это к отмене задания, или же строки были просто пропущены). Операторы (либо автономные агенты) могут воспользоваться этой информацией, чтобы принять решение о том, как поступать далее: перезапускать задание или вмешаться вручную.

Все запускается довольно просто. Когда инициируется автоматическое конфигурирование Spring Boot для Spring Batch, выполняется поиск `DataSource` и в автоматическом режиме предпринимаются попытки создания соответствующих таблиц метаданных, основанных на схеме пути к классам (пример 11.2).

Пример 11.2. Таблицы метаданных Spring Batch в MySQL

```
mysql> show tables;
+-----------------------------+
| Tables_in_batch             |
+-----------------------------+
| BATCH_JOB_EXECUTION         |
| BATCH_JOB_EXECUTION_CONTEXT |
| BATCH_JOB_EXECUTION_PARAMS  |
| BATCH_JOB_EXECUTION_SEQ     |
```

```
| BATCH_JOB_INSTANCE           |
| BATCH_JOB_SEQ                |
| BATCH_STEP_EXECUTION         |
| BATCH_STEP_EXECUTION_CONTEXT |
| BATCH_STEP_EXECUTION_SEQ     |
+------------------------------+
9 rows in set (0.00 sec)
```

В данном примере были сконфигурированы два уровня отказоустойчивости: определенный этап может повторяться два раза, прежде чем будет зафиксирована ошибка. Мы используем сторонний веб-сервис, который доступен не всегда. Доступность сервиса можно сымитировать, отключив машину от Интернета. Наблюдать отказ подключения можно через подкласс `HttpStatusCodeException`, после чего повторить попытку. Кроме того, нужно пропустить записи, не прошедшие проверку, поэтому политика пропуска сконфигурирована таким образом, что при каждой выдаче исключения, которое может быть присвоено типу `InvalidEmailException`, пропускается обработка текущей записи.

Изучим конфигурирование отдельных этапов. Начнем с настройки (пример 11.3).

Пример 11.3. Конфигурирование первого этапа Step, имеющего только Tasklet

```
package processing;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;

@Configuration
class Step1Configuration {

 @Bean
 Tasklet tasklet(JdbcTemplate jdbcTemplate) {

  Log log = LogFactory.getLog(getClass());

  return (contribution, chunkContext) -> {  ❶
   log.info("starting the ETL job.");
   jdbcTemplate.update("delete from PEOPLE");
   return RepeatStatus.FINISHED;
  };
 }
}
```

❶ `Tasklet` — универсальная функция обратного вызова, в которой можно делать все что угодно. В нашем случае корректируется таблица `PEOPLE` путем удаления всех данных.

`Tasklet` корректирует таблицу для второго этапа, на котором данные считываются из входного файла, проверяется адрес электронной почты, а затем результаты записываются в только что очищенную таблицу БД. На втором этапе данные забираются из `.csv`-файла с тремя столбцами: с именем человека, возрастом и адресом электронной почты (пример 11.4).

Пример 11.4. Содержимое .csv-файла, из которого забираются данные

```
tam mie,30,tammie@email.com
srinivas,35,srinivas@email.com
lois,53,lois@email.com
bob,26,bob@email.com
jane,18,jane@email.com
jennifer,20,jennifer@email.com
luan,34,luan@email.com
toby,24,toby@email.com
toby,24,toby@email.com
...
```

На данном этапе демонстрируется конфигурирование типовых компонентов `ItemReader`, `ItemProcessor` и `ItemWriter` (пример 11.5).

Пример 11.5. Считывание записей из .csv-файла и их загрузка в таблицу базы данных

```
package processing;

import org.springframework.batch.core.configuration.annotation.StepScope;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.batch.item.database.JdbcBatchItemWriter;
import org.springframework.batch.item.database.builder.
JdbcBatchItemWriterBuilder;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.Resource;
import processing.email.EmailValidationService;
import processing.email.InvalidEmailException;

import javax.sql.DataSource;

@Configuration
class Step2Configuration {

 @Bean
 @StepScope
 ❶
 FlatFileItemReader<Person> fileReader(
  @Value("file://#{jobParameters['input']}") Resource in)  ❷
```

```
  throws Exception {

  ❸
  return new FlatFileItemReaderBuilder<Person>().name("file-reader")
   .resource(in).targetType(Person.class).delimited().delimiter(",")
   .names(new String[] { "firstName", "age", "email" }).build();
 }

 @Bean
 ItemProcessor<Person, Person> emailValidatingProcessor(
  EmailValidationService emailValidator) {  ❹
  return item -> {
   String email = item.getEmail();
   if (!emailValidator.isEmailValid(email)) {
     throw new InvalidEmailException(email);
   }
   return item;
   };
 }

 @Bean
 JdbcBatchItemWriter<Person> jdbcWriter(DataSource ds) { ❺
  return new JdbcBatchItemWriterBuilder<Person>()
   .dataSource(ds)
   .sql(
    "insert into PEOPLE( AGE, FIRST_NAME, EMAIL)"
     + " values (:age, :firstName, :email)").beanMapped().build();
 }
}
```

❶ Bean-компоненты, снабженные аннотацией `@StepScope`, не являются синглтонами. Они создаются заново при каждом запуске экземпляра задания.

❷ В аннотации `@Value` для получения входных параметров задания из контекста Spring Batch `jobParameters` используется язык выражений Spring Expression Language. Для пакетных заданий это обычная практика: задание можно запустить сегодня с указанием на файл, отражающий сегодняшнюю дату, и то же самое задание можно запустить завтра с другим файлом для другой даты.

❸ Используемый в среде Spring Batch предметно-ориентированный язык конфигурирования Java (Java configuration DSL) предоставляет удобные API компоновщика для настройки типовых реализаций компонентов `ItemReader` и `ItemWriter`. Здесь выполняется конфигурирование компонента `ItemReader`, считывающего из файла строку с запятыми в качестве разделителей и отображающего столбцы на имена, а те, в свою очередь, отображаются на поля нашего POJO-объекта `Person`. Затем поля накладываются на экземпляр POJO-объекта `Person`, который возвращается из `ItemReader` для накопления.

❹ Настраиваемый компонент `ItemProcessor` просто передает полномочия нашей реализации `EmailValidationService`, которая, в свою очередь, вызывает REST API.

❺ `JdbcBatchItemWriter` получает результаты `ItemProcessor<Person,Person>` и записывает их в исходное хранилище данных SQL, используя предоставляемую нами SQL-инструкцию. Поименованные параметры для SQL-инструкции соответствуют свойствам Java-Bean в экземпляре POJO-объекта `Person`, созданного компонентом `ItemProcessor`.

Усвоение данных завершено. На завершающем этапе результаты будут проанализированы, чтобы определить распространенность в записях того или иного возраста (пример 11.6).

Пример 11.6. Анализ данных и отправка отчета о результатах в выходной файл

```
package processing;
import org.springframework.batch.core.configuration.annotation.StepScope;
import org.springframework.batch.item.database.JdbcCursorItemReader;
import org.springframework.batch.item.database.builder.
JdbcCursorItemReaderBuilder;
import org.springframework.batch.item.file.FlatFileItemWriter;
import org.springframework.batch.item.file.builder.FlatFileItemWriterBuilder;
import org.springframework.batch.item.file.transform.DelimitedLineAggregator;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.Resource;

import javax.sql.DataSource;
import java.util.Collections;
import java.util.Map;

@Configuration
class Step3Configuration {

❶
@Bean
JdbcCursorItemReader<Map<Integer, Integer>> jdbcReader(DataSource dataSource) {
 return new JdbcCursorItemReaderBuilder<Map<Integer, Integer>>()
  .dataSource(dataSource)
  .name("jdbc-reader")
  .sql("select COUNT(age) c, age a from PEOPLE group by age")
  .rowMapper(
   (rs, i) -> Collections.singletonMap(rs.getInt("a"), rs.getInt("c")))
  .build();
}

❷
@Bean
@StepScope
FlatFileItemWriter<Map<Integer, Integer>> fileWriter(
 @Value("file://#{jobParameters['output']}") Resource out) {
 //@formatter:off
```

```
  DelimitedLineAggregator<Map<Integer, Integer>> aggregator =
    new DelimitedLineAggregator<Map<Integer, Integer>>() {
   {
    setDelimiter(",");
    setFieldExtractor(ageAndCount -> {
     Map.Entry<Integer, Integer> next = ageAndCount.entrySet().iterator()
      .next();
     return new Object[] { next.getKey(), next.getValue() };
    });
  }
 };
 //@formatter:on

 return new FlatFileItemWriterBuilder<Map<Integer, Integer>>()
  .name("file-writer").resource(out).lineAggregator(aggregator).build();
 }
}
```

❶ Компонент `JdbcCursorItemReader` выполняет запрос, а затем обращается в наборе результатов к каждому результату. Используя тот же объект `RowMapper<T>`, который применялся шаблоном `JdbcTemplate` среды Spring, компонент отображает каждую строку на объект, который нам нужен для процессора и (или) записывателя: один ключ `Map<Integer,Integer>`, отражающий возраст и количество его появления в наборе результатов.

❷ Записыватель содержит данные о состоянии и при каждом запуске задания `Job` должен быть создан заново, поскольку при выполнении каждого задания результаты записываются в файл с другим именем. Компоненту `FlatFileItemWriter` нужен экземпляр `LineAggregator`, чтобы выяснить, как изменить входящий POJO-объект (наш `Map<Integer,Integer>`) и превратить его в столбцы, записываемые в выходной `.csv`-файл.

Теперь осталось только запустить задание! Автоконфигурирование, имеющееся в среде Spring Boot, будет по умолчанию при включении запускать задание. Этому заданию требуются параметры (входные и выходные), так что мы отключили поведение по умолчанию и запустили задание явным образом в экземпляре `CommandLineRunner` (пример 11.7).

Пример 11.7. Точка входа в наше пакетное приложение

```
package processing;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.configuration.annotation.
EnableBatchProcessing;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.web.client.RestTemplate;

import javax.sql.DataSource;
import java.io.File;

@EnableBatchProcessing
@SpringBootApplication
public class BatchApplication {

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

 @Bean
 RestTemplate restTemplate() {
  return new RestTemplate();
 }

 @Bean
 JdbcTemplate jdbcTemplate(DataSource dataSource) {
  return new JdbcTemplate(dataSource);
 }

 ❶
 @Bean
 CommandLineRunner run(JobLauncher launcher, Job job,
  @Value("${user.home}") String home) {
  return args -> launcher.run(job,
   new JobParametersBuilder().addString("input", path(home, "in.csv"))
    .addString("output", path(home, "out.csv")).toJobParameters());
 }

 private String path(String home, String fileName) {
  return new File(home, fileName).getAbsolutePath();
 }
}
```

❶ Код `CommandLineRunner` выполняется при запуске приложения. В нем жестко задаются ссылки на пути к локальной файловой системе, которые передаются в виде экземпляров `JobParameter`.

По умолчанию задание `Job` будет запускаться в синхронном режиме. Метод `JobLauncher#run` возвращает объект `JobExecution`, с помощью которого можно выяснить состояние задания после его завершения. Если у сконфигурированного `JobLauncher` имеется `TaskExecutor`, то задание можно запускать в асинхронном режиме.

Среда Spring Batch предназначена для безопасной обработки больших объемов данных, но пока мы уделяли внимание только одному узлу. В разделе «Удаленное разделение задания Spring Batch на части с помощью рассылки сообщений» далее в этой главе будут рассмотрены приемы такого разделения.

Универсальная абстракция Spring Cloud Task предназначена для управления процесссами и придания свойств отслеживамости тем из них, которые запускаются для полного выполнения, но затем прерываются. Абстракция будет рассмотрена чуть позже, а сейчас хотелось бы отметить следующий момент. Она работает с любым сервисом на основе Spring Boot, в котором определяется реализация компонентов `CommandLineRunner` или `ApplicationRunner`, представляющих собой простые интерфейсы функций обратного вызова. При наличии в Spring bean-компонента они получают обратный вызов с принадлежащим приложению массивом `String[] args` из метода `main(String args[])`. Среда Spring Boot выполняет автоматическое конфигурирование компонента `CommandLineRunner`, запускающего любые экземпляры задания Spring Batch `Job` в контексте Spring-приложения. Получается, задания Spring Batch уже являются первыми кандидатами на запуск и управление в качестве заданий, выполняемых с помощью абстракции Spring Cloud Task! Более подробно этот вопрос будет рассмотрен в разделе «Управление задачами» далее в этой главе.

Диспетчеризация

Возникает вопрос: как организовать диспетчеризацию заданий? Если они относятся к разряду развертываемых объемных `.jar`-файлов, извлекающих свою конфигурацию из окружающих источников (наподобие параметров командной строки или переменных среды окружения), то при наличии только одного узла вполне может оказаться, что достаточно прибегнуть к старой доброй команде `cron`.

Но при необходимости более детализированно управлять способами диспетчеризации заданий можно просто воспользоваться `ScheduledExecutorService` в JDK или даже немного поднять уровень абстрагирования и применить имеющуюся в Spring аннотацию `@Scheduled`, которая, в свою очередь, приведет к делегированию полномочий экземпляру `java.util.concurrent.Executor`. Основной недостаток такого подхода заключается в отсутствии учета выполнения или невыполнения задания. Встроенное понятие кластера в нем отсутствует. Что будет в случае отказа того узла, на котором выполняется задание? Будет ли задание восстановлено и перезапущено на другом узле? Что, если потребуется как-то восстанавливать отказавший узел? Как справиться с классической проблемой раздвоения роли главной управляющей системы, когда два узла, каждый из которых работает в предположении, что он является лидером, в конечном итоге работают одновременно?

Существуют коммерческие диспетчеры, например BMC, Flux Scheduler или Autosys. Все они вполне удачно справляются с планировкой рабочих нагрузок в кластере независимо от типа задания. Функционируют они и в облачной среде,

хотя, возможно, вопреки вашим надеждам, и не так легко. Если же требуется более жесткий контроль над диспетчеризацией и жизненными циклами ваших заданий, то можете ознакомиться с интеграцией среды Spring с диспетчером Quartz Enterprise Job (http://www.quartz-scheduler.org/). Он неплохо работает в кластере и должен иметь все необходимое для выполнения задания. Кроме того, он является средством с открытым кодом и достаточно легко действует в облачной среде.

Можно воспользоваться и другим подходом, который заключается *в выборе руководящего инструмента* для наделения узлов в кластере лидирующими полномочиями и для их смещения с этой роли. Лидирующий узел должен иметь отслеживаемое состояние, в противном случае возникнет риск выполнения одного и того же задания дважды. В среде Spring Integration имеется абстракция, предназначенная для поддержки вариантов применения, связанных с выбором лидера и распределенными блокировками, с делегированием реализаций Apache Zookeeper, Hazelcast и др. Это упрощает смену руководства с передачей его от одного узла к другому в режиме транзакции. Среда предоставляет реализации, взаимодействующие с Apache Zookeeper, Hazelcast и Redis. Такой узел, работающий в роли лидера, будет планомерно распределять спектр задач по другим узлам кластера. Обмен данными между узлами может сводиться к простой рассылке сообщений. Вопросы такой рассылки будут рассматриваться в текущей главе чуть позже.

Существует множество ответов, но стоит подчеркнуть, что особого интереса сам вопрос не представляет, поскольку сегодня многие рабочие нагрузки управляются событиями, обходясь без диспетчера заданий.

Удаленное разделение задания Spring Batch на части с помощью рассылки сообщений

Ранее мы уже выяснили, что `TaskExecutor` упрощает распараллеливание операций считывания в задании `Job`, выполняемом в среде Spring Batch. Эта среда также поддерживает распараллеливание операций записи, для чего используется два механизма: *удаленное разделение* (remote partitioning) и *удаленное фрагментирование* (remote chunking).

С функциональной точки зрения оба механизма передают управление одним из этапов другому узлу кластера, где узлы обычно соединены через среду рассылки сообщений (то есть через экземпляры `MessageChannel` среды Spring).

Удаленное разделение публикует сообщение для другого узла, содержащее диапазон записей для считывания (например, 0–100, 101–200 и т. д.). Там фактически запускаются `ItemReader`, `ItemProcessor` и (или) `ItemWriter`. Этот подход требует, чтобы рабочий узел имел полный доступ ко всем ресурсам лидирующего узла. Если такой узел должен, к примеру, считать диапазон записей из файла, то ему потребуется иметь доступ к этому файлу. Состояния, возвращенные из сконфигурированных

рабочих узлов, скапливаются на лидирующем узле. Следовательно, если задание связано с вводом-выводом данных в `ItemReader` и `ItemWriter`, то отдавайте предпочтение удаленному разделению.

Удаленное фрагментирование похоже на удаленное разделение, за исключением того, что данные считываются на лидирующем узле и отправляются по сети рабочему узлу для обработки. Затем результаты возвращаются на лидирующий узел для записи. Следовательно, если ваше задание связано с большим объемом работы центрального процессора в `ItemProcessor`, то отдавайте предпочтение удаленному фрагментированию.

Как удаленные разделение, так и фрагментирование придают эластичность таким платформам, как Cloud Foundry. Можно запустить столько узлов, сколько требуется для выполнения обработки, запустить задание, а затем высвободить ненужные узлы.

Spring Batch вдохновила Майкла Минеллу (Michael Minella) на превосходную работу по подробному объяснению всех тонкостей, присущих этой среде, в докладе для сообщества пользователей языка программирования Java (Chicago Java Users Group) в 2012 году (https://www.youtube.com/watch?v=CYTj5YT7CZU).

Пакетная обработка почти всегда связана с операциями ввода-вывода, поэтому приложений, подходящих для удаленного разделения, намного больше, чем приложений для удаленного фрагментирования (хотя у вас может сложиться несколько иное соотношение). Рассмотрим пример. Чтобы удаленное разделение заработало, лидирующий и рабочий узлы должны иметь доступ к `JobRepository`, хранилищу Spring Batch, владеющему состояниями различных экземпляров среды. В свою очередь, `JobRepository` нуждается в ряде других компонентов. Поэтому любое приложение с удаленным разделением будет иметь код, выполняемый только на рабочих узлах, код, выполняемый только на лидирующем узле, и код, выполняемый на обоих узлах. Среда Spring, к нашей радости, предоставляет естественный механизм обеспечения этого разделения в рамках единой кодовой базы — профили! Они позволяют пометить конкретные объекты и включать и выключать их при определенных условиях в ходе выполнения программы (пример 11.8).

Пример 11.8. Для этого приложения мы определили набор профилей

```
package partition;

public class Profiles {

 public static final String WORKER_PROFILE = "worker";  ❶

 public static final String LEADER_PROFILE = "!" + WORKER_PROFILE;  ❷
}
```

❶ Это профиль для рабочих узлов.

❷ Профиль, активный при *неактивности* рабочего профиля `worker`.

В приложении с удаленным разделением имеется один или несколько рабочих узлов и один лидирующий узел. Последний координирует выполнение задания с рабочими узлами, используя рассылку сообщений. В частности, общается с ними с помощью экземпляров `MessageChannel`, которые могут быть экземплярами `MessageChannel`, приводимыми в действие средой Spring Cloud Stream, подключенными к такому поставщику сообщений, как RabbitMQ или Apache Kafka.

Рассмотрим простое задание `Job` среды Spring Batch, считывающее все записи из одной таблицы `PEOPLE` и копирующее их в другую пустую таблицу `NEW_PEOPLE`. Этот несложный пример содержит всего один этап, и мы, не отвлекаясь на предметную область задания, можем сфокусироваться на компонентах, необходимых для реализации решения.

В точке входа в приложение активируются Spring Batch и Spring Integration, а также конфигурируются `JdbcTemplate` и исходное средство опроса, которое будут использовать компоненты Spring Integration, периодически опрашивающие нижестоящий компонент (пример 11.9).

Пример 11.9. Точка входа в приложение

```
package partition;

import
 org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.integration.annotation.IntegrationComponentScan;
import org.springframework.integration.dsl.core.Pollers;
import org.springframework.integration.scheduling.PollerMetadata;
import org.springframework.jdbc.core.JdbcTemplate;

import javax.sql.DataSource;
import java.util.concurrent.TimeUnit;

@EnableBatchProcessing
❶
@IntegrationComponentScan
@SpringBootApplication
public class PartitionApplication {

 public static void main(String args[]) {
  SpringApplication.run(PartitionApplication.class, args);
 }
 ❷
 @Bean(name = PollerMetadata.DEFAULT_POLLER)
 PollerMetadata defaultPoller() {
```

```
  return Pollers.fixedRate(10, TimeUnit.SECONDS).get();
 }

 @Bean
 JdbcTemplate jdbcTemplate(DataSource dataSource) {
  return new JdbcTemplate(dataSource);
 }
}
```

❶ Активация Spring Batch и Spring Integration.

❷ Указание на исходное, глобальное средство опроса для реализаций `MessageChannel`, требующих выполнения этого действия.

У приложения имеется одно задание Spring Batch с двумя этапами. Первый, `Tasklet`, подготавливает базу данных, опустошая ее содержимое (MySQL-инструкцией `truncate`) (пример 11.10). Второй, конфигурация которого показана в примере 11.11, является разделенным.

Пример 11.10. Конфигурация разделенного этапа

```
package partition;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.configuration.annotation.
JobBuilderFactory;
import org.springframework.batch.core.launch.support.RunIdIncrementer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Configuration
@Profile(Profiles.LEADER_PROFILE)
❶
class JobConfiguration {

 @Bean
 Job job(JobBuilderFactory jbf, LeaderStepConfiguration lsc) {
  return jbf.get("job").incrementer(new RunIdIncrementer())
   .start(lsc.stagingStep(null, null))  ❷
   .next(lsc.partitionStep(null, null, null, null))  ❸
   .build();
 }
}
```

❶ Задание `Job` присутствует только в профиле лидирующего узла.

❷ `Tasklet` приводит таблицу `NEW_PEOPLE` в исходное состояние.

❸ Разделенный этап `Step` раздает работу другим узлам.

Разделенный этап `Step` ничего не делает на лидирующем узле, действует как некий прокси-узел, который диспетчеризует выполнение задания на рабочих узлах. Данный этап `Step` должен знать количество доступных рабочих узлов (размер сетки).

В примере 11.10 мы добавили свойство для жестко заданного значения размера сетки, но размер пула рабочих узлов должен вычисляться повторно в динамическом режиме с помощью, например, динамического поиска экземпляров сервиса с использованием абстракции Spring Cloud `DiscoveryClient`, опроса API базовой платформы (скажем, Cloud Foundry API Client (https://docs.cloudfoundry.org/buildpacks/java/java-client.html)) на предмет размера.

Пример 11.11. Конфигурация для разделенного этапа

```
package partition;

import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration.annotation.
StepBuilderFactory;
import org.springframework.batch.core.explore.JobExplorer;
import org.springframework.batch.core.partition.PartitionHandler;
import org.springframework.batch.core.partition.support.Partitioner;
import
 org.springframework.batch.integration.partition.MessageChannelPartitionHandler;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.core.MessagingTemplate;
import org.springframework.jdbc.core.JdbcOperations;
import org.springframework.jdbc.core.JdbcTemplate;

@Configuration
class LeaderStepConfiguration {

 ❶
 @Bean
 Step stagingStep(StepBuilderFactory sbf, JdbcTemplate jdbc) {
  return sbf.get("staging").tasklet((contribution, chunkContext) -> {
   jdbc.execute("truncate NEW_PEOPLE");
   return RepeatStatus.FINISHED;
  }).build();
 }

 ❷
 @Bean
 Step partitionStep(StepBuilderFactory sbf, Partitioner p, PartitionHandler ph,
  WorkerStepConfiguration wsc) {
  Step workerStep = wsc.workerStep(null);
  return sbf.get("partitionStep").partitioner(workerStep.getName(), p)
   .partitionHandler(ph).build();
 }

 ❸
 @Bean
 MessageChannelPartitionHandler partitionHandler(
```

```
  @Value("${partition.grid-size:4}") int gridSize,
  MessagingTemplate messagingTemplate, JobExplorer jobExplorer) {
  //@formatter:off
  MessageChannelPartitionHandler ph =
     new MessageChannelPartitionHandler();
  //@formatter:on
  ph.setMessagingOperations(messagingTemplate);
  ph.setJobExplorer(jobExplorer);
  ph.setStepName("workerStep");
  ph.setGridSize(gridSize);
  return ph;
 }

 ❹
 @Bean
 MessagingTemplate messagingTemplate(LeaderChannels channels) {
  return new MessagingTemplate(channels.leaderRequestsChannel());
 }

 ❺
 @Bean
 Partitioner partitioner(JdbcOperations jdbcTemplate,
  @Value("${partition.table:PEOPLE}") String table,
  @Value("${partition.column:ID}") String column) {
  return new IdRangePartitioner(jdbcTemplate, table, column);
 }
}
```

❶ Первый этап `Step` в задании `Job` на лидирующем узле приводит базу данных в исходное состояние.

❷ Следующему этапу, выполняющему разделение, нужно знать, какой рабочий этап запускать на удаленных узлах `Partitioner` и `PartitionHandler`.

❸ Узел `PartitionHandler` отвечает за получение `StepExecution` на лидирующем узле и разделение его на коллекцию экземпляров `StepExecution`, раздаваемую рабочим узлам. `PartitionHandler` требует сведений о количестве рабочих узлов, чтобы спектр задач можно было разделить соответствующим образом. Конкретно эта реализация `PartitionHandler` координирует выполнение задания на рабочих узлах, используя экземпляры `MessageChannel`...

❹ ...и поэтому требует от реализации `MessagingOperations` беспрепятственной оправки и (или) получения сообщений.

❺ Узел `Partitioner` отвечает за разделение рабочей нагрузки. В данном случае разделение сделано по некоторым основным участкам идентификаторов записей в исходной таблице `PEOPLE`. Возможно, из всего приведенного в примере кода это наиболее трудоемкая часть, на которую обычно накладываются особенности предметной области вашего бизнеса и приложения.

Рабочие узлы забирают работу у поставщика и направляют запросы компоненту `StepExecutionRequestHandler`, который затем использует компонент `StepLocator`

с целью выявить в контексте рабочего приложения фактический этап для выполнения (пример 11.12).

Пример 11.12. Конфигурация для разделенного этапа

```
package partition;

import org.springframework.batch.core.explore.JobExplorer;
import org.springframework.batch.core.step.StepLocator;
import org.springframework.batch.integration.partition.BeanFactoryStepLocator;
import org.springframework.batch.integration.partition.StepExecutionRequest;
import org.springframework.batch.integration.partition.StepExecutionRequestHandler;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.support.GenericHandler;
import org.springframework.messaging.MessageChannel;

@Configuration
@Profile(Profiles.WORKER_PROFILE)
❶
class WorkerConfiguration {

 ❷
 @Bean
 StepLocator stepLocator() {
  return new BeanFactoryStepLocator();
 }

 ❸
 @Bean
 StepExecutionRequestHandler stepExecutionRequestHandler(JobExplorer explorer,
  StepLocator stepLocator) {
  StepExecutionRequestHandler handler = new StepExecutionRequestHandler();
  handler.setStepLocator(stepLocator);
  handler.setJobExplorer(explorer);
  return handler;
 }

 ❹
 @Bean
 IntegrationFlow stepExecutionRequestHandlerFlow(WorkerChannels channels,
  StepExecutionRequestHandler handler) {

  MessageChannel channel = channels.workerRequestsChannels();
  //@formatter:off
  GenericHandler<StepExecutionRequest> h =
          (payload, headers) -> handler.handle(payload);
  //@formatter:on
```

```
  return IntegrationFlows.from(channel)
   .handle(StepExecutionRequest.class, h)
   .channel(channels.workerRepliesChannels()).get();
 }
}
```

1. Эти объекты должны быть только на рабочем узле `worker` и под профилем `worker`.
2. Компонент `StepLocator` отвечает за обнаружение реализации этапа `Step` по имени. В данном случае `Step` обнаруживается путем прохода по всем bean-компонентам в реализации `BeanFactory`.
3. Компонент `StepExecutionRequestHandler` — партнер рабочего узла для узла `PartitionHandler` на лидирующем узле: он принимает входящие запросы и превращает их в выполнение заданного этапа `Step`, а результаты отправляет в ответе.
4. Компонент обработчика запросов `StepExecutionRequestHandler` не знает об экземплярах каналов `MessageChannel`, по которым поступят входящие запросы `StepExecutionRequests`, поэтому, чтобы запустить в ответ на сообщение метод `#handle(StepExecutionRequest)`, имеющийся в обработчике запросов `StepExecutionRequestHandler`, и обеспечить тем самым отправку возвращаемого значения в виде сообщения по каналу ответа, используется Spring Integration.

И наконец, в примере 11.13 показана активация текущего этапа с информацией о том, какие данные следует считать. Притом важно, чтобы у считывающего компонента был доступ к тому же состоянию, которое имеется на лидирующем узле (в данном случае к JDBC `DataSource`).

Пример 11.13. Конфигурация для разделенного этапа

```
package partition;

import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.configuration.annotation.StepScope;
import org.springframework.batch.item.database.JdbcBatchItemWriter;
import org.springframework.batch.item.database.JdbcPagingItemReader;
import org.springframework.batch.item.database.Order;
import org.springframework.batch.item.database.builder.JdbcBatchItemWriterBuilder;
import org.springframework.batch.item.database.support.MySqlPagingQueryProvider;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;
import java.util.Collections;

@Configuration
```

```
class WorkerStepConfiguration {

 ❶
 @Value("${partition.chunk-size}")
 private int chunk;

 ❷
 @Bean
 @StepScope
 JdbcPagingItemReader<Person> reader(DataSource dataSource,
  @Value("#{stepExecutionContext['minValue']}") Long min,
  @Value("#{stepExecutionContext['maxValue']}") Long max) {

  MySqlPagingQueryProvider queryProvider = new MySqlPagingQueryProvider();
  queryProvider
   .setSelectClause("id as id, email as email, age as age, first_name as
     firstName");
  queryProvider.setFromClause("from PEOPLE");
  queryProvider.setWhereClause("where id >= " + min + " and id <= " + max);
  queryProvider.setSortKeys(Collections.singletonMap("id", Order.ASCENDING));

  JdbcPagingItemReader<Person> reader = new JdbcPagingItemReader<>();
  reader.setDataSource(dataSource);
  reader.setFetchSize(this.chunk);
  reader.setQueryProvider(queryProvider);
  reader.setRowMapper((rs, i) -> new Person(rs.getInt("id"), rs.getInt("age"),
   rs.getString("firstName"), rs.getString("email")));
  return reader;
 }

 ❸
 @Bean
 JdbcBatchItemWriter<Person> writer(DataSource ds) {
  return new JdbcBatchItemWriterBuilder<Person>()
   .beanMapped()
   .dataSource(ds)
   .sql(
    "INSERT INTO NEW_PEOPLE(age,first_name,email) VALUES(:age, :firstName,
       :email )")
   .build();
 }

 ❹
 @Bean
 Step workerStep(StepBuilderFactory sbf) {
  return sbf.get("workerStep").<Person, Person>chunk(this.chunk)
  .reader(reader(null, null, null)).writer(writer(null)).build();
 }
}
```

❶ Это задание Spring Batch, которое выполняется в конце дня, так что реализация `JdbcPagingItemReader` занимается разделением всего объема записей на части.

❷ Обратите внимание: мы задействуем `JdbcPagingItemReader`, а не реализации, подобные `JdbcCursorItemReader`, поскольку не можем совместно использовать указатель результата JDBC на всех рабочих узлах. В конечном итоге обе реализации позволяют разделить большой набор результатов JDBC `ResultSet` на части меньшего размера, и полезно знать, где и когда применять каждую из этих реализаций. Компонент `ItemReader` получит в контексте выполнения `ExecutionContext`, отправленного из `PartitionHandler`, отображение ключей на значения, которое предоставляет полезную информацию, включая в данном случае диапазоны считываемых строк. Поскольку эти значения для каждого выполнения этапа `Step` уникальны, то компоненту `ItemReader` была дана аннотация `@StepScope`.

❸ Компонент `JdbcBatchItemWriter` записывает поля `Person` POJO-формата в базу данных путем отображения POJO-свойств на поименованные параметры в SQL-инструкции.

❹ И наконец, мы создаем рабочий этап `Step`. Теперь это вам уже хорошо знакомо.

В итоге компоненты приложения подключаются к реализациям `MessageChannel`. Три из таких определений — `leaderRequests`, `workerRequests` и `workerReplies` — даны с использованием Spring Cloud Stream и (в этом примере) RabbitMQ. Последнее определение `MessageChannel` — `leaderRepliesAggregatedChannel` — служит только для соединения двух компонентов в пространстве оперативной памяти. Чтобы увидеть эти определения, вам предстоит самостоятельно ознакомиться с исходным кодом `LeaderChannels`, `WorkerChannels` и `application.properties`; учитывая наше предыдущее рассмотрение среды Spring Cloud Stream, будет излишним уделять здесь этому внимание.

Вам понадобится настройка таблицы `PEOPLE`, а также ее идентично сконфигурированный дубликат `NEW_PEOPLE`. Запустите с помощью профиля `worker` несколько экземпляров рабочих узлов. Затем запустите лидирующий узел, не указывая конкретный профиль. Когда все перечисленное будет запущено, можно запускать экземпляр задания. В этом примере задание `Job` не запускается, но можно инициировать запуск через конечную точку REST, включающую исходный код: `curl -d{} http://localhost:8080/migrate`.

Управление задачами

Среда Spring Boot знает, что делать с вашим заданием `Job`; при запуске приложения она запускает все экземпляры `CommandLineRunner`, включая предоставленный Spring Boot экземпляр автоматического конфигурирования Spring Batch. Но со стороны совсем не видно никакого логического способа узнать, как именно нужно запустить это задание. Мы не можем знать, что в нем дается описание рабочей нагрузки, которая завершится и создаст состояние выхода. У нас нет общей инфраструктуры, фиксирующей время запуска и окончания

задачи. Нет и инфраструктуры, которая поддержала бы нас при выдаче исключения. Все эти понятия относятся к работе Spring Batch. А как справиться с экземплярами, не имеющими отношения к заданиям `Job` Spring Batch, например с `CommandLineRunner` или `ApplicationRunner`? Здесь на помощь приходит среда Spring Cloud Task. Как следует из названия, она предоставляет способ идентификации, выполнения и детального исследования задач!

Задача (task) — это нечто запускаемое и имеющее ожидаемое состояние выхода. Задачи являются идеальными абстракциями для любого эфемерного процесса или рабочей нагрузки, на выполнение которых требуется атипично большое время (больше, чем затрачивается на идеальную транзакцию в основном потоке запроса и ответа сервиса; на задачу может уйти несколько секунд, часов или даже дней!). В задачах даются описания однократно запускаемых рабочих нагрузок (в ответ на возникновение события) или запускаемых по расписанию. К наиболее распространенным примерам относятся:

- чьи-либо запросы на создание и отправку системой сообщения электронной почты с требованием переустановки пароля (Reset Password);
- задание `Job` среды Spring, запускаемое при поступлении в каталог нового файла;
- приложение, которое периодически собирает мусор из ненужных файлов либо проверяет данные на несогласованность или журналы очереди сообщений;
- создание документа (или отчета) в динамическом режиме;
- перекодировка мультимедиа.

Ключевой аспект задачи — предоставление унифицированного интерфейса для параметризации. Задачи среды Spring Cloud Task принимают параметры и конфигурационные свойства Spring Boot. При указании параметров Spring Cloud Task надежно сохранит их в виде аргументов `CommandLineRunner` или `ApplicationRunner` либо же в качестве экземпляров Spring Batch `JobParameter`. Задачи можно сконфигурировать с помощью конфигурационных свойств Spring Boot. Вскоре будет показано, что среда Spring Cloud Data Flow разумна настолько, что позволяет вам исследовать список известных свойств для текущей задачи и отобразить форму относящихся к ней свойств.

Если в задаче нужно воспользоваться интеллектуальной интеграцией со средой Spring Cloud Data Flow, то следует включить процессор конфигурации Spring Boot (org.springframework.boot : spring-boot-configuration-processor) и определить компонент @ConfigurationProperties.

Рассмотрим простой пример (пример 11.14). Запустим новый, абсолютно пустой проект Spring Boot, а затем добавим `org.springframework.cloud : spring-cloud-task-starter`.

Пример 11.14. Автономное приложение Spring Boot, содействующее компоненту CommandLineRunner, который запускает для нас среда Spring Cloud Task

```
package task;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.task.configuration.EnableTask;
import org.springframework.cloud.task.repository.TaskExplorer;
import org.springframework.context.annotation.Bean;
import org.springframework.data.domain.PageRequest;

import java.util.stream.Stream;

❶
@EnableTask
@SpringBootApplication
public class HelloTask {

 private Log log = LogFactory.getLog(getClass());

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

 @Bean
 CommandLineRunner runAndExplore(TaskExplorer taskExplorer) {
  return args -> {
   Stream.of(args).forEach(log::info);

   ❷
   taskExplorer.findAll(new PageRequest(0, 1)).forEach(
    taskExecution -> log.info(taskExecution.toString()));
  };
 }
}
```

❶ Активация Spring Cloud Task.

❷ Внедрение Spring Cloud Task `TaskExplorer` для исследования хода выполнения текущей запущенной задачи.

При запуске кода примера аргумент `CommandLineRunner` выполняется так же, как от него это и ожидалось в приложении, не основанном на применении задач. Но тут весьма кстати у нас появляется возможность внедрить `TaskExplorer` по динамически исследуемому состоянию запущенной задачи (или фактически любых задач). Компоненту `TaskExplorer` известно о существовании в контексте вашего приложения `CommandLineRunner`, `ApplicationRunner` и экземпляров `Job` среды Spring Batch.

Вы получаете поддержку для рабочих нагрузок Spring Batch `Job` с пакетной интеграцией Spring Cloud Task (`org.springframework.cloud : spring-cloud-task-batch`).

К теме применения Spring Cloud Task мы еще вернемся, когда будем рассматривать вопрос о потоке данных Spring Cloud Data Flow.

Интеграция с рабочим потоком, ориентированная на процесс

Хотя Spring Batch дает элементарные возможности, связанные с рабочим потоком, цель этой среды — поддержка обработки больших наборов данных, а не объединение автономных агентов и агентов-людей в полноценный рабочий поток. Использование *рабочего потока* — пример явного моделирования прохождения работы через систему автономных агентов (и агентов-людей!). В его системах определяется машина состояний и конструкции для моделирования прохождения состояния по направлению к цели. Системы рабочего потока проектируются в качестве как технического компонента, так и описания бизнес-процесса высокого уровня.

Рабочий поток во многом перекликается с идеями управления бизнес-процессом и его моделирования (и обе идеи по непонятной причине сведены в аббревиатуру BPM). BPM означает наличие технической возможности описания и запуска процесса рабочего потока, но также и управленческую дисциплину автоматизации бизнеса. Аналитики используют BPM для идентификации бизнес-процессов, которые моделируются ими и выполняются с помощью таких машин рабочих потоков, как Activiti (https://www.activiti.org/).

Системы рабочих потоков поддерживают простые приемы моделирования процессов. Как правило, системы предоставляют инструменты, используемые в режиме проектирования, которые облегчают визуальное моделирование. Их основная цель — свести все к компоненту, минимально определяющему процесс как для бизнеса, так и для технологов. Модель процесса не является непосредственно выполняемым кодом, это серии этапов. Предоставление соответствующего поведения для различных этапов возлагается на проектировщиков. Системы рабочих потоков обычно обеспечивают способ сохранения и запроса состояния различных процессов. Подобно Spring Batch, такие системы предоставляют встроенный механизм для проектирования компенсирующего поведения в случаях отказов.

В плане проектирования системы рабочих потоков помогают избавить сервисы и объекты от сохранения состояния и освободить от того, что в противном случае было бы несущественным состоянием процесса. Мы видели множество систем, где поступательные изменения сущностного объекта через мимолетные разовые процессы были представлены в качестве булевых значений (`is_enrolled`, `is_fulfilled` и т. п.) в объекте и в самих базах данных. Эти булевы значения, применяемые в виде флагов, запутывали конструкцию сущностного объекта при весьма сомнительной пользе.

Системы рабочих потоков отображают роли на последовательность этапов. Эти роли более или менее соответствуют дорожкам в языке UML. Механизм рабочего потока, подобно дорожке, может предоставить описание списков *человеческих* задач, а также автономной работы.

Подходит ли рабочий поток всем и каждому? Конечно же, нет. Мы увидели, что рабочие потоки наиболее полезны тем организациям, которые имеют сложные процессы или ограничения, связанные с правилами или политикой, и нуждаются в способе опроса состояния процесса. Механизм рабочего потока оптимален для партнерских процессов, где люди и сервисы подстраиваются под более обширные бизнес-требования; в качестве примеров можно привести одобрение кредита, соблюдение правовых норм, обзор страхового покрытия, пересмотр документа и печать документа.

Рабочий поток упрощает конструкцию, чтобы освободить вашу бизнес-логику от наслоения состояния, требуемого для поддержки аудита и отчетности процессов. В этом разделе поток рассматривается по той причине, что, будучи похожим на пакетную обработку, он предоставляет эффективный способ устранить сбой в сложном, многоагентном и многоузловом процессе. В качестве надстройки над механизмом рабочего потока можно смоделировать координатор саги выполнения, поскольку механизм рабочего потока сам по себе не является таким координатором. Но правильное использование повлечет массу предполагаемых преимуществ. Далее будет показано, что рабочие потоки также поддаются горизонтальному масштабированию в облачной среде через инфраструктуру рассылки сообщений.

Разберем несложное упражнение. Предположим, имеется процесс регистрации нового пользователя. Бизнесу новые регистрации нужны в качестве показателя. Концептуально процесс регистрации весьма прост: пользователь через форму отправляет новый запрос на регистрацию, который затем должен быть проверен и при наличии ошибок исправлен. Как только форма без ошибок будет принята, должно быть отправлено соответствующее сообщение электронной почты. Пользователю нужно щелкнуть кнопкой мыши на этом сообщении и инициировать известную конечную точку, подтверждая факт адресуемости его электронной почты. Он может сделать это в течение минуты, или недели, или... года! Долговременная транзакция все еще действительна. Можно сделать процесс еще изощреннее и указать в определении нашего рабочего потока истечение срока ожидания и дополнительные действия. Но на данный момент пример и так достаточно интересен, поскольку при достижении цели зачисления нового клиента охватывает работу как компьютера, так и человека (рис. 11.2).

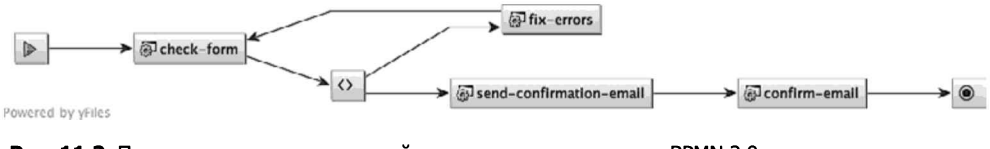


Рис. 11.2. Процесс регистрации, который смоделирован с помощью BPMN 2.0 и показан в просмотре BPMN 2.0, выданном технологией yFiles в интегрированной среде разработки IntelliJ IDEA

Проект Activiti компании Alfresco (https://www.activiti.org/) представляет собой *механизм бизнес-процесса*. Он поддерживает определение процесса в стандарте XML под названием BPMN 2.0, который пользуется надежной поддержкой среди нескольких поставщиков инструментальных средств и интегрированных сред разработки. Создатель бизнес-процесса определяет эти BPMN-процессы, задействуя различные инструменты (от, скажем, WebMethods, Activiti или IBM), и может затем запустить процесс, которому дано определение, в Activiti (или теоретически в любом другом инструменте). Кроме того, Activiti предоставляет среду моделирования, легко поддается развертыванию в автономном режиме или использованию в облачном сервисе и имеет лицензию Apache 2. И, что очень кстати для нас, Activiti также предлагает удобное автоконфигурирование в среде Spring Boot.

Пока шла работа над этой книгой, основные разработчики проекта Activiti создали полностью обратно совместимое ответвление от проекта под названием Flowable (https://www.flowable.org/), на которое получена новая лицензия Apache 2. Если вам нравится Activiti, то вы должны непременно обратить внимание на данное ответвление.

В Activiti в `Process` дается определение первоначального процесса. Компонент `ProcessInstance` является экземпляром отдельно взятого выполнения заданного процесса. Уникальность определения `ProcessInstance` задается имеющимися переменными процесса, параметризирующими выполнение последнего. Их следует представлять как аргументы командной строки, подобные параметрам `JobParameter` для задания `Job` в Spring Batch.

Механизм Activiti, подобно Spring Batch, сохраняет свое состояние выполнения в РБД. При использовании автоконфигурирования Spring Boot и при наличии где-либо в контексте вашего приложения источника данных `DataSource` для вас будут автоматически установлены соответствующие таблицы. Они показаны в примере 11.15.

Пример 11.15. Таблицы метаданных Activiti, созданные в MySQL

```
mysql> show tables;
+----------------------------+
| Tables_in_activiti         |
+----------------------------+
| ACT_EVT_LOG                |
| ACT_GE_BYTEARRAY           |
| ACT_GE_PROPERTY            |
| ACT_HI_ACTINST             |
| ACT_HI_ATTACHMENT          |
| ACT_HI_COMMENT             |
| ACT_HI_DETAIL              |
| ACT_HI_IDENTITYLINK        |
| ACT_HI_PROCINST            |
```

```
| ACT_HI_TASKINST              |
| ACT_HI_VARINST               |
| ACT_ID_GROUP                 |
| ACT_ID_INFO                  |
| ACT_ID_MEMBERSHIP            |
| ACT_ID_USER                  |
| ACT_PROCDEF_INFO             |
| ACT_RE_DEPLOYMENT            |
| ACT_RE_MODEL                 |
| ACT_RE_PROCDEF               |
| ACT_RU_EVENT_SUBSCR          |
| ACT_RU_EXECUTION             |
| ACT_RU_IDENTITYLINK          |
| ACT_RU_JOB                   |
| ACT_RU_TASK                  |
| ACT_RU_VARIABLE              |
+------------------------------+
24 rows in set (0.00 sec)
```

По умолчанию механизмом автоконфигурирования Spring Boot в каталоге `src/main/resources/processes` ожидается любой документ BPMN 2.0. В примере 11.16 показано определение процесса `signup`, выполняемое BPMN 2.0.

Пример 11.16. Определение бизнес-процесса signup.bpmn20.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:activiti="http://activiti.org/bpmn"
        xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
        typeLanguage="http://www.w3.org/2001/XMLSchema"
        expressionLanguage="http://www.w3.org/1999/XPath"
        targetNamespace="http://www.activiti.org/bpmn2.0">

    <process name="signup" id="signup">
        ❶
        <startEvent id="start"/>

        ❷
        <sequenceFlow sourceRef="start" targetRef="check-form"/>

        ❸
        <serviceTask id="check-form" name="check-form"
             activiti:expression="#{checkForm.execute(execution)}"/>

        <sequenceFlow sourceRef="check-form"
          targetRef="form-completed-decision-gateway"/>

        ❹
        <exclusiveGateway id="form-completed-decision-gateway"/>

        <sequenceFlow name="formOK" id="formOK"
```

```
                sourceRef="form-completed-decision-gateway"
                targetRef="send-confirmation-email">
      <conditionExpression xsi:type="tFormalExpression">${formOK == true}
      </conditionExpression>
    </sequenceFlow>

    <sequenceFlow id="formNotOK" name="formNotOK"
                sourceRef="form-completed-decision-gateway"
                targetRef="fix-errors">
      <conditionExpression xsi:type="tFormalExpression">${formOK == false}
      </conditionExpression>
    </sequenceFlow>

    ❺
    <userTask name="fix-errors" id="fix-errors">
        <humanPerformer>
            <resourceAssignmentExpression>
                <formalExpression>customer</formalExpression>
            </resourceAssignmentExpression>
        </humanPerformer>
    </userTask>

    <sequenceFlow sourceRef="fix-errors" targetRef="check-form"/>

    ❻
    <serviceTask id="send-confirmation-email" name="send-confirmation-email"
       activiti:expression="#{sendConfirmationEmail.execute(execution)}"/>

    <sequenceFlow sourceRef="send-confirmation-email"
      targetRef="confirm-email"/>

    ❼
    <userTask name="confirm-email" id="confirm-email">
        <humanPerformer>
            <resourceAssignmentExpression>
                <formalExpression>customer</formalExpression>
            </resourceAssignmentExpression>
        </humanPerformer>
    </userTask>

    <sequenceFlow sourceRef="confirm-email" targetRef="end"/>

    <endEvent id="end"/>
  </process>
</definitions>
```

❶ Первое состояние — `startEvent`. Вполне определенные запуск (`start`) и окончание (`end`) есть у всех процессов.

❷ Элементы `sequenceFlow` служат для механизма Activiti в качестве руководства по переходу к следующему действию. Они задают логику потока и в самой модели рабочего потока представлены в виде линий.

❸ Компонент `serviceTask` является состоянием в процессе. В нашем определении BPMN 2.0 используется атрибут, определяющий действие `activiti:expression` для делегирования обработки методу `execute(ActivityExecution)` в bean-компоненте Spring (называемом `checkForm`). Это поведение активируется при автоконфигурировании Spring Boot.

❹ После отправки и проверки формы метод `checkForm` дает булеву *переменную процесса* по имени `formOK`, чье значение используется для управления решением о переходе ниже по потоку. Переменная потока принадлежит контексту, видимому участникам данного процесса. Переменные процесса могут быть чем угодно, но мы стремимся по возможности максимально их упростить, чтобы их можно было использовать где-то еще в качестве квитанции о получении реальных ресурсов. Этот процесс ожидает одну входящую переменную процесса по имени `customerId`.

❺ Если форма не соответствует требованиям, то работа переходит пользовательской задаче по исправлению ошибок — `fix-errors`. Задача вносится в рабочий список и назначается человеку. Список поддерживается в Activiti через API `Task`. Не составляет труда запросить рабочий список, после чего запустить и завершить задачи. Список предназначен для моделирования работы, возлагаемой на человека. Когда процесс сталкивается с таким состоянием, он приостанавливается в ожидании явного завершения задачи человеком и попадания в последующее состояние. Рабочие потоки возвращаются из задачи `fix-errors` к состоянию `checkForm`. Если теперь форма исправлена (и соответствует требованиям), то работа переходит к следующему состоянию.

❻ Если форма проходит проверку, то работа переходит к сервис-задаче отправки подтверждения по электронной почте — `send-confirmation-email`. Происходит простая передача полномочий другому bean-компоненту Spring, чтобы тот проделал свою работу, возможно, с помощью `SendGrid`.

❼ В сообщении электронной почты должна содержаться ссылка, щелчок на которой вызывает конечную точку HTTP, а та, в свою очередь, инициирует завершение задачи, ожидающей выполнение, что продвигает процесс к завершению.

Процесс довольно прост, но дает четкое представление о движущихся частях системы. Вполне очевидно, что нужен *некий элемент* для приема вводимых пользователем данных и отображения результата в виде записи о клиенте в базе, а также приемлемый идентификатор клиента `customerId`, который можно будет задействовать для извлечения этой записи в нижестоящих компонентах. Запись о клиенте может быть в неподобающем состоянии (например, неверен адрес электронной почты), и со стороны пользователя может потребоваться повторное обращение к ней. Как только все будет готово к дальнейшей работе, состояние позволит перейти к следующему этапу, где мы отправляем на электронную почту уведомление, получение и подтверждение которого приводят процесс в конечное состояние.

Рассмотрим простой REST API, управляющий данным процессом. Для краткости мы не стали заниматься экстраполяцией на клиентов iPhone или HTML5, но указанное действие, несомненно, будет выполнено на следующем этапе. Для наглядности

REST API реализован как можно проще. Очень логичным следующим шагом может стать использование гипермедиа для управления взаимодействием клиента с REST API при переходе от одного состояния к другому (пример 11.17). Подробнее эта возможность рассмотрена в главе 6, посвященной REST API, в разделе «Гипермедиа», где речь шла о надстройке по имени HATEOAS.

Пример 11.17. Контроллер SignupRestController, управляющий процессом

```
package com.example;

import org.activiti.engine.RuntimeService;
import org.activiti.engine.TaskService;
import org.activiti.engine.task.TaskInfo;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.util.Assert;
import org.springframework.web.bind.annotation.*;

import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping("/customers")
class SignupRestController {

 public static final String CUSTOMER_ID_PV_KEY = "customerId";

 private final RuntimeService runtimeService;

 private final TaskService taskService;

 private final CustomerRepository customerRepository;

 private Log log = LogFactory.getLog(getClass());

❶
 @Autowired
 public SignupRestController(RuntimeService runtimeService,
  TaskService taskService, CustomerRepository repository) {
  this.runtimeService = runtimeService;
  this.taskService = taskService;
  this.customerRepository = repository;
 }

 ❷
 @PostMapping
 public ResponseEntity<?> startProcess(@RequestBody Customer customer) {
  Assert.notNull(customer);
  Customer save = this.customerRepository.save(new Customer(customer
```

```
  .getFirstName(), customer.getLastName(), customer.getEmail()));

 String processInstanceId = this.runtimeService.startProcessInstanceByKey(
  "signup",
  Collections.singletonMap(CUSTOMER_ID_PV_KEY, Long.toString(save.getId())))
  .getId();
 this.log.info("started sign-up. the processInstance ID is "
  + processInstanceId);

 return ResponseEntity.ok(save.getId());
}

❸
@GetMapping("/{customerId}/signup/errors")
public List<String> readErrors(@PathVariable String customerId) {
 // @formatter:off
 return this.taskService
   .createTaskQuery()
    .active()
    taskName("fix-errors")
    .includeProcessVariables()
    .processVariableValueEquals(CUSTOMER_ID_PV_KEY, customerId)
    .list()
     .stream()
     .map(TaskInfo::getId)
   .collect(Collectors.toList());
 // @formatter:on
}

❹
@PostMapping("/{customerId}/signup/errors/{taskId}")
public void fixErrors(@PathVariable String customerId,
 @PathVariable String taskId, @RequestBody Customer fixedCustomer) {

 Customer customer = this.customerRepository.findOne(Long
  .parseLong(customerId));
 customer.setEmail(fixedCustomer.getEmail());
 customer.setFirstName(fixedCustomer.getFirstName());
 customer.setLastName(fixedCustomer.getLastName());
 this.customerRepository.save(customer);

 this.taskService.createTaskQuery().active().taskId(taskId)
  .includeProcessVariables()
  .processVariableValueEquals(CUSTOMER_ID_PV_KEY, customerId).list()
  .forEach(t -> {
   log.info("fixing customer# " + customerId + " for taskId " + taskId);
   taskService.complete(t.getId(), Collections.singletonMap("formOK", true));
  });
}

❺
@PostMapping("/{customerId}/signup/confirmation")
public void confirm(@PathVariable String customerId) {
```

```
  this.taskService.createTaskQuery().active().taskName("confirm-email")
   .includeProcessVariables()
   .processVariableValueEquals(CUSTOMER_ID_PV_KEY, customerId).list()
   .forEach(t -> {
    log.info(t.toString());
    taskService.complete(t.getId());
   });
  this.log.info("confirmed email receipt for " + customerId);
 }
}
```

1. Контроллер будет работать с простым хранилищем Spring Data JPA, а также с двумя сервисами, прошедшими автоконфигурирование благодаря поддержке Activiti Spring Boot. Сервис `RuntimeService` позволяет опрашивать механизм управления процессами на предмет запущенных процессов. Сервис `TaskService` дает возможность опрашивать механизм процессов о запущенных задачах, выполняемых людьми, и о списках задач.
2. Первая конечная точка принимает новую клиентскую запись и сохраняет ее. Новоиспеченный клиент подается новому процессу в качестве входной переменной процесса, где в ее роли выступает идентификатор клиента `customerId`. Здесь процесс перетекает к сервису задач, проверяющему форму `check-form serviceTask`, который проведет номинальную проверку допустимости входящего сообщения электронной почты; если проверка пройдет успешно, то по электронной почте будет отправлено подтверждение. В противном случае клиенту придется исправить все ошибки во входных данных.
3. Нам нужно, чтобы при наличии каких-либо ошибок UI останавливал продвижение вперед, поэтому процесс ставит пользовательскую задачу в очередь. Эта конечная точка запрашивает для конкретного пользователя все невыполненные задачи, а затем возвращает коллекцию идентификаторов таких задач. В идеале может быть возвращена также информация о проверке, которая способна управлять пользовательским восприятием, интерактивно подтверждающим регистрацию.
4. Как только ошибки на стороне клиента устранены, обновленный объект сохраняется в базе данных, а подлежащая выполнению задача помечается как завершенная. На этой стадии процесс переходит к состоянию отправки подтверждения по электронной почте — `send-confirmation-email`, приводящему к отправке сообщения, включающего ссылку, на которой пользователь должен щелкнуть для подтверждения регистрации.
5. Затем на завершающем этапе в дело вступает конечная точка REST, которая выполняет запрос по каким-либо невыполненным задачам, связанным с подтверждениями по электронной почте относительно заданного клиента.

Описанный процесс начинается с потенциально недопустимого объекта, клиента, в правильности состояния которого нужно убедиться, прежде чем продолжить выполнение. Этот процесс смоделирован для поддержки последовательных действий над объектом с возвращением к соответствующему этапу до тех пор, пока будет сохраняться состояние недопустимости объекта.

Завершим данный тур изучением bean-компонентов, обеспечивающих деятельность двух сервисов, работающих с элементами задачи `Task`: `check-form` и `send-confirmation-email`.

Оба bean-компонента (и `CheckForm`, и `SendConfirmationEmail`) не представляют никакого интереса, поскольку являются обычными bean-компонентами Spring. `CheckForm` реализует простую проверку, позволяющую убедиться в заполнении полей имени и фамилии, после чего использует REST API проверки электронной почты Mashape (представленный в ходе рассмотрения Spring Batch), который позволяет убедиться в приемлемости формы электронного адреса. Bean-компонент `CheckForm` проверяет состояние заданной записи `Customer`, а затем предоставляет переменную процесса выполняемому компоненту `ProcessInstance`. Затем эта булева переменная по имени `formOK` используется в момент принятия решения о том, продолжать ли процесс либо принудить пользователя повторить попытку (пример 11.18).

Пример 11.18. Bean-компонент CheckForm, выполняющий проверку состояния клиента

```
package com.example;

import com.example.email.EmailValidationService;
import org.activiti.engine.RuntimeService;
import org.activiti.engine.impl.pvm.delegate.ActivityExecution;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Collections;
import java.util.Map;

import static org.apache.commons.lang3.StringUtils.isEmpty;

@Service
class CheckForm {

 private final RuntimeService runtimeService;  ❶

 private final CustomerRepository customerRepository;

 private final EmailValidationService emailValidationService;

 @Autowired
 public CheckForm(EmailValidationService emailValidationService,
  RuntimeService runtimeService, CustomerRepository customerRepository) {
  this.runtimeService = runtimeService;
  this.customerRepository = customerRepository;
  this.emailValidationService = emailValidationService;
 }

 ❷
 public void execute(ActivityExecution e) throws Exception {
  Long customerId = Long.parseLong(e.getVariable("customerId", String.class));
  Map<String, Object> vars = Collections.singletonMap("formOK",
```

```
    validated(this.customerRepository.findOne(customerId)));
   this.runtimeService.setVariables(e.getId(), vars);  ❸
  }

  private boolean validated(Customer customer) {
   return !isEmpty(customer.getFirstName()) && !isEmpty(customer.getLastName())
    && this.emailValidationService.isEmailValid(customer.getEmail());
  }

}
```

❶ Внедрение Activiti-сервиса `RuntimeService`.

❷ Элемент `ActivityExecution` является объектом контекста. Его можно использовать для обращения к переменным процесса, к самому механизму процесса и к другим интересующим вас сервисам.

❸ Затем его следует применить для формулировки результата теста.

Такая же основная форма используется и в компоненте отправки по электронной почте подтверждающего сообщения `SendConfirmationEmail`.

До сих пор рабочий поток рассматривался в контексте одного узла. Мы игнорировали один из наиболее обещающих аспектов облака — масштаб! Моделирование потока помогает идентифицировать возможности массового распараллеливания обработки. Каждое состояние в процессе ожидает наличия входных данных (в форме переменных процесса и предварительных условий (preconditions)) и предоставляет выходные данные (в форме побочных эффектов и переменных процесса). Состоянием процесса управляет рабочий поток.

Наша бизнес-логика выполняется в зависимости от состояния и в нужное время. Таким образом, все сделанное произошло либо в текущем узле, либо в том, который клиент использует для взаимодействия с системой при выполнении перечня задач, возложенных на пользователя. Если автоматическая обработка занимает весьма существенное время, то безопаснее будет перенести эту работу на любой другой узел! Ведь, в конце концов, мы задействуем облако. Такая возможность имеется, и все сводится к ее применению.

Элемент `serviceTask` в процессе BPMN находится в состоянии ожидания: механизм здесь станет на паузу (и распорядится любыми удерживаемыми ресурсами), пока не получит *явный сигнал*. Таковой может послать все, что находится извне: это может быть прибытие сообщения от поставщика Apache Kafka, получение сообщения по электронной почте, щелчок на кнопке или завершение долговременного процесса. Этим можно воспользоваться для перемещения потенциально долговременного процесса на другой узел, а затем по завершении послать сигнал о том, что процесс должен быть возобновлен. Механизм рабочего потока отслеживает состояние всех управляемых и запущенных процессов, позволяя отправлять запрос любому зависшему процессу и выполнять надлежащие действия. Рабочий поток

дает возможность естественно разделять задействованные процессы, раскладывать их на изолированные этапы и распределять по кластеру, а также предоставляет инструменты для восстановления на случай сбоев в распределении работы.

Рассмотрим пример 11.19, касающийся способов передачи выполнения рабочим узлам в определениях `MessageChannel` в Spring Cloud Stream. По аналогии с тем, что делалось при удаленном разделении задания в Spring Batch, решение будет рассматриваться в понятиях управляющего узла и рабочих узлов. *Лидирующим* будет обычный узел, порождающий обработку. Для инициирования последней у конечного узла есть конечная точка HTTP (http://localhost:8080/start). Процесс заведомо зауряден, так что можно сосредоточиться на распределении.

Пример 11.19. Асинхронный бизнес-процесс async.bpmn20.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns:activiti="http://activiti.org/bpmn"
             id="definitions"
             xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
             typeLanguage="http://www.w3.org/2001/XMLSchema"
             expressionLanguage="http://www.w3.org/1999/XPath"
             targetNamespace="http://www.activiti.org/bpmn2.0">

    <process id="asyncProcess">

        <startEvent id="start"/>

        <sequenceFlow id="f1" sourceRef="start" targetRef="spring-gateway"/>

        ❶
        <serviceTask id="spring-gateway"
          activiti:delegateExpression="#{gateway}"/>

        <sequenceFlow id="f2" sourceRef="spring-gateway"
          targetRef="confirm-movement"/>

        ❷
        <scriptTask id="confirm-movement" scriptFormat="groovy">
            <script>
                println 'Moving on..'
            </script>
        </scriptTask>

        <sequenceFlow id="f3" sourceRef="confirm-movement" targetRef="end"/>

        <endEvent id="end"/>

    </process>

</definitions>
```

❶ В самом начале процесс вводит состояние `serviceTask`. Здесь для обработки этого состояния используется Activiti-делегат в виде bean-компонента Spring `gateway`. Процесс в данном обработчике будет остановлен до получения сигнала.

❷ После получения сигнала процесс продолжит выполнять задачу `scriptTask` и мы увидим фразу `Moving on!`, записанную в консоли.

Bean-компонент `gateway` является ключевым. Это реализация Activiti-поведения при получении задачи — `ReceiveTaskActivityBehavior`, при котором сообщение записывается в канал `leaderRequests MessageChannel`, приводящийся в действие компонентом Spring Cloud Stream. Информационное наполнение сообщения — идентификатор выполнения `executionId`, который потребуется для последующей отправки сигнала о запросе. Нужно позаботиться о его сохранении либо в заголовках сообщения, либо в его информационном наполнении (как это делается в данном примере). При получении ответного сообщения по другому каналу `MessageChannel`, обеспечиваемому компонентом Spring Cloud Stream, ответный поток `Flow` вызывает метод `RuntimeService#signal(executionId)`, который, в свою очередь, возобновляет выполнение процесса (пример 11.20).

Пример 11.20. В LeaderConfiguration определяется порядок отправки запросов и то, что происходит при возвращении ответа от рабочих узлов

```
package com.example;

import org.activiti.engine.ProcessEngine;
import org.activiti.engine.impl.bpmn.behavior.ReceiveTaskActivityBehavior;
import org.activiti.engine.impl.pvm.delegate.ActivityBehavior;
import org.activiti.engine.impl.pvm.delegate.ActivityExecution;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.messaging.Message;

@Configuration
@Profile(Profiles.LEADER)
class LeaderConfiguration {

 @Bean
 ActivityBehavior gateway(LeaderChannels channels) {
  return new ReceiveTaskActivityBehavior() {

   @Override
   public void execute(ActivityExecution execution) throws Exception {

    Message<?> executionMessage = MessageBuilder.withPayload(execution.getId())
     .build();
```

```
    channels.leaderRequests().send(executionMessage);
   }
  };
 }

 @Bean
 IntegrationFlow repliesFlow(LeaderChannels channels, ProcessEngine engine) {
  return IntegrationFlows.from(channels.leaderReplies())
   .handle(String.class, (executionId, map) -> {
    engine.getRuntimeService().signal(executionId);
    return null;
   }).get();
 }
}
```

Рабочий узел взаимодействует только с экземплярами `MessageChannel`: никаких сведений об Activiti или о механизме рабочего потока (кроме заголовка String `executionId`) здесь нет. Рабочие узлы вольны делать что угодно, возможно, они запустят задание Spring Batch `Job` или задачу Spring Cloud `Task`. Рабочий поток идеально подходит для любых долговременных процессов, поэтому все, что нужно сделать, можно выполнить здесь: транскодировать видео, создать документ, проанализировать изображение и т. д. (пример 11.21).

Пример 11.21. В WorkerConfiguration определяется порядок отправки запросов и то, что происходит при возвращении ответа от рабочих узлов

```
package com.example;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.support.GenericHandler;

@Configuration
@Profile(Profiles.WORKER)
class WorkerConfiguration {

 @Bean
 IntegrationFlow requestsFlow(WorkerChannels channels) {

  Log log = LogFactory.getLog(getClass());

  ❶
  return IntegrationFlows.from(channels.workerRequests())
   .handle((GenericHandler<String>) (executionId, headers) -> {
    ❷
    headers.entrySet().forEach(e -> log.info(e.getKey() + '=' + e.getValue()));
```

```
    log.info("sending executionId (" + executionId + ") to workerReplies.");
    return executionId;
   }).channel(channels.workerReplies()) ❸
   .get();
 }
}
```

❶ При поступлении нового сообщения...

❷ ...перебираются заголовки и совершаются другие действия, требуемые согласно бизнес-логике, при условии, что мы сохранили информационное наполнение в виде идентификатора выполнения `executionId`...

❸ ...который возвращается по каналу `workerReplies` лидирующему узлу.

Распределение с помощью рассылки сообщений

Хотя в этой главе мы много рассуждали о долгоиграющих процессах (Spring Batch, Spring Cloud Task и рабочий поток), все эти темы заслуживают обсуждения в книге об облачных технологиях именно благодаря их *коммуникационной составляющей*. Эти проверенные решения оказываются очень кстати, когда можно воспользоваться возможностями масштабирования, открывающимися в облаке.

Резюме

Мы лишь поверхностно рассмотрели возможности обработки данных. Естественно, каждая из представленных выше технологий сама по себе говорит о широком круге различных технических приемов. Например, вместе со Spring Cloud Data Flow можно использовать такие средства, как Apache Spark или Apache Hadoop. Кроме того, они неплохо сочетаются друг с другом. А с помощью среды Spring Cloud Stream, выступающей в роли фабрики сообщений, обработка заданий Spring Batch легко поддается масштабированию в пределах кластера.

12 Интеграция данных

Микросервисы оптимизированы под действия команд, совершенствующих независимые части системы. Каждая команда работает над собственным конечным продуктом или функцией независимо от остальных команд организации, имея отдельные кодовую базу, цикл выпусков и, возможно, технологии! Побочный эффект такой изоляции — распределенность сервисов. Границы последних явно выражены, а доступ к данным осуществляется через эти границы. Как следствие, подразумевается использование распределения процессов и сетевых разделов. В главе 9 мы рассматривали способы моделирования ограниченных контекстов и взаимодействия с такими популярными источниками данных, как MongoDB или Redis. Поддержка отдельно взятых сервисов, управляющих собственными данными, не вызывает вопросов, а как заставить узлы сервисов и узлы источников данных обмениваться информацией? Как они договорятся о состоянии?

В текущей главе мы рассмотрим несколько разных способов (как старых, так и новых) получения данных из различных микросервисов и их интеграции. Одним из ключевых вопросов, который мы постараемся решить, будет соблюдение целостности данных в условиях распределенных вычислений. Литературы по распределенными системам с подробным описанием их особенностей очень много. Есть ряд фундаментальных статей (например, Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System (https://people.eecs.berkeley.edu/~brewer/cs262b/update-conflicts.pdf)), подробно описывающих проблему согласованности в архитектуре распределенной базы данных. Есть также труд Эрика Брюэра (Eric Brewer) CAP Theorem (https://dl.acm.org/citation.cfm?id=343502), в котором утверждается, что любая распределенная система может иметь не более двух из трех востребованных свойств, таких как:

- согласованность (consistency, C), эквивалентная наличию единой актуальной копии данных;
- высокая доступность (high availability, A) этих данных (для обновления);
- допуск к сетевым разделам (tolerance to network partitions, P).

На практике теорема CAP при наличии сетевых разделов исключает лишь возможность идеальной доступности и согласованности. Но потребность в последних возникает довольно редко. Мы же уделим внимание схемам, позволяющим создавать системы, которые в конечном итоге достигают согласованного состояния, и рассмотрим способы интеграции схем управления отказами или компенсационные действия в случае отказов. Целью данной главы не является обозначение и рассмотрение всех возможных режимов отказов! Согласно классификации распределенных систем, представленной Кайлом Кингсбери (Kyle Kingsbury) в его фундаментальном труде (https://github.com/aphyr/distsys-class), существует множество довольно веских причин, по которым система может отказать!

Наборы данных, с которыми мы работаем, подстраиваются под особенности наших приложений и экономических показателей. Архитектура этих наборов получает информацию, начиная от рабочих нагрузок, ориентированных на традиционные рабочие дни и являющихся, по сути, автономными, пакетными и конечными, и заканчивая международными рабочими нагрузками, имеющими формат 24/7, характеризующимися постоянной доступностью и ориентированными на события и потоки. Эти различные типы рабочих нагрузок должны поддерживаться любыми технологиями, которые мы задействуем.

Распределенные транзакции

На первый взгляд может показаться, что помочь в обеспечении согласованности между сервисами способны распределенные транзакции, поддерживаемые в Java с помощью JTA API. JTA — реализация архитектуры X/Open XA. В двухфазной фиксированной транзакции имеется *координатор*, прикрепляющий ресурсы к транзакции. Затем каждому ресурсу предлагается подготовиться к фиксации. Ресурсы сообщают о своей готовности к продолжению, после чего от них в тот же момент в качестве итогового действия координатор требует фиксации транзакции. Если каждый ресурс отвечает, что он выполнил фиксацию, то транзакция проходит успешно. В противном случае от ресурсов требуют выполнения отката.

Распределенные транзакции плохо вписываются в облачный мир. Единственное место сбоя зачастую координатор, который требует широкого обмена данными между узлами, прежде чем провести эффективную обработку транзакции. Этот обмен данными может привести к бесполезной перегрузке сетей. Распределенные транзакции также требуют, чтобы все участвующие ресурсы были осведомлены о координаторе транзакций. А это вряд ли возможно в системе, которая полностью полагается на REST API, не осуществляющие обмен данными по протоколу X/Open. Иногда необходим и такой подход, но применять его мы не рекомендуем. Если нужно научиться забирать и перемещать существующие приложения на основе JTA, то обратите внимание на рассмотрение вопроса распределенных транзакций с помощью JTA, приведенное в разделе «Управление транзакциями с помощью JTA и XA» приложения.

Изоляция сбоев и постепенное снижение качественных характеристик

В конечном счете цель состоит в получении приемлемых результатов от возможно распределенных сервисов. Сервисы подвержены сбоям, но это компенсируется за счет многократного повторения запроса до тех пор, пока сервис не станет доступен. Иными словами, проявляйте настойчивость! Выполнить повторный запрос или даже любую нестабильную часть работы можно с помощью библиотеки Spring Retry. Она была получена из Spring Batch (вопросы, касающиеся Spring Batch и других технологий, разработанных для поддержки долговременной обработки, были рассмотрены в разделе «Пакетные рабочие нагрузки» главы 11), и теперь ее можно использовать независимо от этой технологии. Библиотека будет полезна при попытках настроить соединения промежуточного уровня, такие как источники данных или очереди сообщений. Подобная схема весьма пригодится при становлении инфраструктуры в возможно недетерминированном порядке, где, к примеру, база данных приступает к обслуживанию запросов после того, как был запущен зависящий от нее сервис. Spring Retry можно воспользоваться для вызова нижестоящих сервисов, которые могут быть доступны или недоступны.

Рассмотрим простой REST-клиент, вызывающий другой сервис. При успешном завершении вызова будет возвращен ответ сервиса, а в случае сбоя — выдано исключение, которое Spring Retry направит к методу-обработчику, имеющему аннотацию `@Recover`. Метод восстановления возвращает тот же самый ответ, что и восстанавливающийся метод. В нашем примере он возвратит строку `OHAI`. Он будет делать это за столько попыток, сколько вы ему разрешите сделать. По умолчанию он будет вызываться три раза. Он станет давать все увеличивающиеся задержки по времени, прежде чем исчерпает попытки и наконец потерпит неудачу (пример 12.1). Чтобы активировать Spring Retry, нужно в конфигурационном классе применить аннотацию `@EnableRetry`.

Пример 12.1. Spring Retry помогает предпринимать повторные попытки с возрастающим временем между ними для вызова «капризных» нижестоящих сервисов

```
package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.retry.annotation.Backoff;
import org.springframework.retry.annotation.Recover;
import org.springframework.retry.annotation.Retryable;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import java.util.Date;
```

```
import java.util.Map;

@Component
public class RetryableGreetingClient implements GreetingClient {

 private final RestTemplate restTemplate;

 private final String serviceUri;

 private Log log = LogFactory.getLog(getClass());

 @Autowired
 public RetryableGreetingClient(RestTemplate restTemplate,
  @Value("${greeting-service.uri}") String domain) {
  this.restTemplate = restTemplate;
  this.serviceUri = domain;
 }

 ❶
 @Retryable(include = Exception.class, maxAttempts = 4,
    backoff = @Backoff(multiplier = 5))
 @Override
 public String greet(String name) {
  long time = System.currentTimeMillis();

  Date now = new Date(time);

  this.log.info("attempting to call the greeting-service " + time + "/"
   + now.toString());

  ParameterizedTypeReference<Map<String, String>> ptr =
   new ParameterizedTypeReference<Map<String, String>>() {
   };

  return this.restTemplate
   .exchange(this.serviceUri + "/hi/" + name, HttpMethod.GET, null, ptr, name)
   .getBody().get("greeting");
  }

  ❷
  @Recover
  public String recoverForGreeting(Exception e) {
   return "OHAI";
  }

}
```

❶ Это метод, способный дать сбой, в случае которого он выдаст исключение. Если нужно повторить попытку и восстановиться при указанном типе сбоя, то можно указать конкретный тип исключения `Exception`.

❷ Можно указать сколько угодно восстанавливаемых методов, типизированных с помощью исключения `Exception`, из которого они будут восстанавливаться.

Spring Retry может оказаться именно тем средством, которое вам необходимо для быстрого и легкого восстановления, а также применения эвристических способов разумного отката. Но иногда сервисам нужно больше времени на восстановление, и поток запросов и повторных попыток может негативно повлиять на их способность к возвращению в диалоговый режим работы. Чтобы пропускать запросы в зависимости от их состояния на основе их успешности или возвращаться к методу восстановления, как это делалось с помощью Spring Retry, можно применить *автомат защиты* (circuit breaker)

Данный автомат выполняет часть таких же действий, что и Spring Retry, являясь компонентом, инициирующим в исключительном случае механизм отката. Но это не просто обработчик типа `try-catch`. В нем содержится конечный автомат. Если автомат защиты отслеживает достаточное количество последовательных сбоев, то перенаправит запросы непосредственно на маршрут отката, отказываясь от попыток вызова по тому маршруту, который дает сбои. Этот алгоритм отличается от действий среды Spring Retry, поскольку она не занимается упреждающим отклонением запросов.

На данный момент в Spring поддерживаются два автомата защиты. Кроме того, можно применить подобный автомат, предоставленный проектом Spring Retry, а также интеграцию Spring Cloud для автомата защиты Netflix Hystrix. Посмотрим, что собой представляет последний.

Когда следует задействовать Hystrix на основе Spring Retry? На данный момент поддержка возможности вести наблюдение решена в Hystrix более эффективно. Hystrix включает сопровождающую информационную панель, получающую в качестве входной информации поток событий, отправляемый сервером, который это средство может задействовать для визуализации потока запросов, проходящих через автомат защиты. Интеграция, имеющаяся в среде Spring для Hystrix, предоставляемая библиотекой Javanica, созданной сообществом разработчиков, может показаться немного недружелюбной. Это связано с тем, что интеграция требует от String назвать метод, который должен использоваться, в то время как подход, применяемый в Spring Retry, опирается на исключения и на обработчики типизированных исключений. В настоящее время Hystrix позволяет воспрепятствовать запросам с помощью отдельных пулов потоков, чего нет в Spring Retry. Изолированный от потоков автомат защиты Hystrix может намного легче уйти от сбойного вызова, если запрос к сервисной зависимости должен продлиться слишком долго. Для этого, конечно же, требуется намного больше ресурсов, поскольку у каждого клиента есть свой собственный пул потоков. Hystrix также поддерживает изоляцию на основе использования семафоров, эффективность которой более или менее сопоставима с эффективностью применения Spring Retry. Если вам подходит изоляция на основе семафоров, у вас ограниченные вычислительные ресурсы и нужен более понятный API (и такой,

который более активно поддерживается сообществом), то воспользуйтесь Spring Retry. В случае настоятельной необходимости в изоляции на основе применения потоков или в визуализации автоматов защиты обратитесь к Hystrix.

Рассмотрим клиент, вызывающий потенциально капризный сервис, код которого показан в примере 12.2.

Пример 12.2. Этот автомат защиты дает нашему нижестоящему сервису время на то, чтобы перевести дух

```
package demo;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import java.util.Date;
import java.util.Map;

@Component
public class CircuitBreakerGreetingClient implements GreetingClient {

 private final RestTemplate restTemplate;

 private final String serviceUri;

 private Log log = LogFactory.getLog(getClass());

 @Autowired
 public CircuitBreakerGreetingClient(RestTemplate restTemplate,
  @Value("${greeting-service.uri}") String uri) {
  this.restTemplate = restTemplate;
  this.serviceUri = uri;
 }

 ❶
 @HystrixCommand(fallbackMethod = "fallback")
 public String greet(String name) {
  long time = System.currentTimeMillis();
  Date now = new Date(time);
  this.log.info("attempting to call " + "the greeting-service " + time + "/"
   + now.toString());

  //@formatter:off
  ParameterizedTypeReference<Map<String, String>> ptr =
```

```
    new ParameterizedTypeReference<Map<String, String>>() {
    };
  //@formatter:on
  return this.restTemplate
    .exchange(this.serviceUri + "/hi/" + name, HttpMethod.GET, null, ptr, name)
    .getBody().get("greeting");
 }

  public String fallback(String name) {
    return "OHAI";
  }

}
```

❶ Это метод, склонный к сбоям, и в случае сбоя он выдаст исключение.

Автомат защиты может быть либо открыт, либо закрыт. Если он закрыт, то попытается активировать подходящий путь и в случае выдачи исключения вызовет откат. Если автомат открыт, то направит запрос напрямую методу восстановления. Но со временем снова закроется и попробует возобновить трафик. В случае повторного сбоя автомат вернется в открытое состояние. Он также выдает поток событий, отправляемый сервером, который можно отследить с помощью информационной панели Hystix и Spring Cloud Turbine. Чтобы получить дополнительную информацию, обратитесь к обсуждению наблюдаемых приложений в главе 13.

Если используется такая платформа, как Cloud Foundry, то нет смысла организовывать длительный перерыв в обращении к сбойному нижестоящему сервису. В Cloud Foundry имеется периодически срабатывающий механизм, который будет автоматически перезапускать сбойный сервис, пригодный даже для автоматического масштабирования сервиса в случае пиковых нагрузок. Средства повторных попыток Retry и автоматы защиты облегчают клиенту постепенное снижение качественных характеристик при неизбежном сбое сервиса. Выстраиванием такого снижения будут заниматься Netflix и другие высокоэффективные организации. В качестве примера можно привести сбой вызовов сервиса гипотетической поисковой машины. Если клиенту будет показана трассировка стека, то это негативно повлияет на впечатления пользователя от работы сервиса.

В приведенных выше примерах для нас был актуален метод восстановления; мы должны были думать о случаях отказа и при сбое справиться с компенсационной транзакцией. Это признак хорошего качества программного средства. Предположим, будут появляться ошибки, противодействующие возможности предварительного внедрения более надежных систем. Мы рассмотрели способы выстраивания возможностей быстрого восстановления в случае сбоя сервиса, для которого у нас нет иного незамедлительного выхода. Вещи такого рода хорошо срабатывают, если предусматривается срочное изменение ориентации.

Теперь посмотрим на архитектуры, применяющие рассылку сообщений в качестве способа обеспечить *конечное* сведение сервисов к согласованному состоянию,

даже при условии выхода одного из них на данный момент из строя. Архитектура, построенная на использовании рассылки сообщений, справляется с фактом возможных сбоев, выстраиваясь соответствующим образом, а автомат защиты старается среагировать и компенсировать последствия неизбежного сбоя в целом по разделам. Перефразируя Грегора Хохпе (Gregor Hohpe), соавтора официальной трактовки рассылки сообщений и интеграции приложений в публикации Enterprise Integration Patterns (Addison-Wesley), можно сказать следующее: рассылка сообщений — более честная архитектура. В ней признается, что разъединенные сервисы способны иногда выходить из строя. Основы поддержки рассылки сообщений в Spring-экосистеме мы рассмотрели в главе 10. Пожалуйста, освежите свои представления о них, поскольку в следующих разделах подразумевается использование рассылки сообщений.

Сага-шаблон

В меру распределенной системе, где работа охватывает сразу несколько узлов, неизбежным фактом является сбой некоторых запросов и получение непоследовательных результатов. Изначально сага-шаблон был спроектирован для обработки долговременных транзакций на *одном и том же узле*. Традиционно долговременные транзакции выступают в распределенной системе в качестве узкого места, поскольку ресурсы должны удерживаться на протяжении всей транзакции. Вследствие этого они становятся недоступны системе. Тем самым запирается пропускная способность всей системы.

В 1987 году Гектор Гарсия-Молина (Hector Garcia-Molina) предложил сага-шаблон. Под сагой понималась долговременная транзакция, которую можно записать в виде последовательности подчиненных транзакций. Это коллекция подтранзакций, которую в распределенной системе можно назвать *запросами*. Транзакции должны быть перемежаемыми. Они не могут зависеть друг от друга и должны поддаваться перестроению порядка следования. В каждой транзакции должна также определяться компенсационная транзакция, отменяющая эффект транзакции и возвращающая систему в семантически согласованное состояние. Эти компенсационные транзакции определяются разработчиком. Нужно так продумывать систему при ее проектировании, чтобы обеспечивалась постоянная семантическая согласованность ее состояния. В сага-шаблоне согласованность достигается за счет доступности; побочные эффекты от каждого отказа в доступе видны всей остальной системе до той поры, пока не будет завершена вся сага.

Сага основана на идее использования координатора ее выполнения — saga execution coordinator (SEC). Он ведет *журнал саги*. В нем регистрируется отслеживание текущих транзакций и долговременный ход выполнения записей. Но собственное состояние в ходе работы SEC не хранит. В случае сбоя на замену этому координатору саги приходит простое развертывание нового. SEC отслеживает, какие транзакции

в саге находятся в стадии выполнения и как далеко они продвинулись. При сбое транзакции в саге SEC должен запустить компенсационную транзакцию. Если дает сбой и она, то SEC попробует выполнить ее повторно (при необходимости он будет делать это снова и снова!). Это налагает на архитектуру ряд предусловий. Сага-транзакции должны быть разработаны под семантику *«максимум однажды»*, компенсационные сага-транзакции — под семантику *«хотя бы один раз»*. Последние должны быть идемпотентными и не оставлять побочных эффектов в случае многократного выполнения. Для конкретно этого шаблона существует множество неплохих ресурсов, но в качестве хорошего введения в сага-шаблон мы настоятельно рекомендуем сообщение Кайти Маккэфри (Caitie McCaffrey) на GOTO 2015 (https://www.youtube.com/watch?v=xDuwrtwYHu8). Создать работоспособный SEC в качестве надстройки над такими элементами систем рабочих потоков, которые, к примеру, предоставляются Activiti, не составляет особого труда. (Более подробно Activiti и рабочий поток рассмотрены в разделе «Интеграция с рабочим потоком, ориентированная на процесс» главы 11).

CQRS (Command Query Responsibility Segregation)

Каждый микросервис в системе создается с целью применения *ограниченного контекста*, то есть он внутренне согласован. Объекты в подобном контексте имеют сугубо однозначное толкование, задействуемое лишь в конкретном ограниченном контексте. Поскольку каждый ограниченный контекст имеет собственную БД, то существует множество различных способов использования общих данных. Сервисы управляют записями в собственную базу, но к данным в других системах этим сервисам может понадобиться доступ только для чтения.

Предположим, создается механизм электронной торговли. В нем могут быть различные сервисы: для организации выполнения заказов, управления интерактивной корзиной покупок и доставкой, механизма поиска товаров, сервисы по работе с данным клиентом, с каталогом товаров и т. д. Хотя все эти сервисы служат достижению конечных целей по доставке товаров потребителям, не все из них используют одну и ту же модель предметной области. Такая модель для сервиса непосредственной доставки может применять все различные подробности, касающиеся посылок и транспортных средств доставки, которые совершенно излишни для предметной области, касающейся сервиса корзины покупок.

В такой системе все обновления всей информации, касающиеся товаров, будут управляться каталогом товаров. Другие системы, например механизм поиска товаров, могут управлять поисковым индексом, по которому хранятся, скажем, комментарии рецензента товара и описания товара в кластере Elasticsearch. Они не будут управлять такой информацией, как цена, возможные места складирования конкретного товара и тому подобными данными.

Микросервис может содержать один или несколько *сводных показателей (агрегатов)*. Агрегат состоит из корневого объекта и его дочерних объектов, работа с которым ведется в атомарном режиме: заказ и его позиции, изучение вопросов обслуживания клиентов и последующие действия, пассажир и связанный с ним багаж, пациент и связанная с ним история болезни и т. д.

Как обеспечить другим сервисам видимость записи микросервиса (в режиме транзакции обновляющих агрегат), не прибегая к использованию общих баз данных и нарушений инкапсуляции, которой мы при переходе на применение микросервисов надеемся достичь в первую очередь? Должно ли к механизму записи предъявляться требование обновления всех других сервисов, которые тем или иным образом зависят от этих данных? (Не приведет ли это лишь к замедлению процесса записи?) Кроме того, как обеспечить поддержку оптимального режима чтения совместно используемых данных в различных сервисах? К тому же гипотетическая поисковая машина более качественно будет обслуживаться полнотекстовым механизмом поиска, а не системой управления базами данных, ведь *одного истинного представления* не существует!

CQRS (Command Query Responsibility Segregation — разделение ответственности на команды и запросы) — это шаблон, о котором мы впервые услышали от придумавшего его Грега Янга (Greg Young), предложившего следующее решение: отделить записи от чтений. Командные (command) сообщения управляют обновлениями агрегатов в сервисе. Если нужно задать системе вопросы, то отправляется запрос (query). Запрос поддерживается с помощью представлений (или моделей запросов, query models). Он возвращает результаты, но не имеет наблюдаемых побочных эффектов.

Когда команда обновляет и сохраняет модель предметной области, событие (сообщение) инициирует обновление собственных отображений всех наблюдаемых моделей запросов. Если в конечном счете модель запроса является суммой всех обработанных ею событий, то возникает возможность переделки модели представления путем перевыдачи событий с самого начала. В действительности при условии сохранения каждого события в хранилище событий можно воссоздать состояние всей системы! Это называется *порождением событий*. В момент применения CQRS в его использовании нет необходимости, но одно служит другому. Получить исходное хранилище событий нетрудно. Одни разработчики в качестве такого хранилища используют Apache Kafka, другие — базы данных SQL. Есть даже специально созданные хранилища событий, например Event Store Грега Янга (Greg Young) (https://geteventstore.com/) и Eventuate Криса Ричардсона (Chris Richardson) (http://eventuate.io/), которые можно рассматривать в качестве весьма перспективных кандидатов для использования.

Рассмотрим простой REST API для управления *жалобами*. Клиенты Irate хотят получить возможность подачи жалоб.

Записи фиксируются в сервисе команд (возможно, за счет общения клиента с REST API), который затем публикует их на шине команд. Команда может пред-

ставлять собой подачу новой жалобы, дополнительный комментарий к жалобе или же разъяснение, предоставленное к жалобе представителем службы поддержки клиентов. Для обработки команды инициируется специальный обработчик, который сначала должен проверить допустимость поступившей команды. Если все в порядке, то он опубликует одно или несколько событий предметной области.

Обработчики событий реагируют на новое событие предметной области многими полезными действиями. Первичный результат реакции на такое событие — изменение состояния агрегата. Одно событие предметной области способно также породить еще больше событий, которые можно отправить другим микросервисам. Именно поэтому многие разработчики микросервисов подключаются к CQRS, чтобы получить возможность публикации и подписки на события предметной области, порождаемые приложениями за пределами ограниченного контекста. Такой подход предоставляет механизм обеспечения ссылочной целостности данных предметной области. Сообщения другим микросервисам могут использоваться для обработки событий предметной области, которая позволяет обслуживать беспокоящие разработчиков связи по внешним ключам, ставящие записи предметной области в зависимость от других записей в распределенной системе. Затем сервис запросов реагирует на запросы для обрабатываемого состояния, возвращая их без изменений из хранилища данных моделей запросов.

Изначально модель предметной области (команд) и модель запросов существует в одном приложении, в одном и том же процессе. Когда создается сочетание CQRS с микросервисами, ситуация, мягко говоря, усложняется. Рассмотрим «простой» микросервис, реализующий CQRS.

Является ли система CQRS распределенным монолитом? Многие разработчики, думая о микросервисах в настоящее время, представляют себе независимый сервисный компонент. В большинстве случаев микросервис создается в виде приложения, нацеленного на качественное выполнение единственной задачи. Еще более важное значение имеет возможность обновления и развертывания микросервиса независимо от других сервисов. Обычная система CQRS содержит несколько компонентов, что вызывает недоумение: «Является ли она распределенным монолитом?» Такой монолит получается в случае поставки функции, требующей скоординированного одновременного развертывания нескольких отдельных компонентов. Микросервисы предназначены для предоставления полномочий небольшим независимым командам, которые позволяют выполнять непрерывное развертывание функций в качестве части более объемной экосистемы, составленной из других микросервисов. Поскольку отдельно взятые компоненты CQRS сохраняют возможность независимого развертывания, можно сказать, что каждый блок развертывания по-прежнему удовлетворяет минимуму требований для независимой поставки функций в производственный процесс. Одна функция микросервиса должна требовать как минимум одного развертываемого блока.

Мы могли бы проделать основную часть пути к архитектуре CQRS, воспользовавшись обычными компонентами Spring Cloud Stream и Spring Data, но нам придется самостоятельно заполнить некоторые пробелы, такие как порождение событий и маршрутизация команд. Вместо этого мы применим среду Axon (http://www.axonframework.org/), разработанную Аллардом Байзе (Allard Buijze) (https://twitter.com/allardbz) и его командой в компании Trifork. Axon включает множество подходящих нам полезных функций: прошло около семи лет (и на этом история не заканчивается!), а среда всегда отлично работала со Spring. В ней даже предоставляется удобное автоконфигурирование Spring Boot, которое предлагается в качестве рекомендуемого подхода. Развитие Axon шло также по пути включения все большего количества составляющих экосистемы Spring Cloud.

Проект Axon получил название от конечного элемента аксона (части нейрона), передающего электрические сигналы. У нейронов имеются очень сложные перекрестные связи. Конечный элемент аксон отвечает за передачу сигналов от одного нейрона к другому.

Рассмотрим пример Axon-приложения.

API жалоб

Создадим гипотетический сервис `complaint`, а также еще один микросервис, поддерживающий отчетную статистику для жалоб. Основной сервис будет поддерживать создание жалобы, ее закрытие и добавление к ней комментариев. Нам потребуется обеспечить невозможность добавления комментария к уже закрытой жалобе. Для проведения этих операций (добавления жалобы, ее закрытия и добавления комментариев) все записи обмениваются данными с помощью *команд*. Команда — это сообщение, то есть объект, имеющий параметры, которые наш обработчик команд будет использовать для выполнения своей работы. Командное сообщение не имеет практически никакого поведения — только данные.

В данном примере, как и по всей книге, будет плодотворно использоваться проект Lombok. Это среда, предназначенная для поддержки аннотаций, обрабатываемых в ходе компиляции. В ней определены несколько полезных аннотаций времени компиляции, такие как @Data (которая приводит к добавлению геттеров и сеттеров для полей, имеющихся в объекте, и предоставляет полезные методы toString и equals) и @AllArgsConstructor или @NoArgsConstructor, приводящая к созданию конструкторов для всех полей в классе и соответственно конструктора, не принимающего никаких параметров.

Точкой входа в API жалоб является REST API. HTTP-запросы обрабатываются соответствующими методами в среде Spring MVC, которая затем выполняет диспетчеризацию команд на их шину (пример 12.3).

Пример 12.3. API жалоб

```
package complaints;

import org.axonframework.commandhandling.gateway.CommandGateway;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.util.UriComponents;
import org.springframework.web.util.UriComponentsBuilder;

import java.net.URI;
import java.util.*;
import java.util.concurrent.CompletableFuture;

import static org.springframework.http.MediaType.APPLICATION_JSON_VALUE;

//@formatter:off
@RestController
@RequestMapping(value = "/complaints",
  consumes = APPLICATION_JSON_VALUE,
  produces = APPLICATION_JSON_VALUE)
class ComplaintsRestController {
//@formatter:on

 private final CommandGateway cg;

 @Autowired
 ComplaintsRestController(CommandGateway cg) {
  this.cg = cg;
 }

 ❶
 @PostMapping
 CompletableFuture<ResponseEntity<?>> createComplaint(
  @RequestBody Map<String, String> body) {

  String id = UUID.randomUUID().toString();
  FileComplaintCommand complaint = new FileComplaintCommand(id,
   body.get("company"), body.get("description"));

  return this.cg.send(complaint).thenApply(
   complaintId -> {
    URI uri = uri("/complaints/{id}",
     Collections.singletonMap("id", complaint.getId()));
    return ResponseEntity.created(uri).build();
   });
 }

 ❷
 @PostMapping("/{complaintId}/comments")
```

```
@ResponseStatus(HttpStatus.NOT_FOUND)
CompletableFuture<ResponseEntity<?>> addComment(
 @PathVariable String complaintId, @RequestBody Map<String, Object> body)
 {

 Long when = Long.class.cast(body.getOrDefault("when",
  System.currentTimeMillis()));

 AddCommentCommand command = new AddCommentCommand(complaintId, UUID
  .randomUUID().toString(), String.class.cast(body.get("comment")),
  String.class.cast(body.get("user")), new Date(when));

 return this.cg.send(command).thenApply(commentId -> {

  Map<String, String> parms = new HashMap<>();
  parms.put("complaintId", complaintId);
  parms.put("commentId", command.getCommentId());

  URI uri = uri("/complaints/{complaintId}/comments/{commentId}", parms);

  return ResponseEntity.created(uri).build();
 });
}

@DeleteMapping("/{complaintId}")
CompletableFuture<ResponseEntity<?>> closeComplaint(
 @PathVariable String complaintId) {
 CloseComplaintCommand csc = new CloseComplaintCommand(complaintId);
 return this.cg.send(csc).thenApply(none -> ResponseEntity.notFound().build());
}

private static URI uri(String uri, Map<String, String> template) {
 UriComponents uriComponents = UriComponentsBuilder.newInstance().path(uri)
  .build().expand(template);
 return uriComponents.toUri();
}
}
```

❶ Записи (в виде HTTP-запросов `POST`) проходят через конечную точку `/complaints`, которая, в свою очередь, для отправки команды компоненту шины `CommandBus` использует компонент шлюза `CommandGateway`. Результатом применения `#send` в отношении `CommandGateway` становится тип `CompletableFuture<T>`. Среда Spring MVC знает, как работать с `CompletableFuture<T>`, — она возвратит ответ клиенту только при материализации результата `CompletableFuture`. Тем временем поток, обрабатывающий HTTP-запрос, переместит работу в поток, выполняемый в фоновом режиме.

❷ Чтобы добавить комментарий в текущей жалобе, компонент `AddCommentCommand` должен нести в команде идентификатор родительской жалобы. Он используется в агрегате для привязки команды к правильному экземпляру агрегата.

Командный шлюз (command gateway) выполняет в асинхронном режиме на основе типа диспетчеризацию команд к их соответствующим обработчикам. По сути, он передает полномочия маршрутизатору команд, который принимает решение по использованию обработчиков. По умолчанию это происходит на одном и том же узле, но есть и альтернативные стратегии, предназначенные для балансировки нагрузки, связанной с подобными вызовами. В целях определения местоположения сервиса одна реализация поддерживает JGroups, а другая — абстракцию Spring Cloud `DiscoveryClient`.

Обработчик команд (command handler) в терминологии Axon — просто метод в bean-компоненте, имеющий аннотацию `@CommandHandler`. Обработчики команд могут определяться в любом bean-компоненте Spring. В нашем API поддерживаются три операции, следовательно, используются как минимум три команды: создание жалобы (`FileComplaintCommand`) (пример 12.4), добавление комментария (`AddCommentCommand`) (пример 12.5) и закрытие жалобы (`CloseComplaintCommand`) (пример 12.6).

Пример 12.4. Команда FileComplaintCommand сигнализирует о необходимости создания новой жалобы

```
package complaints;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.axonframework.commandhandling.TargetAggregateIdentifier;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class FileComplaintCommand {

 @TargetAggregateIdentifier
 private String id; (1)

 private String company, description; (2)
}
```

(1) Вскоре значение `@TargetAggregateIdentifier` будет пересмотрено...

(2) ...в противном случае это будет простой объект Java — POJO. Нам необходимо знать о жалобе и о компании, на которую мы хотим подать эту жалобу.

Пример 12.5. Команда AddCommentCommand, сигнализирующая о необходимости добавления комментария к существующей жалобе

```
package complaints;

import lombok.AllArgsConstructor;
import lombok.Data;
```

```
import lombok.NoArgsConstructor;
import org.axonframework.commandhandling.TargetAggregateIdentifier;

import java.util.Date;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class AddCommentCommand {

 @TargetAggregateIdentifier
 private String complaintId;  (1)

 private String commentId, comment, user;

 private Date when;
}
```

(1) Даже притом, что в конечном итоге в нашей модели запросов все это выльется во вступление одного или нескольких объектов в дочернее отношение, сохранится управляемость всем этим с точки зрения владения жалобой, идентификатор которой сюда включен.

Пример 12.6. Команда CloseComplaintCommand, сигнализирующая о том, что жалоба должна быть закрыта

```
package complaints;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.axonframework.commandhandling.TargetAggregateIdentifier;

(1)
@Data
@AllArgsConstructor
@NoArgsConstructor
public class CloseComplaintCommand {

 @TargetAggregateIdentifier
 private String complaintId;
}
```

(1) Единственное, что нам нужно знать, — в отношении какого объекта следует применить действие.

Задача обработчика команд заключается в проверке информационного наполнения объекта команды, исследовании допустимости команды и в дальнейшей манипуляции состоянием модели предметной области. Обработчик команд мог бы загрузить шаблон `JdbcTemplate` или хранилище Spring Data и выполнить запись там же, на месте. Но тогда ему понадобится опубликовать событие, чтобы заинтересованные наблюдатели смогли провести соответствующее обновление своих

моделей запросов. Здесь обработчик команд может выполнить диспетчеризацию событий на шине событий Axon ко всем заинтересованным отслеживателям. Любой компонент может также проявлять интерес к событиям, задав *обработчики событий*. Мы рекомендуем вместо выполнения записи в обработчике команды возложить на обработчики команд публикацию события, а затем выполнить записи в обработчике события. Таким образом мы настраиваемся на мысль об обновлении нашей собственной предметной области, так же как на мысль об обновлении стороннего API.

Может потребоваться некоторая согласованность между командами. Как можно будет узнать в обработчике команд при попытке создания дочернего комментария, что жалоба, на которую нацелена команда добавления комментария `AddCommentCommand`, все еще открыта, если мы не можем загрузить и исследовать родительскую жалобу? Нам нужно каким-то образом получить эту информацию. Обычный компонент Spring на это не способен, поскольку номинальный компонент данной среды должен быть синглтон-объектом, не имеющим состояния. Каждый вызов обработчика события в обычном Spring-компоненте направляется к синглтон-объекту без состояния, являющемуся bean-компонентом, у которого не должно быть никаких воспоминаний о предыдущих транзакциях, связанных с жалобой.

Введите агрегаты Axon. *Агрегат* — компонент, имеющий состояние, чьи особенности представляют собой результат изменений, внесенных после обработки событий предметной области. Для загрузки агрегатов в Axon применяется реализация Axon-хранилища. В одних хранилищах содержатся сами агрегаты (например, в JPA), а в других запрашиваются прошлые события для восстановления текущего состояния (это называется порождением событий). Разница в выборе, который можно сделать на основе требований системы. Наш агрегат относится к агрегатам с порождением событий: он будет воспроизводить события для воссоздания последнего состояния любого заданного агрегата. Рассмотрим пример, а затем посмотрим, что в нем происходит (пример 12.7).

Пример 12.7. В агрегате ComplaintAggregate содержатся как обработчики команд, так и обработчики источников событий

```
package complaints.command;

import complaints.*;
import org.axonframework.commandhandling.CommandHandler;
import org.axonframework.commandhandling.model.AggregateIdentifier;
import org.axonframework.eventsourcing.EventSourcingHandler;
import org.axonframework.spring.stereotype.Aggregate;
import org.springframework.util.Assert;

import static org.axonframework.commandhandling.model.AggregateLifecycle.
apply;

❶
@Aggregate
```

```
public class ComplaintAggregate {

 ❷
 @AggregateIdentifier
 private String complaintId;

 private boolean closed;

 public ComplaintAggregate() {
 }

 ❸
 @CommandHandler
 public ComplaintAggregate(FileComplaintCommand c) {
  Assert.hasLength(c.getCompany());
  Assert.hasLength(c.getDescription());
  apply(new ComplaintFiledEvent(c.getId(), c.getCompany(),
c.getDescription()));
 }

 ❹
 @CommandHandler
 public void resolveComplaint(CloseComplaintCommand ccc) {
  if (!this.closed) {
   apply(new ComplaintClosedEvent(this.complaintId));
  }
 }

 ❺
 @CommandHandler
 public void addComment(AddCommentCommand c) {
  Assert.hasLength(c.getComment());
  Assert.hasLength(c.getCommentId());
  Assert.hasLength(c.getComplaintId());
  Assert.hasLength(c.getUser());
  Assert.notNull(c.getWhen());
  Assert.isTrue(!this.closed);
  apply(new CommentAddedEvent(c.getComplaintId(), c.getCommentId(),
   c.getComment(), c.getUser(), c.getWhen()));
 }

 ❻
 @EventSourcingHandler
 protected void on(ComplaintFiledEvent cfe) {
  this.complaintId = cfe.getId();
  this.closed = false;
 }

 ❼
 @EventSourcingHandler
```

```
 protected void on(ComplaintClosedEvent cce) {
  this.closed = true;
 }

}
```

1. Стереотипная аннотация `@Aggregate` сигнализирует о том, что этот объект является bean-компонентом, чьим жизненным циклом управляет Spring.
2. Когда команда направляется к шлюзу `CommandGateway`, ее атрибут `@TargetAggregateIdentifier` привязывает эту команду к данному агрегату за счет значения ее поля, имеющего аннотацию `@AggregateIdentifier`.
3. Новый экземпляр агрегата будет создан, когда команда `FileComplaintCommand` пройдет диспетчеризацию. Если проверка пройдет успешно, то случится диспетчеризация события `ComplaintFiledEvent`. Оно записывается в хранилище событий, а затем направляется к любому обработчику событий и обработчику источника событий (подобному `on(ComplaintFiledEvent cce)`) в этом агрегате и в других компонентах.
4. Жалоба будет закрыта, когда команда `CloseComplaintCommand` пройдет диспетчеризацию. Если проверка завершится успешно, то произойдет диспетчеризация события `ComplaintClosedEvent`. Оно записывается в хранилище событий, а затем направляется к любому обработчику событий и обработчику источника событий (подобному `on(ComplaintClosedEvent cce)`) в этом агрегате и в других компонентах.
5. После диспетчеризации команды `AddCommentCommand` к жалобе будет добавлен новый комментарий. Если проверка пройдет успешно, то выполнится диспетчеризация события `CommentAddedEvent`.
6. Этот первый обработчик источника событий инициализирует агрегат, предоставляя ему идентификатор агрегата и (в качестве дополнительной меры) присваивая булевой переменной `closed` значение `false`.
7. Второй обработчик источника событий при получении события `ComplaintClosedEvent` устанавливает для булевой переменной `closed` значение `true`.

Таким образом, Axon поддерживает два типа отслеживателей событий. В случае обработки события вне агрегата при указании обработчика событий используется `@EventListener`. Если агрегат, кроме всего прочего, обрабатывает событие и оно способствует надежной идентификации агрегата, то при указании обработчика событий применяется `@EventSourcingHandler`.

Когда обработчик команд публикует событие, предназначенное для управления конкретным экземпляром агрегата, Axon требует способа маршрутизации такого события к нужному экземпляру агрегата. События могут предоставлять идентификатор (имеющий аннотацию `@TargetAggregateIdentifier`), который привязывает его к конкретному экземпляру агрегата. Если в агрегате имеется обработчик

команд, то Axon загрузит нужный экземпляр агрегата из хранилища и обработает команду в этом экземпляре агрегата.

Зачем использовать порождение событий? В нашей простой предметной области этот механизм обслуживает варианты применения сервиса клиентов, поскольку полезно сохранять историю всех изменений, касающихся жалобы. Нам нужно получить возможность воспроизводить событие, приводящее к конечному состоянию агрегата. Если предпочтение отдается хранилищу Axon, а не простой загрузке агрегата из базы данных, то воспользуйтесь для агрегата аннотацией @Entity и уберите аннотацию @AggregateIdentifier, заменив ее аннотацией JPA-идентификатора @Id. Следует также убрать все методы, имеющие аннотацию @EventSourcingHandler.

Здесь используется `ComplaintAggregate` наряду со своими обработчиками команд и событий. При публикации команды `FileComplaintCommand` этот факт служит для создания нового экземпляра данного агрегата. При публикации команда `AddCommentCommand` порождает событие, имеющее целевой идентификатор `@AggregateIdentifier`, который Axon задействует для маршрутизации обработки обратно к данному экземпляру агрегата. Axon будет восстанавливать агрегат, загружая события из хранилища событий и вызывая соответствующие обработчики источников событий, которые (при воспроизведении по порядку) приведут к получению агрегата, отображающего его самое последнее состояние.

Можно заметить, что события являются зеркальными отображениями команд, превращающими глагол в повелительном наклонении в причастие прошедшего времени: например, `FileComplaintCommand` приводит к `ComplaintFiledEvent`. Это типичная ситуация, но нет никаких причин, мешающих команде порождать сразу несколько событий, которые, в свою очередь, запустят больше команд. Обработчики источников событий в данном агрегате сохраняются в хранилище событий автоматически при вызове метода `AggregateLifecycle#apply` внутри обработчика команд. Когда Axon попробует реконструировать агрегат, он будет знать, что нужно вызвать те же обработчики источников событий в том же самом порядке для достижения самого последнего известного допустимого состояния агрегата.

Посмотрим на события. Как и команды, они представляют собой всего лишь посылки с данными с весьма небольшим внутренним состоянием (примеры 12.8–12.10).

Пример 12.8. Диспетчеризация события ComplaintFiledEvent выполняется при подаче жалобы

```
package complaints;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
```

```
public class ComplaintFiledEvent {

 private final String id;

 private final String company;

 private final String complaint;
}
```

Пример 12.9. Диспетчеризация события CommentAddedEvent выполняется при добавлении комментария к родительской жалобе

```
package complaints;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class ComplaintClosedEvent {

 private String complaintId;
}
```

Пример 12.10. Диспетчеризация события ComplaintClosedEvent выполняется при закрытии жалобы

```
package complaints;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class ComplaintClosedEvent {

 private String complaintId;

}
```

События привязывают обновления к модели команд, чтобы обновить модель или модели запросов. Компоненты в модели запросов (или модели представления) отслеживают события и обновляют соответствующим образом модель запросов. В данном случае модель запросов также управляется хранилищем Spring Data JPA, при этом она может обрабатываться с помощью MongoDB, HBase, Neo4J, файла в файловой системе или чего-нибудь еще. Модель представления состоит из двух JPA-объектов: `ComplaintQueryObject` и `CommentQueryObject`. Последний владеет экземплярами `CommentQueryObject`.

Обработчик `ComplaintEventProcessor` преобразует события в операции над моделью запросов, в случае надобности обновляя или сохраняя JPA-объекты (пример 12.11).

Пример 12.11. Обработчик ComplaintEventProcessor

```
package complaints.query;

import complaints.CommentAddedEvent;
import complaints.ComplaintClosedEvent;
import complaints.ComplaintFiledEvent;
import org.axonframework.eventhandling.EventHandler;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.Collections;

@Component
public class ComplaintEventProcessor {

 private final ComplaintQueryObjectRepository complaints;

 private final CommentQueryObjectRepository comments;

 ❶
 @Autowired
 ComplaintEventProcessor(ComplaintQueryObjectRepository complaints,
  CommentQueryObjectRepository comments) {
  this.complaints = complaints;
  this.comments = comments;
 }

 @EventHandler
 public void on(ComplaintFiledEvent cfe) {
  ComplaintQueryObject complaint = new ComplaintQueryObject(cfe.getId(),
   cfe.getComplaint(), cfe.getCompany(), Collections.emptySet(), false);
  this.complaints.save(complaint);
 }

 @EventHandler
 public void on(CommentAddedEvent cae) {
  ComplaintQueryObject complaint = this.complaints
   .findOne(cae.getComplaintId());

  CommentQueryObject comment = new CommentQueryObject(complaint,
   cae.getCommentId(), cae.getComment(), cae.getUser(), cae.getWhen());
  this.comments.save(comment);
 }

 @EventHandler
 public void on(ComplaintClosedEvent cce) {
  ComplaintQueryObject complaintQueryObject = this.complaints.findOne(cce
   .getComplaintId());
  complaintQueryObject.setClosed(true);
  this.complaints.save(complaintQueryObject);
 }
}
```

❶ В этом обработчике событий используются две реализации хранилищ на основе Spring Data JPA.

Мы уже позаботились о записи обновлений, по сути, асинхронным способом, независимым от обработки команд. Обработчики событий обновляют модель предметной области жалобы в асинхронном режиме и независимо от команд, выполняющих диспетчеризацию событий. Мы также можем опубликовать эти события для других узлов системы, используя рассылку сообщений.

Особенности Axon заключаются во встроенном распространении событий по AMQP — протоколу, поддерживаемому такими брокерами, как RabbitMQ. Нам нужно сконфигурировать свое приложение, настроив его на обмен данными с экземпляром RabbitMQ. Добавим команду запуска Spring Boot RabbitMQ, `spring-boot-starter-amqp`, и сконфигурируем фабрику подключений RabbitMQ `ConnectionFactory` (здесь можно положиться на исходные установки или указать свойства в вашей среде `Environment`), а Axon станет транслировать события соответствующим образом. Посмотрим на конфигурацию, требуемую для публикации событий из нашего API жалоб. Сначала, как показано в примере 12.12, нужно настроиться на имя пункта назначения, которым мы хотим воспользоваться в свойствах нашего приложения `application.properties`.

Пример 12.12. Свойства application.properties

```
axon.amqp.exchange=complaints
logging.level.org.axonframework=DEBUG
```

Это, конечно, относится к обмену `complaints` (то есть жалобами) в RabbitMQ, которые могут быть или не быть, поэтому им нужно сконфигурировать также компоненты AMQP. В AMQP создание и администрирование компонентов на стороне брокера выполняется подобно части протокола по созданию очередей и обмену, поэтому все клиенты RabbitMQ могут провести настройку, как сделано в примере 12.13.

Пример 12.13. Конфигурирование AmqpConfiguration

```
package complaints;

import org.springframework.amqp.core.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
class AmqpConfiguration {

 private static final String COMPLAINTS = "complaints";

 ❶
 @Bean
 Exchange exchange() {
```

```
  return ExchangeBuilder.fanoutExchange(COMPLAINTS).build();
 }

 ❷
 @Bean
 Queue queue() {
  return QueueBuilder.durable(COMPLAINTS).build();
 }

 ❸
 @Bean
 Binding binding() {
  return BindingBuilder.bind(queue()) //
   .to(exchange()).with("*").noargs();
 }

 ❹
 @Autowired
 public void configure(AmqpAdmin admin) {
  admin.declareExchange(exchange());
  admin.declareQueue(queue());
  admin.declareBinding(binding());
 }
}
```

❶ Сообщения поступают на обмен...

❷ ...а затем ставятся в очередь.

❸ Эти два действия связываются вместе с использованием привязки...

❹ ...и нужно позаботиться о том, чтобы в RabbitMQ они были объявлены соответствующим образом.

API статистики жалоб

Имея такую инфраструктуру, нетрудно будет подключить к этим событиям другие узлы, чтобы они могли обновить соответствующие модели запросов. Модуль `demo-complaints-stats` представляет собой REST API, реагирующий на событие `ComplaintFiledEvent` с целью обновления полученных жалоб в имеющемся окружении. Нужно указать Axon на потребление событий из RabbitMQ и их направление к соответствующим обработчикам событий. В Axon представлена концепция, называемая *процессором событий*, поскольку события могут поступать из разных источников. В нашей конфигурации указывается, что этот процессор, основанный на применении AMQP, называется `complaints` (пример 12.14).

Пример 12.14. application.properties

```
axon.eventhandling.processors.statistics.source=statistics
server.port=8081
```

И наконец, чтобы обработать поступающие сообщения, нужно установить отслеживатель событий AMQP. Мы должны привязать поступающие сообщения к процессору на основе AMQP, воспользовавшись свойством в конфигурации, показанным в примере 12.15.

Пример 12.15. ComplaintsStatsApplication

```
package complaints;

import com.rabbitmq.client.Channel;
import org.axonframework.amqp.eventhandling.spring.SpringAMQPMessageSource;
import org.axonframework.serialization.Serializer;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class ComplaintsStatsApplication {


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

 ❶
 @Bean
 SpringAMQPMessageSource statistics(Serializer serializer) {
  return new SpringAMQPMessageSource(serializer) {

   @Override
   @RabbitListener(queues = "complaints")
   public void onMessage(Message message, Channel channel) throws Exception {
    super.onMessage(message, channel);
   }
  };
 }
}
```

❶ Настройка порядка использования и отправки поступающих событий.

Основной отслеживатель в этом простом REST API находится в самом API. В нем хранится (допускающее параллельную обработку!) отображение числа жалоб, отслеживаемых для каждой компании (пример 12.16).

Пример 12.16. Контроллер StatisticsRestController

```
package complaints;

import org.axonframework.config.ProcessingGroup;
```

```
import org.axonframework.eventhandling.EventHandler;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.atomic.AtomicLong;

❶
@ProcessingGroup("statistics")
@RestController
class StatisticsRestController {
 //@formatter:off
 private final ConcurrentMap<String, AtomicLong> statistics =
  new ConcurrentHashMap<>();
 //@formatter:on

 ❷
 @EventHandler
 public void on(ComplaintFiledEvent event) {
  statistics.computeIfAbsent(event.getCompany(), k -> new AtomicLong())
   .incrementAndGet();
  }

 ❸
 @GetMapping
 public ConcurrentMap<String, AtomicLong> showStatistics() {
  return statistics;
 }
}
```

❶ Нам нужны только сообщения о событиях из группы процессоров `statistics`.

❷ При поступлении события обновляется находящееся в памяти отображение Map...

❸ ...которое затем предоставляется в конечной точке REST. В результате по умолчанию получается HTTP-запрос `GET` по адресу http://localhost:8081/.

Таким образом, мы создали два сервиса, один из которых опирается на другой, не будучи с ним связанным. В эту смесь нетрудно ввести еще больше сервисов. Компоненты не обязаны знать или заботиться о том, как другие сервисы управляют своим состоянием.

Пока что мы лишь поверхностно коснулись возможностей, открываемых при использовании Axon и CQRS. Axon 3 является *существенным* пересмотром среды, благодаря которому она полностью охватывает Spring Boot. Axon уже обладает многогранной интеграцией с экосистемой Spring. Мы же в будущих выпусках надеемся увидеть интеграцию с экземплярами каналов сообщений Spring `MessageChannel` вместо конкретной привязки к RabbitMQ. Axon упрощает работу с CQRS, делая ее понятнее и последовательнее. Можно начать с модели команд и запросов, находящейся на том же самом узле, а затем за счет рассылки сообщений расширить ее на несколько узлов. Можно начать без поддержки порождения

событий и далее ввести такую поддержку, если того потребуют обстоятельства. Кроме того, в Axon предоставляется поддержка агрегатов, участвующих в сагах, в чем мы убедились, изучая раздел «Сага-шаблон» данной главы, но эту тему придется оставить до лучших времен.

Среда потока данных Spring Cloud Data Flow

В главе 10 было показано, как удачные абстракции — Spring Integration и Spring Cloud Stream — позволяют нам работать с другими системами в понятиях каналов сообщений `MessageChannel`. Компонент Spring Cloud Stream предоставляется в целях установки требуемого соглашения и внешнего конфигурирования подключений через поставщики сообщений. Что касается наших сервисов, то данные поступают из канала и выходят через него. Каналы — часть архитектуры конвейеров и фильтров, подключающих различные компоненты друг к другу. Для этих компонентов используется унифицированный «интерфейс»: `Message<T>`. Это очень похоже на работу в командной строке стандартных устройств ввода-вывода `stdin` и `stdout`: входные и выходные данные. Передача вполне понятна, поскольку все компоненты для производства или потребления данных согласны взаимодействовать с этой разновидностью информационного наполнения. Составить в среде оболочки UNIX произвольную композицию сложных решений, применяя утилиты командной строки, которые сконцентрированы на выполнении только одной конкретной задачи, и организуя канал данных, проходящий через `stdin` и `stdout`, совсем не сложно. То же самое можно сделать с нашими микросервисами на основе Spring Cloud. Среда Spring Cloud Stream повышает уровень абстрагирования, позволяя сфокусироваться на бизнес-логике и игнорировать подробности передачи данных от одного сервиса к другому. Появляется возможность создавать системы более высокого порядка и управлять сервисами рассылки сообщений с помощью потока данных Spring Cloud Data Flow.

Spring Cloud Data Flow позволяет удобным способом создавать многоступенчатые архитектуры, управляемые событиями (staged event-driven architectures, SEDA), которые являются идеальными шаблонами в облаке, где сервисы нуждаются в достаточной надежности, чтобы справиться с дополнительной нагрузкой и выполнить горизонтальное масштабирование.

Что такое SEDA? Это программная архитектура, разбивающая управляемые событиями приложения на набор ступеней, связанных очередями. Такой подход позволяет избегать больших издержек, связанных с моделями одновременных вычислений на основе потоков, и отсоединить диспетчеризацию событий и потоков от логики приложения. Сервисы могут сохранять доступность за счет регулирования входящей нагрузки на каждой очереди событий. Тем самым не допускается перегрузка ресурсов, когда запросы превышают возможности сервиса. Дополнительные сведения можно найти в главе 10 (рассмотрение вопроса рассылки сообщений).

В основу Spring Cloud Data Flow положены понятия потоков (streams) и задач (tasks).

Поток представляет собой логическую последовательность различных модулей на основе Spring Cloud Stream, то есть приложений. Spring Cloud Data Flow, по сути, развертывает и запускает различные приложения и переопределяет их исходные обязательные адресаты потока Spring Cloud Stream, благодаря чему данные перетекают из одного узла в другой в соответствии с нашим описанием. В мире потоков легко поддаются моделированию такие сложные сценарии обработки событий, как анализ состояния датчиков, анализ регистрационных записей, интернет-трейдинг и многие другие событийно-ориентированные сценарии.

А *задача* есть любой процесс, чье состояние выполнения нам нужно исследовать и от которого мы в конечном счете, если и не сразу, ожидаем завершения.

Собственно говоря, среда Spring Cloud Data Flow не занимается *развертыванием* ваших приложений, она возлагает эту работу на реализацию Spring Cloud Deployer. Там, где большинство аналитических платформ (таких как Apache Flink, или Apache Spark, или Apache Hadoop) определяют собственные среды выполнения кластеризации, команда Spring Cloud Data Flow постаралась обеспечить Spring Cloud Data Flow возможностью использования существующих и вполне эффективных сред выполнения. Проект Spring Cloud Deployer (https://github.com/spring-cloud/spring-cloud-deployer) определяет абстракцию для управления задачами или потоками локально либо же на средах выполнения и платформах, подобных Cloud Foundry (https://www.cloudfoundry.org/), Apache YARN, Mesos или Kubernetes. Существуют также внесенные и поддерживаемые сообществом реализации для Hashicorp Nomad (https://github.com/donovanmuller/spring-cloud-dataflow-server-nomad) и RedHat Openshift (https://github.com/donovanmuller/spring-cloud-dataflow-server-openshift). При этом предоставляются два ключевых интерфейса: `AppDeployer` и `TaskLauncher`. Первый связан с жизненным циклом развертывания, а второй — с жизненным циклом выполнения. `AppDeployer` поддерживает развертывание, его отмену и запрос состояния приложения (с потенциально неопределенным временем существования). `TaskLauncher` поддерживает запуск, отказ от выполнения, прекращение работы и очистку данных экземпляров развернутых таким образом приложений.

В этом примере рассматривается локальная реализация Spring Cloud Data Flow, требующая только скомпилированных .jar-компонентов, развернутых для работы в локальном хранилище Maven.

Чтобы найти реализацию Data Flow для вашей структуры распространения программного продукта, проверьте содержимое страницы проекта Spring Cloud Data Flow (http://cloud.spring.io/spring-cloud-dataflow/). Там вы найдете реализации для многочисленных технологий, включая *Local Server*, который будет использоваться

в данном примере. Для получения реального горизонтального масштабирования комплексные тесты для кода, рассматриваемого в этой главе, развертывают и используют реализацию Spring Cloud Data Flow Cloud Foundry Server.

Потоки

Потоки подразделяются на три типа компонентов: *источники* (где создаются сообщения), *процессоры* (где поступающее сообщение обрабатывается, а затем отправляется) и *приемники* (где поступающее сообщение потребляется). В Spring Cloud Data Flow каждый из этих компонентов во время выполнения проявляется как отдельный, запуская экземпляр приложения, приводимый в действие средой Spring Cloud Stream. Источник и приемник более или менее соответствуют понятию входящего и исходящего адаптера Spring Integration соответственно, а процессор в целом — преобразователю Spring Integration. Все потоки состоят из комбинаций этих трех компонентов. Spring Cloud Data Flow автоматически компонует и объединяет эти экземпляры приложений с привязками Spring Cloud Stream, подключая одно запущенное приложение к другому в отдельном процессе или вообще в ином узле.

Изначально сервер Spring Cloud Data Flow представляет собой чистый холст. Ему нужны зарегистрированные приложения — строительные блоки более сложных решений — для составления из них пользовательских определений потоков и задач. Прежде чем приступить к написанию и регистрированию пользовательских приложений, попробуйте применить несколько из многочисленных заранее созданных приложений, предоставляемых проектом Spring Cloud Stream App Starters (https://github.com/spring-cloud/spring-cloud-stream-app-starters). Приложения можно зарегистрировать вручную, по одному или все сразу, с помощью файла свойств приложения. Spring Cloud Data Flow запустит эти приложения в любой из реализаций Spring Cloud Deployer, которая соединит их, задействуя любой из редакторов связей Spring Cloud Stream. Это значит, что требуются несколько параметров конфигурации: должно ли приложение ссылаться на компонент Maven или на компонент хранилища Docker? Должна ли содержаться в описании приложения ссылка на приложение, скомпилированное в отношении редактора связей Spring Cloud Stream RabbitMQ или же привязки к Spring Cloud Stream Apache Kafka? К счастью, определения имеются на главной странице Spring Cloud Stream App Starters (http://cloud.spring.io/spring-cloud-stream-app-starters/).

Можно также составить каталог всех пользовательских приложений в вашей системе и зарегистрировать их с помощью сервера Spring Cloud Data Flow. Ниже приведен простой файл свойств приложения для нескольких модулей (пример 12.17). В каждой строке дается определение типа компонента (источник, приемник, процессор и т. д.), логическое имя приложения и URL, разрешение которого может быть выполнено с привлечением реализации `org.springframework.core.io.Resource` в пути к классам. Spring Cloud Data Flow предоставляет новые реализации, поддерживающие наряду с другими возможностями поиск хранилища Maven и реестра Docker.

Пример 12.17. Файл определения свойств пользовательского приложения

```
source.account-web=maven://org.cnj:account-web:0.0.1-SNAPSHOT
sink.account-worker=maven://org.cnj:account-worker:0.0.1-SNAPSHOT
source.order-web=maven://org.cnj:order-web:0.0.1-SNAPSHOT
sink.order-worker=maven://org.cnj:order-worker:0.0.1-SNAPSHOT
source.payment-web=maven://org.cnj:payment-web:0.0.1-SNAPSHOT
sink.payment-worker=maven://org.cnj:payment-worker:0.0.1-SNAPSHOT
source.warehouse-web=maven://org.cnj:warehouse-web:0.0.1-SNAPSHOT
sink.warehouse-worker=maven://org.cnj:warehouse-worker:0.0.1-SNAPSHOT
```

Если ни одно из них не соответствует вашим требованиям, то нетрудно будет создать пользовательское приложение.

Предположим, нужно подключить пользовательское приложение к определению потока. Задействуйте Spring Initializr (http://start.spring.io/), выберите тот редактор связей, который хотите применить (RabbitMQ или Apache Kafka), а затем заполните код обычным процессором, получающим входящее String-сообщение, добавляющим { перед и } после тела сообщения, после чего отправляющим сообщение дальше (пример 12.18).

Пример 12.18. Обычный процессор Spring Cloud Data Flow

```
package stream;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Processor;
import org.springframework.integration.annotation.MessageEndpoint;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.messaging.Message;

@MessageEndpoint
❶
@EnableBinding(Processor.class)
❷
@SpringBootApplication
public class ProcessorStreamExample {

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

 @ServiceActivator(inputChannel = Processor.INPUT, outputChannel =
 Processor.OUTPUT)
 public Message<String> process(Message<String> in) {
  return MessageBuilder.withPayload("{" + in.getPayload() + "}")  ❸
    .copyHeadersIfAbsent(in.getHeaders()).build();
 }
}
```

❶ Это компонент рассылки сообщений Spring Integration, который...

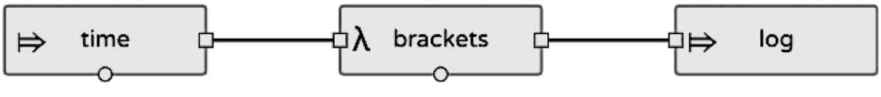
❷ ...зависит от исходной привязки Spring Cloud Stream `Processor`, предоставляющей общеизвестное `MessageChannel`-определение входа и выхода.

❸ Процессор заключает свое информационное наполнение в фигурные скобки.

Это процессорное приложение установлено в наше локальное хранилище Maven с идентификатором группы `cnj`, идентификатором компонента `stream-example` и версией `1.0.0-SNAPSHOT`. Если зарегистрировать данный процессор как приложение в файле определения приложения под именем, скажем, `brackets`, то его можно будет использовать в более крупных решениях. Оно становится еще одним строительным блоком. Сочетая пользовательские и заранее созданные приложения, можно собрать более сложные решения, используя DSL-язык потока Spring Cloud Data Flow. Предположим, нужно подключить выход источника `time` (создающего каждую секунду сообщение `Message`) к только что импортированному нами процессору `brackets`, а затем подключить выход процессора `brackets` к компоненту по имени `log`, регистрирующему свои входные данные. Такой поток легко реализуется с помощью DSL-языка потока Spring Cloud Data Flow, о чем свидетельствуют приведенные ниже пример 12.19 и рисунок.

Пример 12.19. Обычное определение потока

```
time | brackets | log
```



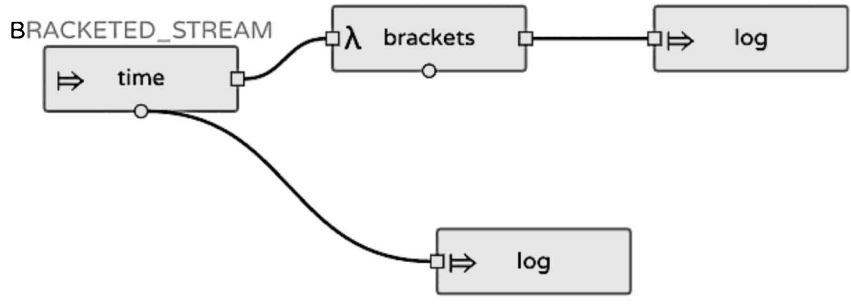
Это определение приведет к формированию трех различных процессов операционной системы, возможно, на различных узлах, в зависимости от реализации используемой среды Spring Cloud Deployer. Процессы связаны друг с другом в динамическом режиме с помощью редактора связей Cloud Stream (RabbitMQ, Apache Kafka и т. д.). Предположим, что нужно регистрировать продукт как первого потока, так и источника `time`. Для этого можно применить *отвод* в Spring Cloud Data Flow, чтобы *вдобавок* к любым уже установленным подключениям направить продукт одного компонента (источника или процессора) другому компоненту. Посмотрим, как это делается в примере 12.20 и на показанном ниже рисунке.

Пример 12.20. Отвод результатов источника time

```
BRACKETED_STREAM=time | brackets | log   ❶
:BRACKETED_STREAM.time > log   ❷
```

❶ Назначение определения имени переменной...

❷ ...затем получение по указателю значения компонента, чей выход нужно отвести применительно к имени переменной.



Потоки идеально подходят для потенциально бесконечного количества обработок. Поскольку компоненты подключаются через очереди, то они легко поддаются масштабированию вверх и вниз, используя любые механизмы, доступные в структуре распределения используемой вами реализации Spring Cloud Deployer. В Cloud Foundry это так же просто, как написать `cf scale -i ..` , где `..` заполняется логическим именем, присвоенным экземпляру приложения для компонента в определении потока Spring Cloud Data Flow. Очереди поглощают дополнительную нагрузку, гарантируя, что даже при отсутствии достаточной вычислительной мощности система в целом останется стабильной. Потоки — отличный способ описать многоступенчатые архитектуры, управляемые событиями (SEDA).

Задачи

Среда Spring Cloud Data Flow может также управлять жизненным циклом задач, выполняемых на основе применения Spring Batch. Для создания приложений Spring Cloud используйте Spring Initializr. Чтобы активировать Spring Batch, Spring Cloud Task и шлюз к Spring Cloud Task, добавьте к пути к классам проекта строки `org.springframework.cloud : spring-cloud-task-core`, `org.springframework.cloud : spring-cloud-task-batch` и `org.springframework.boot : spring-boot-starter-batch` (пример 12.21).

Пример 12.21. Простое задание Spring Batch Job, определенное в примере Spring Cloud Task

```
package task;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration.annotation.
  EnableBatchProcessing;
import org.springframework.batch.core.configuration.annotation.
  JobBuilderFactory;
import org.springframework.batch.core.configuration.annotation.
  StepBuilderFactory;
```

```
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.task.configuration.EnableTask;
import org.springframework.context.annotation.Bean;

@EnableTask
(1)
@EnableBatchProcessing
@SpringBootApplication
public class BatchTaskExample {

 private Log log = LogFactory.getLog(getClass());

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

 @Bean
 Step tasklet(StepBuilderFactory sbf, BatchTaskProperties btp) {
  return sbf.get("tasklet").tasklet((contribution, chunkContext) -> {
   log.info("input = " + btp.getInput());
   return RepeatStatus.FINISHED;
  }).build();
 }

 @Bean
 Job hello(JobBuilderFactory jbf) {
  Step step = this.tasklet(null, null);
  return jbf.get("batch-task").start(step).build();
 }
}
```

(1) Активация абстракции Spring Cloud Task.

Для дальнейшего использования установите это приложение задачи в свое локальное хранилище Maven с групповым идентификатором `cnj`, идентификатором компонента `task-example` и версией `1.0.0-SNAPSHOT`.

REST API

Основа Spring Cloud Data Flow — сервер, предоставляющий REST API для управления потоками и задачами. Сервер сохраняет свое состояние в реляционной базе данных. По умолчанию он использует встроенный экземпляр H2, но можно предоставить и определение собственного источника данных. Управление этим API осуществляется через шаблон Spring Cloud Data Flow `DataFlowTemplate`, веб-приложение, поставляемое вместе с сервером Data Flow, или интерактивную оболочку.

При возможности установки Spring Cloud Data Flow в качестве модуля Spring Boot рекомендуем для каждого типа Spring Cloud Deployer, раздаваемого и рассматриваемого на странице проекта (http://cloud.spring.io/spring-cloud-dataflow/), просто запустить уже готовые «под ключ» реализации `.jar`-приложения.

Все будет сводиться к загрузке текущей версии с последующим запуском `.jar`-приложения с помощью команды `java`.

При запуске версии Cloud Foundry получается тот же самый основной процесс, за исключением того, что дополнительно указывается ряд свойств (или переменных среды), которые помогают клиенту Foundry, используемому в аутентификации установщика и взаимодействующему с вашим адресатом Cloud Foundry.

Запустите сервер, и он, готовый к использованию, заработает на порте 9393.

Знакомство с клиентами Data Flow

При запущенном сервере у нас появятся следующие запросы:

- регистрация существующих задач и потоковых приложений (тех, что находятся в хранилище Maven или в хранилище Docker);
- определение потоков и задач;
- запуск потоков и задач;
- опрос статуса запущенных потоков и задач.

Следовательно, нам понадобится клиент Spring Cloud Data Flow! Какой из них использовать, полностью зависит от того, что вы пытаетесь получить. Если вы оператор или разработчик, то, скорее всего, вам сначала нужно будет применить инструментальную панель. Она предоставит пошаговый интерфейс, выдающий все полезные функции Spring Cloud Data Flow в качестве функций UI. Если есть желание узнать обо всем доступном, провести эксперимент, ускорив свою работу и наблюдая за реакцией интерфейса, начните с использования инструментальной панели. Операторы в конечном итоге предпочтут задействовать оболочку, предоставляющую описания высокоуровневых операций в виде сценариев. Если же вам как разработчику понадобится автоматизировать некоторые аспекты Spring Cloud Data Flow или интегрировать ряд операций этой среды в более крупное приложение, то вы сможете оценить `DataFlowTemplate`.

Инструментальная панель

Веб-приложение инструментальной панели устанавливается с адреса http://localhost:9393/dashboard. Она упрощает создание визуального описания сложных приложений Spring Cloud Data Flow и управление ими. Эта панель включает средство

визуального моделирования Spring Flo, которое поддерживает интерактивные определения компонуемых потоков.

Зарегистрировать приложения можно каждое по отдельности или же все разом на вкладке Apps (Приложения) (рис. 12.1).

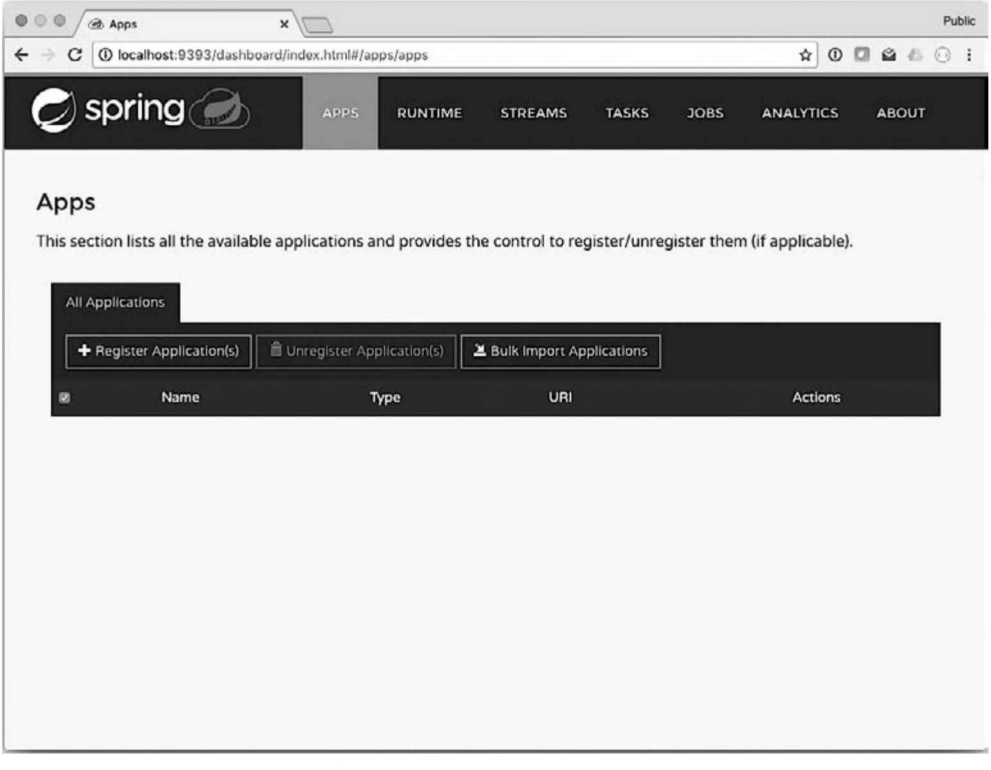


Рис. 12.1. Главный экран инструментальной панели Spring Cloud Data Flow, открытый на вкладке Apps (Приложения)

Для определения приложения среде Spring Cloud Data Flow можно указать любой допустимый и годный для разрешения в `org.springframework.core.io.Resource` URI-идентификатор. Ранее мы устанавливали в хранилище Maven процессор и приложение задачи. Для установки процессора присвойте ему имя, выберите в типе приложения пункт Processor и введите URI: `maven://cnj:stream-example:1.0.0-SNAPSHOT`. На рис. 12.2 показан снимок экрана пользовательского интерфейса после совершения данных действий.

Можно также зарегистрировать целый набор приложений: либо путем ссылки на файл определения приложений, либо путем переноса содержимого такого файла непосредственно в поле формы ввода. Снимок экрана пользовательского интерфейса после совершения данных действий показан на рис. 12.3.

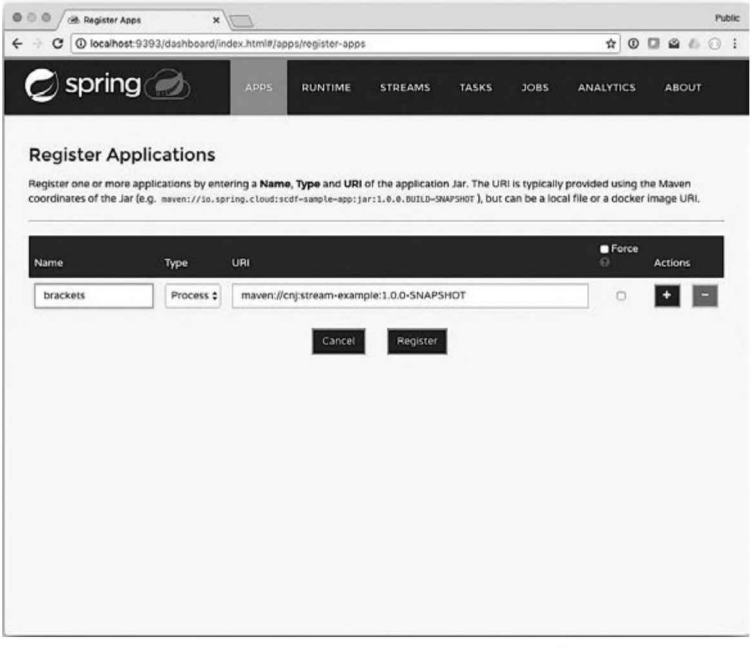


Рис. 12.2. Регистрация пользовательского процессора на инструментальной панели

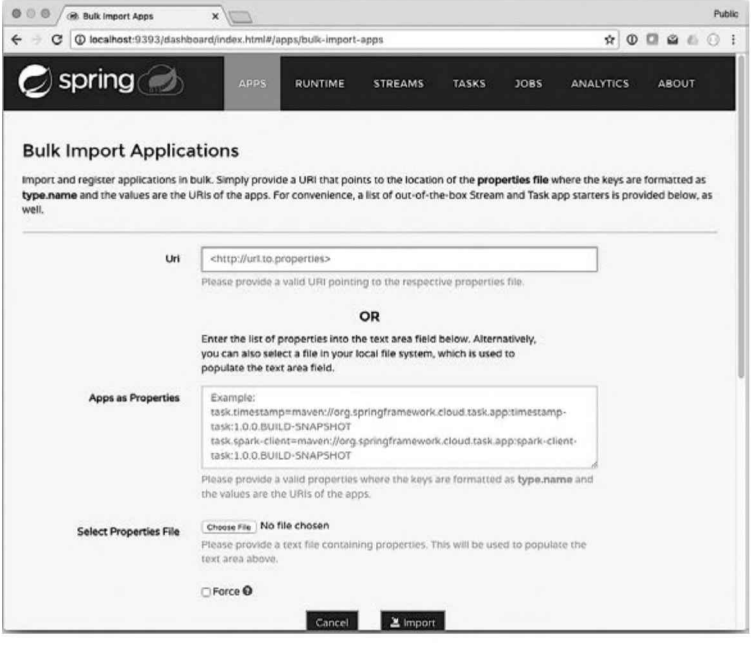


Рис. 12.3. Регистрация подборки определений приложений

Кроме того, можно зарегистрировать подборку некоторых весьма удобных заранее собранных в пакет и созданных приложений, предоставленных командой разработчиков Spring Cloud Data Flow с выбираемой вами привязкой и пакетированием. Копия экрана пользовательского интерфейса на данном этапе работы показана на рис. 12.4.

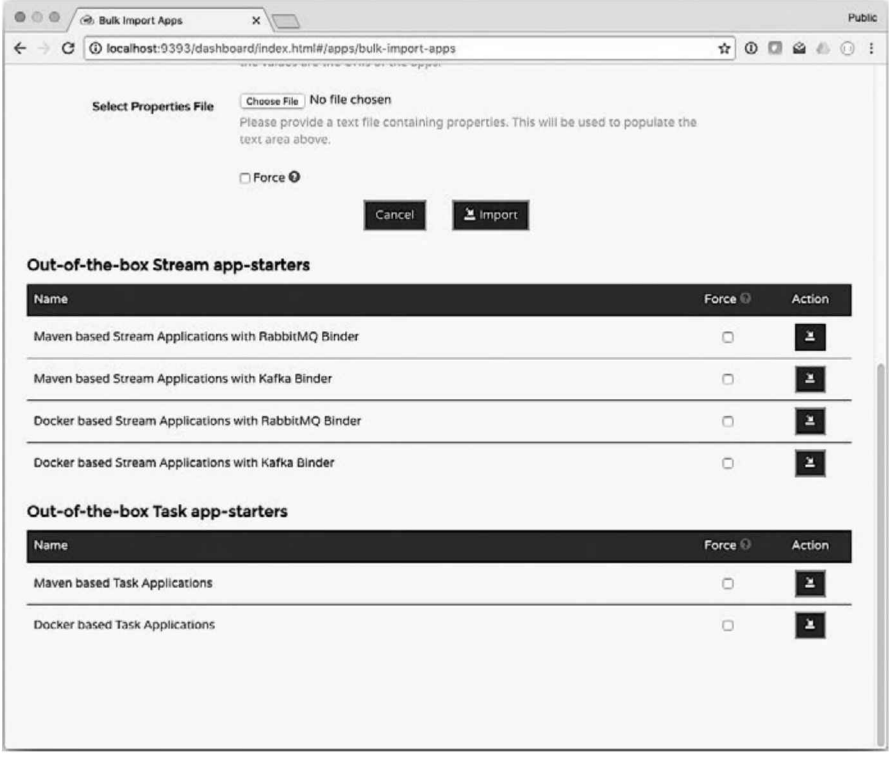


Рис. 12.4. Применение готовых пусковых механизмов Stream-приложений

Теперь, задействуя зарегистрированные приложения, можно выполнить определения пользовательского потока. Копия экрана пользовательского интерфейса на данном этапе работы показана на рис. 12.5.

Самым простой (и, на наш взгляд, наиболее предпочтительный) способ конструирования нового потока — использование средства проектирования Spring Flovisual. Оно одновременно отображает визуальную модель (которой можно манипулировать с помощью мыши) и текст в форме. Данное средство действительно удобно в работе, позволяет перетаскивать элементы из расположенной слева подборки (источник, приемник или процессор) и подключать компоненты с использованием их *портов* (небольших квадратных значков справа или слева от прямоугольных форм). Копия экрана пользовательского интерфейса на данном этапе работы показана на рис. 12.6.

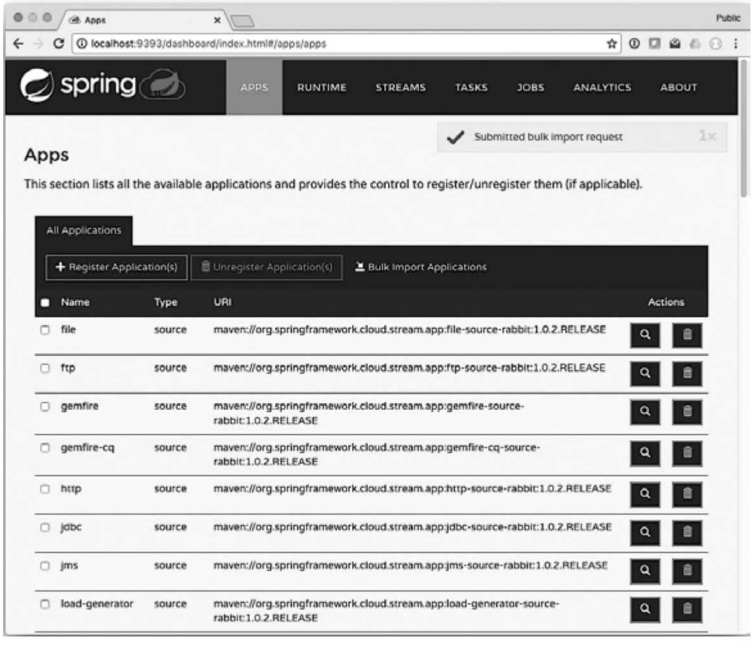


Рис. 12.5. Вкладка Apps (Приложения)

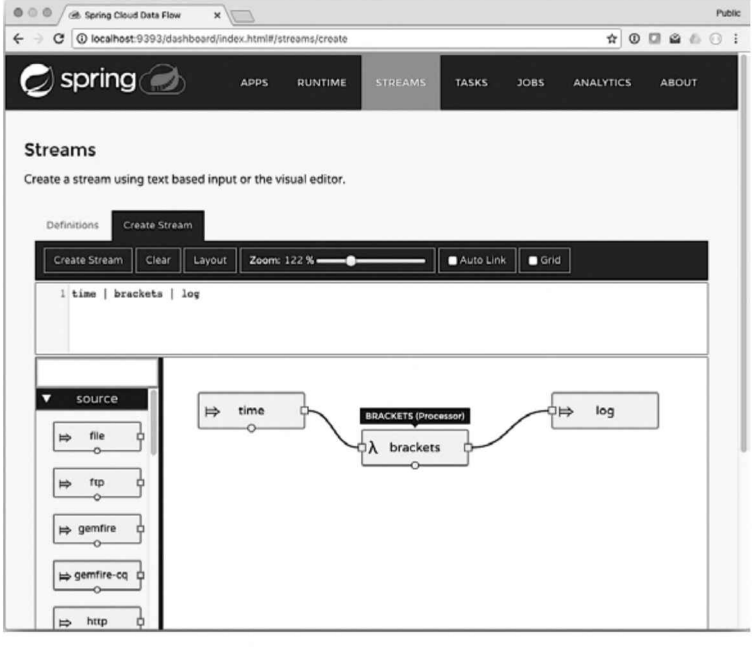


Рис. 12.6. Вкладка Streams (Потоки), конструирование потока

Отсюда очень просто развернуть определение потока. Ему нужно дать логическое имя. Копия экрана пользовательского интерфейса на данном этапе работы показана на рис. 12.7.

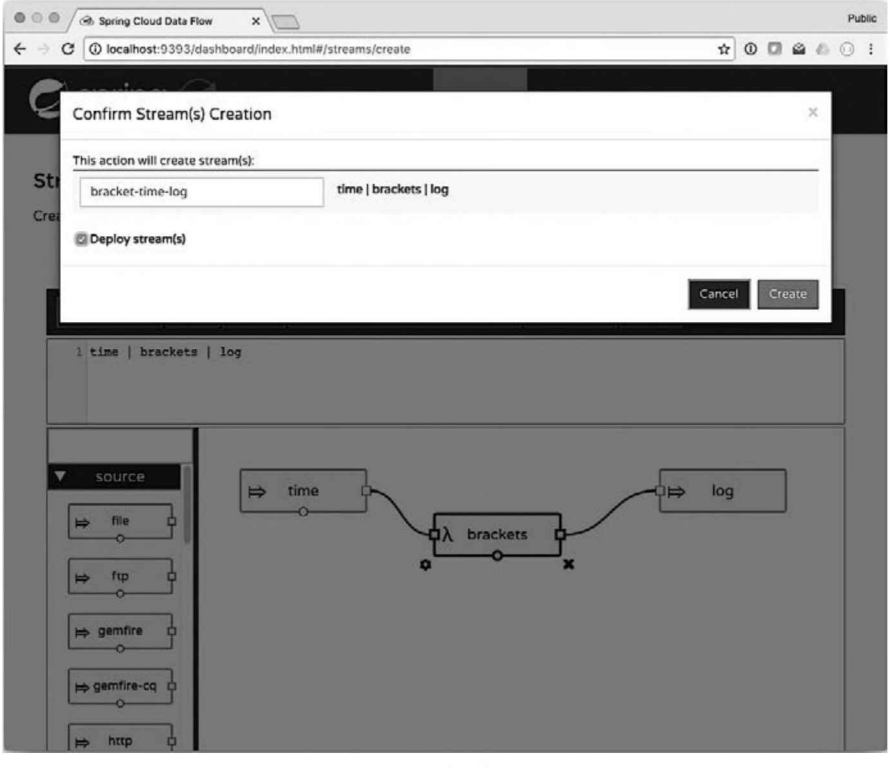


Рис. 12.7. Присвоение имени

Развернутые потоки можно увидеть на вкладке Streams (Потоки). Чтобы раскрыть визуальное представление, щелкните кнопкой мыши на стрелке. Кроме того, вы получите представление об успешности развертывания потока. Если используется локальное средство развертывания, то для получения информации о том, где можно найти записи для развернутых приложений, проверьте регистрационные записи сервера Spring Cloud Data Flow. Снимок экрана пользовательского интерфейса на данном этапе работы показан на рис. 12.8.

Тот же порядок работы применим и для зарегистрированных задач, за исключением того, что задачу не нужно развертывать. Она определяется один раз (на основе своего зарегистрированного приложения), и запускается ее один или несколько экземпляров. Щелкните на значке Create Definition (Создать определение) возле названия соответствующей задачи. Снимок экрана пользовательского интерфейса на данном этапе работы показан на рис. 12.9.

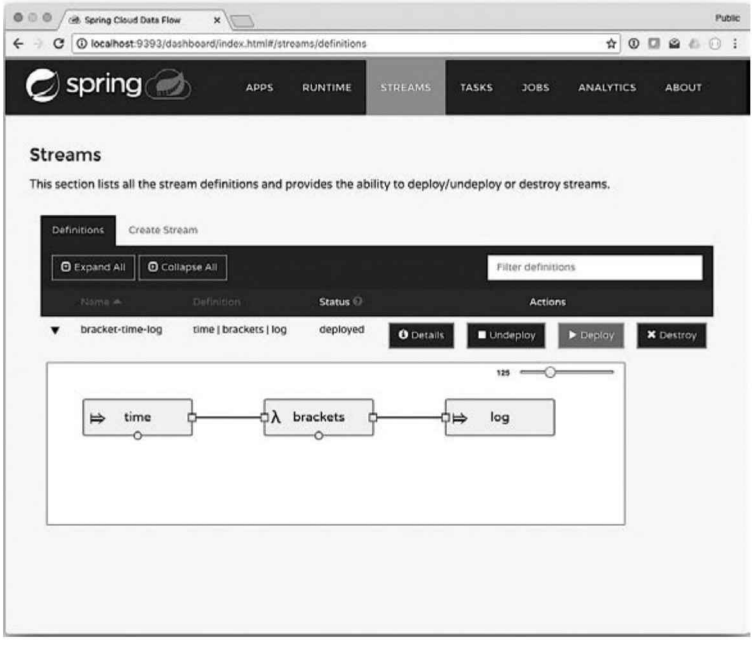


Рис. 12.8. Определения потока

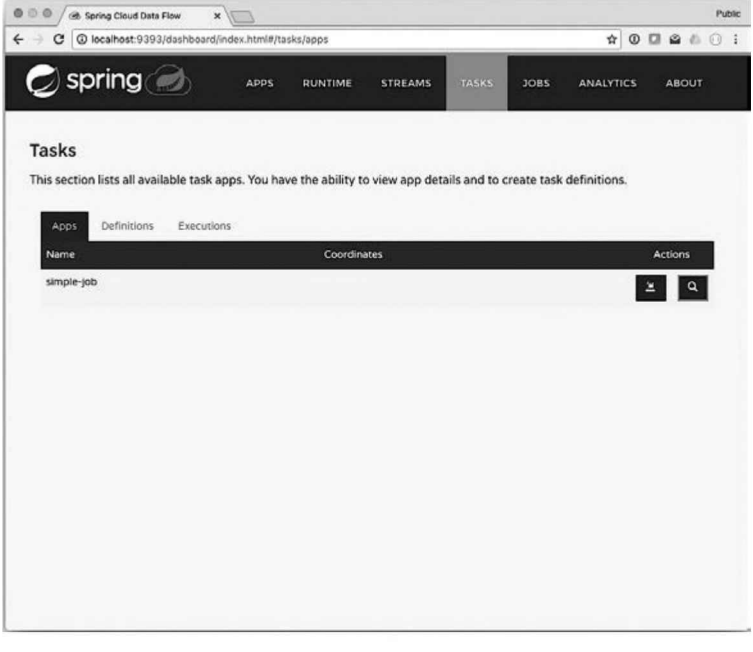


Рис. 12.9. Задача simple-job

Затем определите то, что для этой задачи от запуска к запуску будет истиной. Возможно, для ваших задач всегда понадобится конкретная установка актуатора (Actuator) или отображение конкретного Spring-профиля. Здесь можно дать соответствующие указания. Копия экрана пользовательского интерфейса на данном этапе работы показана на рис. 12.10.

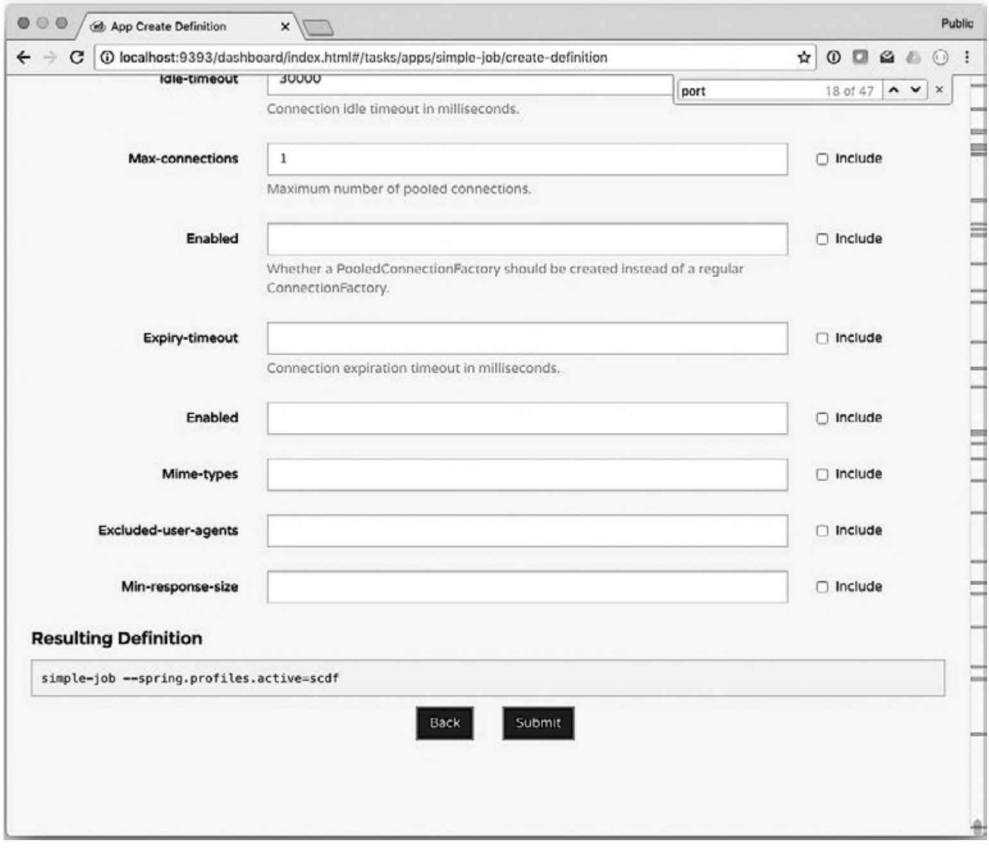


Рис. 12.10. Указание конкретного Spring-профиля

Экземпляр определения задачи можно запустить в любой нужный момент. Копия экрана пользовательского интерфейса на данном этапе работы показана на рис. 12.11.

При запуске экземпляра задачи будет выдан запрос на указание аргументов, относящихся к ее выполнению (тех, которые в качестве аргументов передаются таким средствам, как `CommandLineRunner` и `ApplicationRunner`, а в качестве параметров — заданию Spring Batch `Job`). Снимок экрана пользовательского интерфейса на данном этапе работы показан на рис. 12.12.

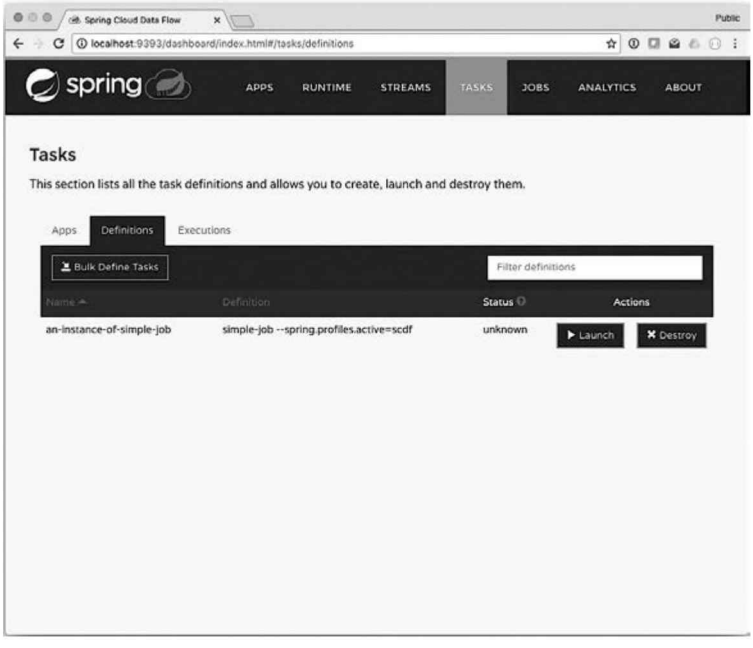


Рис. 12.11. Просмотр определений задач

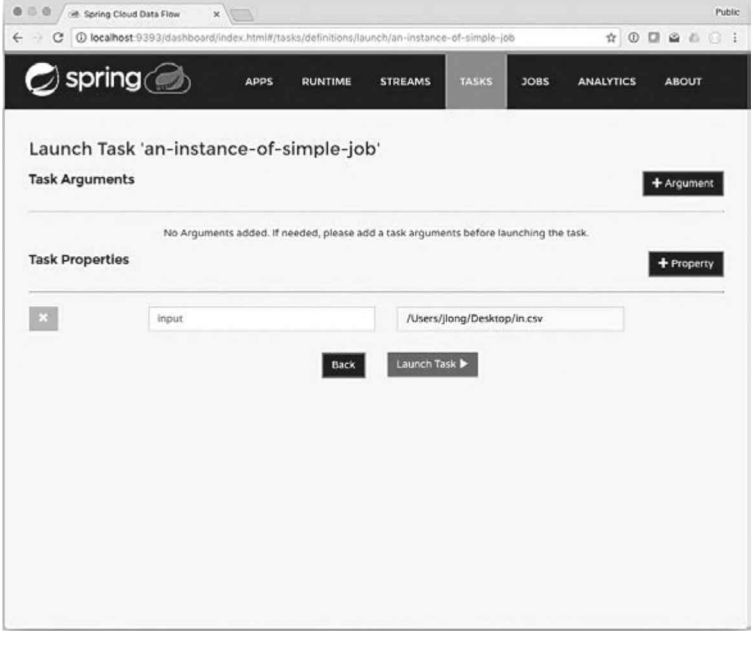


Рис. 12.12. Указание аргументов и свойств экземпляра

И наконец, запустите задачу, нажав кнопку Launch task (Запуск задачи)! Снимок экрана пользовательского интерфейса на данном этапе работы представлен на рис. 12.13.

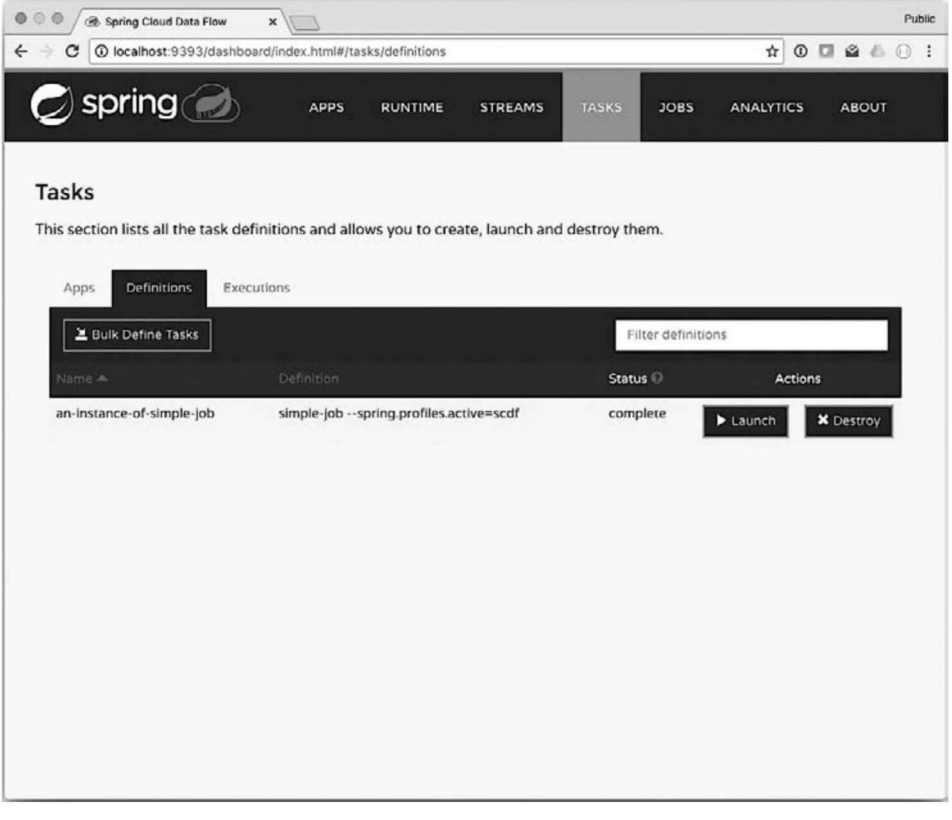


Рис. 12.13. Просмотр задач и их состояния

Оболочка Spring Cloud Data Flow

Если есть желание еще выше поднять свой уровень, то можно освоить интерактивную оболочку, поддерживаемую средой Spring Shell (https://github.com/spring-projects/spring-shell). Загрузите исполняемый файл оболочки, воспользовавшись инструкциями в документации, приведенной на странице Spring Cloud Data Flow (https://cloud.spring.io/spring-cloud-dataflow/). Как только оболочка будет установлена на локальной машине, запустите ее с помощью команды `java –jar spring-cloud-dataflow-server-local-....RELEASE.jar`, заполнив `....` указанием соответствующей версии загруженной оболочки.

Запустите оболочку (пример 12.22).

Пример 12.22. Пустая оболочка

```
  ____                              ____ _                __
 / ___|  _ __   _ __(_)_ __   __ _ / ___| | ___  _   _  __| |
 \___ \ | '_ \ | '__| | '_ \ / _` | |   | |/ _ \| | | |/ _` |
  ___) | | |_) |  |  | | | | | (_| | |___| | (_) | |_| | (_| |
 |____/ | .__/ |_|  |_|_| |_|\__, |  \____|_|\___/ \__,_|\__,_|
  ____  |_|  _            _ |___/          __________
 |  _ \  __ _| |_ __ _  | ___| | _____      __ \ \ \ \ \ \
 | | | |/ _` | __/ _` | | |_  | |/ _ \ \ /\ / /  \ \ \ \ \ \
 | |_| | (_| | || (_| | |  _| | | (_) \ V  V /    / / / / / /
 |____/ \__,_|\__\__,_| |_|   |_|\___/ \_/\_/    /_/_/_/_/_/

1.1.2.RELEASE
Welcome to the Spring Cloud Data Flow shell. For assistance hit TAB or type
"help".
dataflow:>app list
No registered apps.
You can register new apps with the 'app register' and 'app import' commands.
```

Запустите команду `app list`, чтобы увидеть, какие приложения установлены (пример 12.23).

Пример 12.23. Мы не станем следовать предлагаемым рекомендациям

```
dataflow:>app list
No registered apps.
You can register new apps with the 'app register' and 'app import' commands.
```

Ничего нет! Мы не будем изменять ситуацию предлагаемым образом. Импортируем ряд приложений, связанных с RabbitMQ и запакованных в Maven (пример 12.24).

Пример 12.24. Импортирование приложений, связанных с RabbitMQ

```
dataflow:>app import --uri http://bit.ly/stream-applications-rabbit-maven
```

Неплохо! Теперь создадим простой поток, воспользовавшись двумя исходными приложениями: `time`, создающим новое сообщение по таймеру, и `log`, регистрирующим сообщения, которые поступают на `stdout` (пример 12.25).

Пример 12.25. Установка канала Spring Cloud Data Flow

```
dataflow:>stream create --name ticktock --definition "time | log"  ❶
Created new stream 'ticktock'

dataflow:>stream deploy --name ticktock  ❷
Deployed stream 'ticktock'
```

❶ Определение потока, получающего созданные отметки времени и передающего их по каналу регистрационным записям.

❷ Поток нужно развернуть. Эта команда вызовет базовый механизм развертывания, который, в свою очередь, запустит модуль. В случае использования локаль-

ной среды он обычно запустит команду `java -jar ..` в отношении поддающегося разрешению модуля с расширением `.jar`. В Cloud Foundry для каждого приложения он запустит новые экземпляры.

И `time`, и `log` — полноценные приложения Spring Cloud Stream. Они предоставляют общеизвестные каналы: `output` для источника `time`, `input` для приемника `log`, а среда Spring Cloud Data Flow «сшивает» их, подготавливая для сообщений, которые покидают канал `output` приложения `time`, чтобы быть направленными на канал `input` приложения `log`. В консоли сервера Data Flow можно увидеть выходные данные, подтверждающие факт запуска и выполнения приложений, участвующих в работе потока (пример 12.26).

Пример 12.26. Регистрационные записи, подтверждающие запуск приложений и указывающие на различные журналы

```
...
.. : FrameworkServlet 'dispatcherServlet': initialization completed in 11 ms
.. : deploying app ticktock.log instance 0
   Logs will be in /var/folders/cr/grkckb753fld3lbmt386jp740000gn/T/spring-.log
.. o.s.c.d.spi.local.LocalAppDeployer : deploying app ticktock.time instance 0
   Logs will be in /var/folders/cr/grkckb753fld3lbmt386jp740000gn/T/spring-.time
...
```

Регистрационные записи `stdout` для приложения `log` должны показывать бесконечный поток новых меток времени.

Переместим в создаваемую структуру наш пользовательский процессор фигурных скобок. Сначала его нужно будет зарегистрировать (пример 12.27).

Пример 12.27. Использование оболочки для регистрации и развертывания потока

```
dataflow:>app register --name brackets --type processor \  ❶
   --uri maven://cnj:stream-example:jar:1.0.0-SNAPSHOT
Successfully registered application 'processor:brackets'


dataflow:>stream create --name bracketedticktock --definition "time | brackets |
  log"
Created new stream 'bracketedticktock'  ❷


dataflow:>stream list  ❸
```

Stream Name	Stream Definition	Status
bracketedticktock	time | brackets | log	undeployed

```
dataflow:>stream deploy --name bracketedticktock  ❹
Deployed stream 'bracketedticktock'
```

❶ Сначала нужно зарегистрировать приложение, чтобы оповестить о его наличии среду Spring Cloud Data Flow.

❷ Создание потока, «сшивающего» `time`, `log` и только что зарегистрированный компонент `bracket`.

❸ Подтверждение факта регистрации потока.

❹ Развертывание потока.

Теперь, используя `timestamp`, запустим простую задачу и посмотрим, как она выглядит (пример 12.28).

Пример 12.28. Запуск задачи timestamp

```
dataflow:>task create --name whattimeisit --definition timestamp  ❶
Created new task 'whattimeisit'


dataflow:>task list  ❷
```

Task Name	Task Definition	Task Status
whattimeisit	timestamp	unknown

```
dataflow:>task launch --name whattimeisit  ❸
Launched task 'whattimeisit'


dataflow:>task list  ❹
```

Task Name	Task Definition	Task Status
whattimeisit	timestamp	complete

❶ Создание задачи, запускающей обычную Spring Cloud Task по имени `timestamp`, и вывод значения на экран.

❷ Перечисление существующих задач.

❸ Запуск задачи и присвоение ей логического имени, по которому позже на нее можно будет ссылаться.

❹ Вывод списка задач. В данном случае задача завершена и мы видим отображение этого состояния.

Изучите регистрационные записи — и увидите выведенное время. А теперь запустим задачу на основе использования среды Spring Batch (пример 12.29).

Пример 12.29. Использование оболочки для регистрации и запуска нашей пользовательской задачи на основе использования среды Spring Batch

```
dataflow:>app register --name args --type task \  ❶
 --uri maven://cnj:task-example:jar:1.0.0-SNAPSHOT
```

```
Successfully registered application 'task:args'

dataflow:>task create --definition "args --p1=1 --p2=2" --name args  ❷
Created new task 'args'

dataflow:>task launch --name args  ❸
Launched task 'args'

dataflow:>task list  ❹
```

Task Name	Task Definition	Task Status
args	args	complete

❶ Сначала нужно зарегистрировать приложение, и мы получаем от среды Spring Cloud Data Flow соответствующее уведомление.

❷ Создание (потенциально) параметризированного экземпляра приложения.

❸ Запуск экземпляра задачи.

❹ Изучение состояния выхода из задачи.

Оболочка Spring Cloud Data Flow обладает весьма высокой эффективностью, поскольку поддерживает как сеансы интерактивной работы с сервером, *так и* возможность выполнения сценариев. Но при необходимости получить полный контроль следует присмотреться к шаблону `DataFlowTemplate`.

DataFlowTemplate

Для управления приложениями, потоками, задачами, пользователями и многим другим можно задействовать шаблон `RestTemplate` и работать с API Spring Cloud Data Flow. При посещении страницы по адресу http://localhost:9393 видно, что эти API удобно размещаются в качестве ссылок гиперсреды. Кроме того, можно выйти в конечные точки Spring Boot Actuator, показываемые по умолчанию на странице http://localhost:9393/management/. Попасть туда, приложив ряд усилий, можно будет после проведения аутентификации, обработки перемещений по связям гиперсреды и т. д. Если вместо этого воспользоваться JVM-машиной (которую, как мы надеемся, к данному месту книги вы уже освоили), то можно применить `DataFlowTemplate` — удобный клиентский интерфейс для всех интересных функциональных возможностей на сервере Spring Cloud Data Flow. Добавьте к пути к классам соответствующую библиотеку: `org.springframework.cloud : spring-cloud-dataflow-rest-client`. С помощью `DataFlowTemplate` будет удобно устанавливать все, что нужно проинициализировать, при запуске приложения в сервере Spring Cloud Data Flow. Эта инициализация может включать определения

приложений, использующих их потоки и задачи, и развертывания последних. Рассмотрим обычный компонент, отслеживающий события содержимого приложения `ApplicationReadyEvent` (которые Spring Boot публикует практически сразу после запуска приложения при его инициализации) и обменивающийся данными с только что установленным сервером (пример 12.30).

Пример 12.30. Инициализатор Data Flow

```
package dataflow;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.cloud.dataflow.rest.client.DataFlowTemplate;
import org.springframework.cloud.dataflow.rest.client.TaskOperations;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import java.net.URI;
import java.net.URL;
import java.util.*;
import java.util.stream.Stream;

@Component
class DataFlowInitializer {

 private final URI baseUri;

 @Autowired
 public DataFlowInitializer(@Value("${server.port}") int port) {
  this.baseUri = URI.create("http://localhost:" + port);
 }

 @EventListener(ApplicationReadyEvent.class)
 public void deployOnStart(ApplicationReadyEvent e) throws Exception {

  ❶
  DataFlowTemplate df = new DataFlowTemplate(baseUri, new RestTemplate());

  ❷
  Stream
   .of("http://bit.ly/stream-applications-rabbit-maven",
    new URL(baseUri.toURL(), "app.properties").toString()).parallel()
   .forEach(u -> df.appRegistryOperations().importFromResource(u, true));


  ❸
  TaskOperations taskOperations = df.taskOperations();
```

```
  Stream.of("batch-task", "simple-task").forEach(tn -> {
   String name = "my-" + tn;
   taskOperations.create(name, tn);
   Map<String, String> properties = Collections.singletonMap(
    "simple-batch-task.input", System.getenv("HOME") + "Desktop/in.csv");
   List<String> arguments = Arrays.asList("input=in", "output=out");
   taskOperations.launch(name, properties, arguments);
  });

  ❹
  Map<String, String> streams = new HashMap<>();
  streams.put("bracket-time", "time | brackets | log");
  streams
   .entrySet()
   .parallelStream()
   .forEach(
    stream -> df.streamOperations().createStream(stream.getKey(),
     stream.getValue(), true));
  }
}
```

❶ Определение `DataFlowTemplate` максимально позже инициализации сервера Data Flow (при публикации Spring ApplicationContext-события `ApplicationReadyEvent`), поскольку `DataFlowTemplate` выполняет в своем конструкторе REST-вызовы в адрес сервера Data Flow. Прежде чем выполнить вызов, нужно убедиться в готовности REST API.

❷ Предварительные требования к приложениям, относящиеся как к стандартному набору определений приложений, так и к ряду наших определений пользовательских задач и потоков.

❸ Регистрация и запуск экземпляров двух задач, `batch-task` и `simple-task`.

❹ Этот код определяет (и развертывает) поток `bracket-time`.

Среда Spring Cloud Data Flow позволяет разработчикам создавать системы высшего порядка из небольших компонентов, ориентированных на выполнение единственной задачи. Среда оптимизирована под масштабируемую обработку и интеграцию приложений и данных. По сути, Spring Cloud Data Flow — механизм пакетной обработки и рассылки для мира облачных вычислений.

Резюме

Мы лишь приступили к изучению возможностей обработки данных, еще не погрузившись в их глубины. По своей природе каждая из этих технологий, в свою очередь, связана со множеством различных других технологий. Так, задействуя Spring Cloud Data Flow, можно воспользоваться Apache Spark или Hadoop. Хорошо

сочетаются друг с другом Spring Cloud Task и Activiti или механизм Spring State. Вполне по силам будет распространить обработку задания Spring Batch по кластеру, применив, скажем, в качестве фабрики сообщений среду Spring Cloud Stream. Мы рассмотрели технические приемы и технологии, разработанные для поддержки восстановления правильных результатов при обработке данных. Сначала был изучен метод повторных попыток и метод восстановления данных, затем рассмотрено применение шаблонов, построенных на использовании рассылки сообщений, таких как сага-шаблон, разделение ответственности на команды и запросы — Command Query Responsibility Segregation (CQRS) и применение многоступенчатых архитектур, управляемых событиями, — staged event-driven architectures (SEDA).

Часть IV
Промышленная эксплуатация

13 Отслеживаемая система

Для создания систем, ценных с точки зрения производства, существует два технических аспекта.

- *Нужно создать систему, обладающую достаточной масштабируемостью и способную справляться с бизнес-требованиями.* На этом аспекте мы в основном и заостряли внимание в данной книге. Рассматривали шаблоны, поддерживающие безопасное внедрение распределения в систему, которые, в свою очередь, поддерживают облегченное распределение работы. Мы по большому счету отмахнулись от вопросов планирования мощностей, поскольку нашей основной целью стало избавление от единой точки отказа, а не освоение масштабов Google. Хорошо разделенная система способна еще давать сбои в достаточно более широком масштабе, но подобные вопросы можно решать по мере их поступления.

  Идя навстречу этим требованиям, нужно четко понимать, что подразумевается под *масштабируемостью*. Это понятие можно охарактеризовать в системе множеством различных способов, но пока вы не осмыслите место своей системы, невозможно будет понять, по каким направлениям вы хотите ее развивать. С этими оценочными критериями можно будет подойти к модели масштабирования вашей системы. Одна весьма полезная модель определена законом универсальной масштабируемости — Universal Scalability Law (http://www.perfdynamics.com/Manifesto/USLscalability.html), который был выведен Бароном Шварцем (Baron Schwartz) в книге Practical Scalability Analysis with the Universal Scalability Law (https://www.vividcortex.com/resources/universal-scalability-law/). Кода Хейл (Coda Hale) создал неплохую библиотеку моделирования usl4j (https://github.com/codahale/usl4j), которая — при заданных наблюдениях (оценках) двух величин закона Литтла (Little's law) (https://en.wikipedia.org/wiki/Little%27s_law) — дает модель прогнозирования одной величины из другой. Такая модель поддерживает масштабирование, отвечающее конкретным требованиям. Благодаря этому (при условии поддержки горизонтального масштабирования) облегчается жизнь облачных платформ, подобных Cloud Foundry. Порядок соответствующих оценок системы будет рассмотрен в данной главе. Такие оценки способствуют созданию моделей масштабирования.

- *Нужно создать систему, успешно справляющуюся с неожиданностями в своей работе.* Приложение должно поддерживать простые способы восстановления.

  Оба этих требования предполагают понимание сути системы, то есть наличие умения произвести ее оценку. Кроме технических требований, понимание сути распространяется также и на бизнес-аспекты. Если действовать поэтапно и по-настоящему гибко, то программный продукт после каждого этапа доработки, если он даже еще не выпущен в новой версии, должен все же доставляться и публиковаться. Заказчиком может быть клиент, некоммерческая организация, проект с открытым кодом или вы сами. Заказчик дает описание того, что именно придает ценность программному продукту. Чем быстрее программа попадет к заказчику, тем раньше проявится ее польза. Выпущенное и уже используемое в работе ПО снижает риски, сопряженные с непрерывной разработкой, за счет приобретения бизнес-ценности.

А как узнать, пригодна ли программа к работе? Само ПО сохраняет молчание. Оно ведет себя совершенно спокойно как в рабочем, так и в нерабочем состоянии. При создании приложения в него нужно встраивать показатели *работоспособности*. Это помогает рассчитывать на правильное поведение программы по *главным показателям*. Они служат основой для оцениваемых улучшений в поведении.

В данной главе будут рассмотрены *нефункциональные* или *межфункциональные* возможности, в которых нуждаются все приложения, если мы надеемся ввести их в эксплуатацию. Основное внимание в главе будет уделено возможности *отслеживания состояния* — оценке того, насколько адекватно внутренние состояния системы могут быть получены за счет знания их внешних проявлений.

Проблема в следующем: эти возможности *не являются* факторами, определяющими различия в бизнесе; они не являются тем, что в вашей организации относится к бизнесу! И тем не менее они имеют решающее значение для непрерывной и безопасной работы и развития приложения. Майкл Нигард (Michael Nygard) в своем эпическом фолианте Release It! (Pragmatic) выделил данные возможности и все, что имеет к ним отношение, в особую, трудную для осмысления категорию. Кульминационным моментом в его рассуждениях стало то, что завершение разработки кода еще не означает готовности последнего к практическому применению. Программа не может быть выпущена, если не готова к эксплуатации. Она должна быть в завершенном, работоспособном состоянии.

Вы это создали, и вам же с этим работать

Исторически у разработчиков образовалась тенденция к игнорированию подобного вида нефункциональных требований. В конце концов, если набор функций формируется не ими и заказчики не приветствуют их участия в данном процессе, то зачем вообще заострять на этом внимание? Естественно, при создании проекта

подобные вопросы вообще не должны волновать! Разработчики не желают возиться с корректировкой кода для поддержки данных требований в условиях постоянно растущего дефицита времени, отводимого на добавление функциональных возможностей, которые связаны с видоизменениями бизнеса. Какое им дело до того, что программа даст сбой глубокой ночью, с этим разберется служба поддержки! Но никакой службе поддержки или оперативной службе не захочется обременять себя программой в виде черного ящика (совершенно для них невидимой), которая начинает сбоить в четыре утра. Поэтому они проявили особую заботу об отслеживаемости и сделали все от них зависящее, чтобы сохранить изменения, предназначенные для поддержки программы. В результате разработчики станут создавать программы и перебрасывать их через воображаемую стену оперативной службе, а та будет оснащать все, что можно, инструментами, не прибегая к изменению кода.

Как только разработчики и операторы начинают осознавать существование друг друга, то, как правило, возникают разногласия по поводу предполагаемой потери автономии, воспринимаемой разработчиками из-за ввода приложения в эксплуатацию. Разработчикам хочется выпустить код, а операторам — убедиться в стабильности системы, запущенной в производство. Такое развитие событий далеко от идеала. Возникает нестыковка бизнес-результатов и стабильности системы. Разумеется, в наше время разговор ведется о *DevOps*, то есть об идее о том, что и операторы, и разработчики должны обеспечивать как стабильность системы, так и достижение бизнес-результатов. И те и другие — две стороны одной монеты, работающие над достижением общих целей. Многие весьма успешно работающие организации приняли простую установку для сведения по времени разработчиков и операторов: «Вы это создали, и вам же с этим работать». Теперь командам поручена поддержка их кода на стадии эксплуатации.

Если вы знаете о том, что вас могут разбудить в четыре утра для устранения сбоя системы, то будете крайне заинтересованы в создании надежных систем. Таких, которые не будут иметь уязвимостей и будут включать поддержку возможностей отслеживания сбоев и быстрого восстановления работоспособности!

Таинственные убийства, связанные с микросервисами

Еще более значимой поддержка отслеживаемости становится в облаке при создании распределенных систем. При использовании микросервисов каждый перерыв в работе сродни таинственному убийству (https://twitter.com/honest_update/status/651897353889259520). В журнале Increment (представляющем новый захватывающий экскурс в мир работы дежурных операторов) рассматривается вопрос:

«Что происходит, когда поступает срочный вызов?» (https://increment.com/on-call/when-the-pager-goes-off/). В статье дается описание стандартной высокоуровневой структуры, которой в ответ на возникновение аварийной ситуации придерживается основная часть успешных организаций отрасли информационных технологий. В общих чертах основные этапы выглядят следующим образом.

- *Расстановка приоритетов.* Некий элемент в системе вышел из строя. На данном этапе задача заключается в определении того, где именно *может* случиться сбой, в оценке воздействия и серьезности инцидента и в его классификации.
- *Координация действий.* Здесь необходимо приступить к минимизации негативных последствий. Эту работу может проделать другая команда (разработчики) или та же команда, которая занималась расстановкой приоритетов при исчерпании инцидента. Как правило, общение групп происходит по установленным каналам (чат, Slack, Skype, Google Hangouts и т. д.). Работа должна быть задокументирована и доведена до всех заинтересованных структур. Во многих организациях создаются штабы ликвидации последствий и настраивается конференц-связь.
- *Устранение негативных последствий.* На данном этапе целью ставится снижение отрицательного влияния произошедшего инцидента и восстановление стабильности системы, но не устранение основной причины и не ликвидация сути проблемы. К примеру, если в системе произошел сбой по причине развертывания, то этап устранения негативных последствий будет заключаться в откате изменений, а не в попытке устранения основной проблемы.
- *Решение проблемы.* Устранение последствий может лишь остановить развитие инцидента, исключив его дальнейшее влияние на других пользователей, но не избавить систему от основной причины сбоя. На этапе решения проблемы разработчики анализируют и устраняют основные причины сбоя. Активные пользователи могут испытывать затруднения, требуя решения проблемы. Основное значение имеет время, но команды должны проявлять осторожность, соблюдая обычные приемы защиты качества, такие как тестирование любых исправлений. Во многих организациях оценивается как среднее время ликвидации аварии (mean time to mitigation, MTTM), так и среднее время восстановления (mean time to resolution, MTTR).
- *Последующая работа.* На данном этапе организация стремится извлечь уроки из инцидента путем тщательного анализа произошедшего, выработки и постановки задач дальнейшей работы и проведения совещаний по разбору инцидентов. Разбор происходит только после выполнения всех задач последующей работы.

На каждом этапе *чрезвычайно* важно получать в организации поддержку в вопросах обмена мнениями и возможности визуального контроля внутри самой системы.

Операции двенадцати факторов

Приложение, введенное в действие, предназначено для промышленного применения. В данной главе мы покажем, что такое приложение (помимо функционирования в рамках решения бизнес-задач, являющихся его отличительным признаком) будет взаимодействовать с экосистемой разнообразных инструментов (централизованной обработкой регистрационных записей, программами управления работоспособностью и процессом, диспетчерами заданий, распределенными системами трассировки и т. д.).

Мы постараемся, создавая приложения с прицелом на соблюдение основных положений манифеста 12-факторного приложения и работая с учетом соглашений Spring Boot и Spring Cloud, извлечь пользу из этой инфраструктуры, если она доступна. Но не стоит заблуждаться: нам понадобится нечто, способное предложить эту инфраструктуру. Развертывание в производственную конфигурацию вслепую, без поддерживающей инфраструктуры нам не подойдет. Если в манифесте 12-факторного приложения дается описание набора полезных, понятных, поддерживающих чистоту облачных вычислений принципов для создания приложений, готовых к эксплуатации, то нужно что-то для выполнения второй стороны контракта и поддержки того, что Эндрю Клей Шафер (Andrew Clay Shafer) (https://twitter.com/littleidea) назвал *операциями 12 факторов*. Нужен некий элемент, который запускает приложения и сервисы в виде процессов, не сохраняющих состояния, предоставляет общеизвестный жизненный цикл приложения, упрощает внешнее конфигурирование, поддерживает управление регистрационными записями, обеспечивает опорные сервисы, облегчает горизонтальное масштабирование приложений, предоставляет декларативную привязку портов и т. д. И конечно же, с поддержкой всех эксплуатационных требований отлично справляется Cloud Foundry.

В данной главе мы рассмотрим способы поочередного извлечения информации из узлов и способы централизации этой информации в целях поддержки получения на одном экране общего представления, которое необходимо для оперативной оценки поведения системы. Здесь важно то, что мы получаем картину поведения *системы*, а не приложения в ней.

Карта не есть территория. Точно так же, как при разглядывании карты Манхэттена получаешь о нем куда менее четкое представление, чем при прогулке по самому району, система обладает непредсказуемым поведением, которое не в состоянии отобразиться в схеме архитектуры этой системы. Его можно зафиксировать лишь с помощью эффективного мониторинга, охватывающего всю систему.

Новый курс

В требованиях к успешному развертыванию приложений на стадии их эксплуатации кардинальных изменений не произошло. Изменения коснулись того, насколько в организации, которая должна добиться успеха, обособленные разработчики могут

позволить себе отстраниться от проблем, с которыми сталкиваются операторы, и насколько безразличными могут быть операторы к требованиям приложения, способным угрожать стабильности приложения. Раньше эстафета от разработчиков к операторам была некой непонятной поставкой, чем-то напоминающей нечто совместимое с контейнером сервлета `.war`. Приложение, развернутое в этом «черном ящике», пользовалось рядом предоставляемых в контейнере сервисов, наподобие встроенного сценария запуска-остановки и центрального канала регистрации. Для извлечения пользы из любого контейнера операторам приходилось к ним приспосабливаться. Затем операторам нужно было обзавестись полным набором поддерживающей инфраструктуры, чтобы добиться стабильной работы приложения на стадии эксплуатации.

- *Планирование и управление процессами.* Какой компонент запустит приложение и изящно завершит его работу? Как обеспечить, что оно не будет дважды запущено на одном и том же хосте?
- *Поддержка и восстановление работоспособности приложения.* Как операторы смогут узнать о том, что приложение работает успешно? Что произойдет при аварийной остановке процесса? Что будет при отказе самого хоста?
- *Управление регистрационными записями.* Как операторы видят регистрационные записи, заполняемые из экземпляров приложений? Как осуществляется сбор и анализ?
- *Видимость и информационная открытость приложений.* Как операторы фиксируют состояние приложения или количественно определяют состояние приложения по показателям, а также проводят их анализ и визуализацию?
- *Управление маршрутизацией.* Доступно ли приложение из Интернета? Имеет ли оно сбалансированную нагрузку? Происходит ли соответствующая корректировка маршрутов при перезапуске приложения на другом хосте?
- *Распределенная трассировка.* Кто обращается к системе? Какие сервисы участвуют в обработке транзакции и какова задержка соответствующих запросов? Как выглядит усредненный график вызовов?
- *Управление производительностью приложения.* Каким образом операторы проводят диагностику и решают сложные проблемы производительности приложения?

> Плохая система неизменно будет одерживать верх над хорошим человеком.
>
> *Уильям Эдвардс Деминг (W. Edwards Deming)*

Деминг сказал это применительно к людям, а не к распределенным системам или системам, под управлением которых они работают, но мы полагаем, что данное высказывание подходит и в нашем случае. Если система, используемая нами для управления нашими программами, мешает им работать правильно, то люди не смогут что-либо изменить.

Крайне важно, чтобы поддержка отслеживаемости и соответствующих передовых приемов была максимально свободной от всяческих трений — это позволит упростить ее внедрение во все сервисы и проекты. Как крупным, так и мелким организациям известна грозная корпоративная Wiki-страница (The 500 Easy Steps to Production («500 легких шагов к эксплуатации»)), заполненная описаниями однообразной ручной работы, проделываемой перед развертыванием сервиса. И здесь на страже наших интересов свои сильные стороны проявляют среды Cloud Foundry и Spring Boot. Первая предоставляет поддержку требований операций 12 факторов, а вторая (и имеющееся в ней автоконфигурирование, о котором шла речь в главе 2 при рассмотрении основ Spring Boot) подстраивает приложение под соблюдение принципов 12 факторов. Эти среды сводят издержки на осмысление правильных действий практически к нулю. Остается только все единожды сделать правильно, а затем многократно этим пользоваться. Однообразная трудоемкая работа — враг скорости.

Такая инфраструктура трудно поддается правильному формированию и весьма затратна в разработке. Кроме того, очень важно понимать: она никак не связана с функционированием в рамках решения бизнес-задач. Мы считаем, что такие категоричные платформы, как Heroku или дистрибутивы открытого кода Cloud Foundry, предоставляют наилучшее сочетание поддерживающей инфраструктуры и легкости применения. При использовании этих платформ делаются предположения, что приложения имеют транзакционную природу, являются онлайн-веб-приложениями, написанными в соответствии с принципами манифеста 12-факторного приложения. Эти предположения упрощают для платформы соответствие конкретным требованиям; они сокращают область возможных изменений в кодовой базе. Благодаря им поддерживается автоматизация и увеличивается скорость разработки.

Отслеживаемость

Отслеживаемости достичь непросто. При использовании монолитного приложения задача несколько упрощается. Если в системе что-то идет не так, то по крайней мере не возникают вопросы о том, где именно произошла ошибка: она произошла внутри сборки! В распределенной системе все существенно усложняется, поскольку взаимодействие между компонентами приводит к критической изоляции сбоев. Вы бы не стали ездить на одноместной машине без приборов, датчиков и окон, позволяющих видеть ситуацию; а как бы вы без видения ситуации надеялись справиться с регулировкой воздушного движения сотен самолетов?

Улучшенная видимость способствует непрекращающемуся совершенствованию бизнес-составляющей и смене приоритетов системы. Операторы используют хорошую видимость по крайней мере для непосредственного отслеживания потенциальных системных проблем (оповещения) и для оказания помощи в ответ на инциденты. В ряде случаев они могут также воспользоваться хорошей отслежива-

емостью, чтобы создать автоматизированную или как минимум простую реакцию на инциденты, которая заключается в одном нажатии кнопки.

В идеале операционная и бизнес-видимость могут коррелироваться и использоваться для управления единым восприятием ситуации на одном экране — на панели управления. В данной главе будут рассмотрены способы сбора и анализа *исторических* и *нынешних* состояний и способы поддержки прогнозируемых *расчетных данных*. Последние, в частности, выстраиваются на основе моделей, опирающихся на исторические данные.

Историческими называются данные, хранящиеся *где-либо* за какой-нибудь период времени. Они могут способствовать долгосрочному осмыслению бизнеса, а более свежие данные способны поддерживать ведение операционной телеметрии, анализ ошибок и отладку.

Ваша система представляет собой довольно сложный механизм, который имеет большое количество движущихся частей. Трудно понять, какую именно информацию следует собирать, а какую нужно просто проигнорировать, поэтому лучше перестраховаться и собрать как можно больше данных.

Сравнение отслеживаемости и частоты получения данных при внедрении и извлечении

Некоторые средства мониторинга и отслеживаемости используют подход на основе *извлечения*, при котором централизованная инфраструктура извлекает данные из сервисов через определенные интервалы времени, а некая инфраструктура мониторинга ожидает события, связанные с состоянием различных узлов, для *внедрения* в них этой информации. Многие средства, которые будут рассмотрены в данной главе, могут работать в одном или в другом стиле или иногда применять сразу оба. Решение по выбору подхода остается за вами.

Для многих организаций одним из обсуждаемых вопросов является частота получения данных. Какова частота обновления инфраструктуры мониторинга? В динамической среде приход и уход всего окружающего могут происходить по мере необходимости. Разумеется, когда речь заходит о специализированных задачах, срок действия сервиса может исчисляться секундами или минутами. Если в системе используется мониторинг на основе извлечения данных, то интервал между извлечениями может превышать весь срок работы приложения! Инфраструктура мониторинга «слепнет» по отношению к целым работающим компонентам и может пропускать существенные изменения данных в ту или иную сторону. Это становится одним из самых весомых аргументов в пользу выбора для таких компонентов мониторинга на основе внедрения данных.

И здесь значительную помощь может оказать гибкость Spring: эта среда зачастую предоставляет события, которыми можно воспользоваться для инициирования событий мониторинга. По мере чтения этой главы спросите себя, на чем основан данный подход: на извлечении (pull) или внедрении (push) — и как в случае необходимости можно было бы превратить нечто имеющее в основе извлечение во что-то использующее внедрение.

Получение текущего состояния приложения с помощью Spring Boot Actuator

Текущее состояние приложения относится к информации, которую следует выводить на информационную панель, а затем, возможно, отобразить в офисе на большом экране, на котором она будет видна всем присутствующим. Эту информацию можно не сохранять или сохранять для дальнейшего использования. Текущее состояние похоже на спидометр автомобиля: его задача — лаконичнее и как можно быстрее выяснить наличие или отсутствие проблемы.

Если нужно выделить состояние системы при его визуализации (красным цветом для выражения опасности, зеленым — для выражения нормального хода событий или желтым — для выражения негативных, но все еще находящихся в пределах допустимого изменений), то на какую информацию падет ваш выбор? На информацию о текущем состоянии.

Текущее состояние может содержать информацию о памяти, пулах потоков, подключениях к базам данных и общем количестве обработанных запросов. В него может входить статистика, например количество запросов в секунду (для всех частей системы, у которых имеются запросы, включая конечные точки HTTP и очереди сообщений), 95 процентилей для показателей времени отклика, обнаруженных ошибок и состояния автоматов защиты.

Среда *Spring Boot Actuator* предоставляет готовую поддержку для получаемой через конечные точки информации о приложении. Точки собирают информацию и иногда взаимодействуют с другими подсистемами. Просмотреть эти конечные точки можно множеством различных способов (используя, к примеру, технологии REST или JMX). Наше внимание будет сконцентрировано на конечных точках REST. Вообще, точки можно подключать, и различные подсистемы на основе Spring Boot часто привносят дополнительные конечные точки. Чтобы воспользоваться средой Spring Boot Actuator, к сборке проекта следует добавить строку `org.springframework.boot : spring-boot-starter-actuator`. Пользовательские конечные точки будут предоставляться с помощью технологии REST, если добавить строку `org.springframework.boot : spring-boot-starter-web`.

В документации по Spring Boot (https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/) фигурирует следующее описание: «Актуатор — производственный термин, ссылающийся на механическое устройство для перемещения чего-либо или управления чем-либо. Актуаторы могут из небольшого изменения создать существенное перемещение».

Некоторые конечные точки Actuator перечислены в табл. 13.1.

Таблица 13.1. Некоторые конечные точки Actuator

Конечная точка	Использование
/info	Предоставляет информацию о текущем сервисе
/metrics	Выдает количественные показатели о сервисе
/beans	Предоставляет схему всех объектов, созданных для вас средой Spring Boot
/configprops	Показывает информацию обо всех свойствах, доступных для конфигурирования текущего приложения Spring Boot
/mappings	Предоставляет сведения обо всех конечных точках HTTP, известных среде Spring Boot в текущем приложении, а также другие метаданные (например, указанные типы содержимого или операции HTTP в отображении Spring MVC)
/health	Выводит описание компонентов системы: UP, DOWN и т. д. Кроме того, возвращает коды состояния HTTP
/loggers	Показывает и вносит изменения в средства ведения учетных записей
/auditevents	Показывает все экземпляры AuditEvent, записанные с помощью AuditEventRepository. Речь идет о событиях, относящихся к подключению аутентифицированных объектов Principal к событиям в системе. Может также сохранять и выдавать пользовательские события
/cloudfoundryapplication	Предоставляет информацию из пользовательского интерфейса управления на основе Cloud Foundry, дополненную информацией Spring Boot Actuator. Страница состояния приложения может включать полный вывод Spring Boot /health в дополнение к running (выполняемым) или stopped (остановленным). Эта информация является защищенной и требует действующий токен от сервиса аутентификации и авторизации Cloud Foundry UAA. Если ваше приложение выполняется не в среде Cloud Foundry, то данную конечную точку можно отключить с помощью строки management.cloudfoundry.enabled=false
/env	Возвращает все известные свойства среды, например те, которые являются переменными среды операционной системы или результатами выполнения метода System.getProperties()

Рассмотрим некоторые конечные точки более подробно.

Показатели

> Все обобщения делаются на основе слишком малого количества показателей. По крайней мере это можно отнести на мой счет.
>
> *Паранд Даругар (Parand Darugar) (https://twitter.com/parand/status/854793437043761152)*

В качестве показателей выступают числа. В среде Spring Boot Actuator имеется три разновидности показателей: *открытые* (которые мы рассмотрим чуть ниже), *датчики* и *счетчики*. Датчики за один раз записывают одно значение и не требуют табуляции. Счетчик записывает разницу (приращение или уменьшение), он представляет собой значение, достигаемое со временем, и использует табуляцию. Показатели являются числами, поэтому их просто сохранять, сводить в схему и получать по запросу. Есть показатели, о которых будут заботиться исключительно операции: это информация о хосте, касающаяся использования оперативной памяти, дискового пространства и количества запросов в секунду. Все сотрудники будут интересоваться семантическими показателями: сколько запросов было сделано за последний час, сколько заказов размещено, новых учетных записей зарегистрировано, сколько товаров было продано и в каком количестве и т. д. Изначально среда Spring Boot предоставляет эти показатели в конечной точке `/metrics` (пример 13.1).

Пример 13.1. Показатели из конечной точки приложения /metrics

```
{
   "classes" : 9731,
   "heap.committed" : 570368,
   "nonheap.used" : 72430,
   "systemload.average" : 3.328125,
   "gauge.response.customers.id" : 7,
   "gc.ps_marksweep.count" : 2,
   "nonheap" : 0,
   "counter.status.200.customers" : 1,  ❶
   "counter.status.200.customers.id" : 2,
   "mem.free" : 390762,
   "heap.used" : 179605,
   "classes.unloaded" : 0,
   "gauge.response.star-star.favicon.ico" : 4,
   "instance.uptime" : 47231,
   "counter.status.200.star-star.favicon.ico" : 2,
   "threads.peak" : 21,
   "nonheap.init" : 2496,
   "threads.totalStarted" : 27,
   "mem" : 642797,
   "httpsessions.max" : -1,
   "counter.customers.read.found" : 2,
   "gc.ps_marksweep.time" : 96,
```

```
   "uptime" : 52379,
   "threads" : 21,
   "customers.count" : 6,
   "gc.ps_scavenge.count" : 6,
   "heap.init" : 262144,
   "httpsessions.active" : 0,
   "nonheap.committed" : 74112,
   "gc.ps_scavenge.time" : 87,
   "counter.status.200.admin.metrics" : 2,
   "datasource.primary.usage" : 0,
   "processors" : 8,  ❷
   "gauge.response.customers" : 9,
   "heap" : 3728384,
   "gauge.response.admin.metrics" : 4,
   "threads.daemon" : 19,
   "datasource.primary.active" : 0,
   "classes.loaded" : 9731
}
```

❶ Показатели включают количество запросов, их пути и коды их HTTP-состояния. Здесь `200` — код HTTP-состояния.

❷ Среда Spring Boot Actuator также собирает другую основную информацию о системе, например количество доступных процессоров.

Показатели уже записывают для нас массу полезной информации: все сделанные запросы (и соответствующие коды HTTP-состояния) и информацию о среде (например, потоки, принадлежащие JVM, загруженные классы и сведения о любых сконфигурированных экземплярах `DataSource`). Сообразуясь с условиями, среда Spring Boot регистрирует показатели на основе задействованных подсистем.

Можно внести свой вклад в показатели, воспользовавшись сервисом `org.springframework.boot.actuate.metrics.CounterService` для записи разницы («сделан еще один запрос») или сервисом `org.springframework.boot.actuate.metrics.GaugeService` для получения абсолютных значений («к этому чату подключено 140 пользователей») (пример 13.2).

Пример 13.2. Сбор пользовательских показателей с помощью CounterService

```
package demo.metrics;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.CounterService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import java.net.URI;

@RestController
@RequestMapping("/customers")
```

```
public class CustomerRestController {

 private final CounterService counterService; ❶

 private final CustomerRepository customerRepository;

 @Autowired
 CustomerRestController(CustomerRepository repository,
  CounterService counterService) {
  this.customerRepository = repository;
  this.counterService = counterService;
 }

 @RequestMapping(method = RequestMethod.GET, value = "/{id}")
 ResponseEntity<?> get(@PathVariable Long id) {
  return this.customerRepository.findById(id).map(customer -> {
   String metricName = metricPrefix("customers.read.found");
   this.counterService.increment(metricName); ❷
   return ResponseEntity.ok(customer);
  }).orElseGet(() -> {
   String metricName = metricPrefix("customers.read.not-found");
   this.counterService.increment(metricName); ❸
   return ResponseEntity.class.cast(ResponseEntity.notFound().build());
  });
 }

 @RequestMapping(method = RequestMethod.POST)
 ResponseEntity<?> add(@RequestBody Customer newCustomer) {
  this.customerRepository.save(newCustomer);
  ServletUriComponentsBuilder url = ServletUriComponentsBuilder
   .fromCurrentRequest();
  URI location = url.path("/{id}").buildAndExpand(newCustomer.getId()).toUri();
  return ResponseEntity.created(location).build();
 }

 @RequestMapping(method = RequestMethod.DELETE)
 ResponseEntity<?> delete(@PathVariable Long id) {
  this.customerRepository.delete(id);
  return ResponseEntity.notFound().build();
 }

 @RequestMapping(method = RequestMethod.GET)
 ResponseEntity<?> get() {
  return ResponseEntity.ok(this.customerRepository.findAll());
 }

 ❹
 protected String metricPrefix(String k) {
  return k;
 }

}
```

1. `CounterService` сконфигурирован для вас в автоматическом режиме. Если вы задействуете Java 8, то получите более производительную реализацию, чем при использовании прежних версий Java.
2. Запись количества запросов, в результате которых было успешно найдено соответствие...
3. ...и количества промахов.
4. В следующем примере этот метод будет переопределен, чтобы изменить ключ, используемый для показателя.

Сервисы `CounterService` и `GaugeService` получают показатели по мере их обновления в транзакциях. Для получения показателей нужно обновить код, чтобы выводить верные показатели при достойном внимания событии. Вставить инструменты в путь запроса бизнес-компонентов не так-то легко, вероятно, проще будет получить эти показатели чуть позже. Интерфейс `org.springframework.boot.actuate.endpoint.PublicMetrics` среды Spring Boot Actuator поддерживает централизованный сбор показателей. Spring Boot предоставляет внутреннюю реализацию этого интерфейса для выявления информации о среде JVM, Apache Tomcat, сконфигурированного компонента `DataSource` и т. д. В примере 13.3 будет рассмотрен код, получающий информацию о клиентах в нашем приложении.

Пример 13.3. Пользовательская реализация PublicMetrics, предоставляющая количество клиентов, присутствующих в системе

```
package demo.metrics;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.endpoint.PublicMetrics;
import org.springframework.boot.actuate.metrics.Metric;
import org.springframework.stereotype.Component;

import java.util.Collection;
import java.util.HashSet;
import java.util.Set;

@Component
class CustomerPublicMetrics implements PublicMetrics {

 private final CustomerRepository customerRepository;

 @Autowired
 public CustomerPublicMetrics(CustomerRepository customerRepository) {
  this.customerRepository = customerRepository;
 }

 @Override
```

```
 public Collection<Metric<?>> metrics() {

  Set<Metric<?>> metrics = new HashSet<>();

  long count = this.customerRepository.count();

  ❶
  Metric<Number> customersCountMetric = new Metric<>("customers.count", count);
  metrics.add(customersCountMetric);
  return metrics;
 }
}
```

❶ Этот показатель содержит совокупное количество записей клиентов `Customer` в базе данных.

До сих пор мы рассматривали показатели в качестве величин на фиксированный момент времени. Они представляли значение на момент его просмотра. Показатели пригодны к использованию, но не имеют контекста. Для некоторых значений фиксированное во времени значение не имеет смысла. При отсутствии истории, то есть без видения перспективы во времени, трудно узнать, что именно представляет значение: улучшение или ухудшение. Благодаря оси времени можно получить значение и вывести статистику: средние арифметические значения, средние значения выборок, математические ожидания, процентили и т. д.

Среда Spring Boot Actuator хорошо интегрируется с библиотекой Dropwizard Metrics. Кода Хейл (Coda Hale) (https://twitter.com/coda), работая в Yammer, создал эту библиотеку для получения способов оценки, подсчетов и нескольких других типов показателей. Чтобы приступить к работе со средствами Dropwizard Metrics, следует к пути к классам добавить строку `io.dropwizard.metrics : metrics-core`.

Библиотека Dropwizard Metrics включает поддержку показателей. *Показатель* выполняет замер количества событий во времени (например, «заказы в секунду»). Если в пути к классам указана библиотека Dropwizard Metrics и перед каждым показателем, снятым сервисом `CounterService` или `GaugeService`, стоит префикс `meter.`, то среда Spring Boot Actuator передаст полномочия реализации вычислений и сохранения этого показателя объекту `Meter` библиотеки Dropwizard Metrics.

В документации Dropwizard Metrics (https://metrics.dropwizard.io/3.1.0/manual/core/) сказано: «Объекты `Meter` замеряют интенсивность поступления событий несколькими различными способами. Под интенсивностью понимается средняя интенсивность поступления событий. Обычно данный объект используется для получения общего представления, но поскольку он выдает общий ход событий на протяжении всего жизненного цикла вашего приложения (например, общее количество обработанных запросов, поделенное на количество секунд выполнения процесса), то не дает представления о только что произошедшем. К тому же в показателях записываются и три различные величины экспоненциально-взвешенных скользящих средних: 1-, 5- и 15-минутные скользящие средние».

Поскольку `Meter` задействует экспоненциально-взвешенные скользящие средние, то ему не нужно запоминать *все* фиксируемые значения; он запоминает набор

значений, выбранных по времени. Это приводит к экономии памяти даже при длительных периодах времени. Разберем наш пример с использованием `Meter`, а затем проанализируем эффект записанных показателей. Для этого просто расширим контроллер `CustomerRestController`, рассмотренный в примере 13.3, переопределив метод `metricPrefix`, чтобы поставить перед всеми показателями сервиса `CounterService` префикс `meter.` (примеры 13.4, 13.5).

Пример 13.4. Сбор измеренных показателей с помощью CounterService и библиотеки Dropwizard Metrics

```
package demo.metrics;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.CounterService;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/metered/customers")
public class MeterCustomerRestController extends CustomerRestController {

 @Autowired
 MeterCustomerRestController(CustomerRepository repository,
  CounterService counterService) {
  super(repository, counterService);
 }

 @Override
 protected String metricPrefix(String k) {
  return "meter." + k; ❶
 }
}
```

❶ Установка перед всеми записываемыми показателями префикса `meter`.

Пример 13.5. Показатели на базе Dropwizard Meter

```
{
   "meter.customers.read.fifteenMinuteRate" : 0.102683423806518,
   "meter.customers.read.meanRate" : 0.00164167411117908,
   "meter.customers.read.fiveMinuteRate" : 0.0270670566473226,
   "meter.customers.read.oneMinuteRate" : 9.07998595249702e-06
   ...
}
```

Счетчик и датчик фиксируют одно значение. Показатель фиксирует количество значений за период времени. Показатель не сообщает о частотах значений в наборе данных. *Гистограмма* — статистическое распределение значений в потоке данных: она показывает, сколько раз появлялось конкретное значение, и позволяет отвечать на вопросы наподобие: «Какой процент заказов содержит в корзине более одного товара?» В гистограмме Dropwizard Metrics `Histogram` показывается минимум, максимум, среднее и медиана, а также процентили: 75, 90, 95, 98, 99 и 99,9-й.

Если вы проходили начальный курс статистики, то знаете, что для вывода этих значений с полной точностью необходимы все точки данных. Даже за короткий период времени это может быть колоссальный объем информации. Предположим, ваше приложение зафиксировало 1000 логических транзакций в секунду и в каждом запросе было еще десять запросов или действий. В течение суток получится 864 000 000 значений (24 × 60 × 60 × 1000 × 10)! При использовании Java получается, что тип `long` занимает восемь байт, и за сутки выходит более шести гигабайт данных! Конечно, такой высокий трафик бывает не у всех приложений, но есть и немало программ, у которых он имеется. В любом случае можно рассматривать этот вопрос под разными направлениями масштабирования и в результате прийти к выводу, что в конечном итоге все вполне может вылиться в серьезную проблему.

В библиотеке Dropwizard Metrics для хранения статистически репрезентативных выборок по мере их появления используется *коллектор отбора проб*. По прошествии времени в Dropwizard Histogram создаются выборки прежних значений, которые применяются для создания в будущем новых выборок. Результат неидеальный (из-за потерь), но вполне эффективный. Среда Spring Boot Actuator выполнит автоматическое преобразование любого значения, отправленного с использованием сервиса `GaugeService`, имеющего ключевой префикс показателя `histogram` в Dropwizard Histogram, при условии, что библиотека Dropwizard Metrics указана в пути к классам.

Рассмотрим пример 13.6, в котором фиксируются гистограммы загруженных файлов.

Пример 13.6. Сбор распределения размеров выложенных файлов с использованием реализации гистограммы Dropwizard Metrics

```
package demo.metrics;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.GaugeService;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;

@RestController
@RequestMapping("/histogram/uploads")
public class HistogramFileUploadRestController {

 private final GaugeService gaugeService;

 private Log log = LogFactory.getLog(getClass());

 @Autowired
```

```
 HistogramFileUploadRestController(GaugeService gaugeService) {
  this.gaugeService = gaugeService;
 }

 @RequestMapping(method = RequestMethod.POST)
 void upload(@RequestParam MultipartFile file) {
  long size = file.getSize();
  this.log.info(String.format("received %s with file size %s",
   file.getOriginalFilename(), size));
  this.gaugeService.submit("histogram.file-uploads.size", size);  (1)
 }
}
```

(1) Чтобы получить закулисное преобразование в экземпляры Dropwizard Histogram всех записанных показателей, их нужно снабдить префиксом `histogram.`.

В примере 13.6 создается гистограмма для размеров загруженных файлов. Чтобы выложить три случайно взятых разных по размеру файла (табл. 13.2), мы воспользовались командой `curl`.

Таблица 13.2. Взятые для примера файлы и их размеры

Размер файла	Имя файла	Частота загрузки
8.0K	${HOME}/Desktop/1.png	2
32K	${HOME}/Desktop/2.png	5
40K	${HOME}/Desktop/3.png	3

Конечная точка `/metrics` подтверждает данные, показанные в таблице (пример 13.7).

Пример 13.7. Показатели, полученные с помощью гистограммы Dropwizard

```
{
   ...
   "histogram.file-uploads.size.snapshot.98thPercentile" : 38803,
   "histogram.file-uploads.size.snapshot.999thPercentile" : 38803,
   "histogram.file-uploads.size.snapshot.median" : 29929,
   "histogram.file-uploads.size.snapshot.mean" : 27154.1998413605,
   "histogram.file-uploads.size.snapshot.75thPercentile" : 38803,
   "histogram.file-uploads.size.snapshot.min" : 6347,
   "histogram.file-uploads.size.snapshot.max" : 38803,
   "histogram.file-uploads.size.count" : 10,
   "histogram.file-uploads.size.snapshot.95thPercentile" : 38803,
   "histogram.file-uploads.size.snapshot.99thPercentile" : 38803,
   "histogram.file-uploads.size.snapshot.stdDev" : 11639.4103925448,
   ...
}
```

В библиотеке Dropwizard также поддерживаются таймеры. С помощью *таймера* измеряется интенсивность вызова конкретного фрагмента кода и распределение

его продолжительности. Таймер отвечает на вопрос: «Сколько *обычно* тратится времени на данный вид запроса и что собой представляют нетипичные продолжительности?» Среда Spring Boot Actuator выполнит автоматическое преобразование любых показателей, начинающихся с префикса `timer.`, которые отправлены в `Timer` с использованием сервиса `GaugeService`.

Запросы времени можно выполнять с помощью различных механизмов. Среда Spring поставляется с весьма достойным классом `StopWatch`, который вполне подходит для наших целей (пример 13.8).

Пример 13.8. Получение показателей, связанных с временем, используя Spring StopWatch

```
package demo.metrics;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.GaugeService;
import org.springframework.http.ResponseEntity;
import org.springframework.util.StopWatch;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class TimedRestController {

 private final GaugeService gaugeService;

 @Autowired
 public TimedRestController(GaugeService gaugeService) {
  this.gaugeService = gaugeService;
 }

 @RequestMapping(method = RequestMethod.GET, value = "/timer/hello")
 ResponseEntity<?> hello() throws Exception {
  StopWatch sw = new StopWatch();  (1)
  sw.start();
  try {
   Thread.sleep((long) (Math.random() * 60) * 1000);
   return ResponseEntity.ok("Hi, " + System.currentTimeMillis());
  }
  finally {
   sw.stop();
   this.gaugeService.submit("timer.hello", sw.getLastTaskTimeMillis());
  }
 }
}
```

(1) Это класс `StopWatch` среды Spring, который используется здесь для замера продолжительности запроса.

Таймер дает все то же самое, что и гистограмма, и плюс данные, относящиеся конкретно к продолжительности (пример 13.9).

Пример 13.9. Результаты использования Timer в выводе показателей

```
{
   ...
   "timer.hello.snapshot.stdDev" : 11804,
   "counter.status.200.timer.hello" : 7,
   "timer.hello.snapshot.75thPercentile" : 35004,
   "timer.hello.meanRate" : 0.0561559793104086,
   "timer.hello.snapshot.mean" : 27639,
   "timer.hello.snapshot.min" : 2007,
   "timer.hello.snapshot.max" : 42003,
   "timer.hello.snapshot.median" : 35004,
   "timer.hello.snapshot.98thPercentile" : 42003,
   "timer.hello.fifteenMinuteRate" : 0.182311662062598,
   "timer.hello.snapshot.99thPercentile" : 42003,
   "timer.hello.snapshot.999thPercentile" : 42003,
   "timer.hello.oneMinuteRate" : 0.0741487724647533,
   "timer.hello.fiveMinuteRate" : 0.153231174431025,
   "timer.hello.count" : 7,
   "gauge.response.timer.hello" : 28008,
   "timer.hello.snapshot.95thPercentile" : 42003
...
}
```

Библиотека Dropwizard Metrics обогащает подсистему показателей Spring Boot. Она дает нам доступ к сбору статистики, чего в противном случае у нас просто не было бы.

Взаимосвязанные представления показателей

До сих пор все наши примеры запускались на одном узле или хосте, по одному экземпляру на хосте. По мере разрастания масштабов важное значение будет приобретать централизация показателей со всех сервисов и экземпляров. Для централизованного сбора и хранения показателей можно воспользоваться *базой данных временны́х рядов* (time series database, TSDB) (https://www.multitran.ru/c/m.exe?t=5395412_2_1&s1=time%20series%20database). Она оптимизирована под сбор, анализ, а иногда и под визуализацию показателей с течением времени. Существует множество популярных TSDB, среди которых Ganglia (http://ganglia.info/), Graphite (https://github.com/graphite-project/), OpenTSDB (http://opentsdb.net/), InfluxDB (https://www.influxdata.com/) и Prometheus (https://prometheus.io/). Такая база хранит значения для заданного ключа за определенный период времени. Обычно эти базы работают в тандеме с чем-то, что поддерживает графическое отображение хранящейся в них информации. Есть множество превосходных технологий для графического отображения данных временны́х рядов, наиболее популярной из

которых представляется Grafana (https://grafana.com/). Существуют и альтернативные технологии визуализации.

Многие компании оставляют код своих инструментов визуализации открытым, например: Vimeo's GraphExplorer (https://github.com/vimeo/graph-explorer), TicketMaster's Metrilyx (https://tech.ticketmaster.com/2014/08/14/announcing-metrilyx/) и Square's Cubism.js (http://square.github.io/cubism/).

Среда Spring Boot поддерживает запись показателей в базу данных временны́х рядов с помощью реализаций интерфейса MetricsWriter. В готовом виде Spring Boot может публиковать показатели в экземпляр Redis, JMX (что, возможно, более подходит для разработки) за пределами имеющегося в среде Spring `MessageChannel` или в любой сервис, выполняющий обмен данными по протоколу StatsD (https://github.com/etsy/statsd/wiki). Последний изначально был спроектирован на основе Node.js специалистами компании Etsy для действия в качестве прокси-сервера для Graphite/Carbon, но стал настолько популярен, что теперь многие клиенты и сервисы ведут обмен данными именно по этому протоколу, а сам StatsD поддерживает, кроме Graphite, множество реализаций серверной части, включая InfluxDB (https://www.influxdata.com/), OpenTSDB (http://opentsdb.net/) и Ganglia (http://ganglia.info/). Чтобы воспользоваться различными реализациями `MetricWriter`, нужно просто определить bean-компонент соответствующего типа и снабдить его аннотацией `@ExportMetricWriter`. Если же StatsD в вашу практику не вписывается или вам нужен более жесткий контроль, то Dropwizard также поддерживает публикацию показателей в нижестоящие системы с помощью своих реализаций `*Reporter`.

Размерности показателей

Данные, создаваемые в TSDB, — всего лишь данные, даже если их размерности будут казаться очень ограниченными: у показателя, по крайней мере, есть ключ и значение. Проблема в том, что этого очень мало для составления *схемы*. В различных TSDB предоставляются усовершенствования и предлагаются иные размерности показателей. Некоторые базы предлагают ввести понятие *меток* или *тегов*. Но единственная размерность, общая для всех TSDB, — ключ. Во многих встречавшихся нам реализациях использовались иерархические ключи (например, `a.b.c.`). Многие TSDB поддерживают запросы по стандартной маске (например, `a.b.*`), которые станут возвращать все показатели, соответствующие префиксу `a.b`. В каждом компоненте пути, имеющемся в ключе, должна быть четко выраженная цель, и изменяемые компоненты пути должны содержаться в иерархии как можно глубже. Ключи и показатели следует разрабатывать с прицелом на поддержку несомненного роста количества показателей. Нужно весьма тщательно продумывать то, что фиксируется в показателях, либо с помощью иерархических ключей, либо с помощью иных размерностей, подобных меткам или тегам.

А как закодировать запросы относительно разных товаров в одной и той же TSDB? Зафиксируйте имя компонента, например `order-service`.

Как будут кодироваться различные типы деятельности или процессов? HTTP-запросы, вспомогательные запросы на основе рассылки сообщений, вспомогательные пакетные задания или что-либо еще? В каком виде это можно закодировать: `order-service.tasks.fulfillment.validate-shipping` или `order-service.requests.new-order`?

Подумайте, как будет кодироваться информация, с тем чтобы ее в перспективе можно было сопоставить с системами, направленными на управление разработками, такими как Vertica компании HP (https://www.vertica.com/) или Scuba компании Facebook? Нам хотелось бы заявить, что *вся* оперативная телеметрия непосредственно преобразуется в бизнес-показатели, но это неправда. И тем не менее было бы полезно иметь способ сбора данной информации и ее подключения. Это можно сделать непосредственно в самих ключах показателей или, вероятно, с помощью меток и тегов.

Как вы будете сопоставлять запросы на тесты A/B (или эксперименты), где выборка населения происходит через пути кода для заданной функции, ведущей себя не так, как большинство запросов? Это помогает оценить, насколько хорошо работает и воспринимается данная функция. Здесь подразумевается возможное наличие у вас сходных показателей в двух различных путях кода, один из которых является экспериментальной альтернативой по отношению ко второму. Системы для экспериментов и номера уровней экспериментов, которые должны быть вставлены в показатель, имеются во многих организациях.

Как всегда, схема субъективна, а новые TSDB различаются по изобилию и масштабу собранных данных! Одни из них допускают частичную утрату информации, а другие могут выполнять почти бесконечное горизонтальное масштабирование. К выбору своей TSDB, как и к выбору любых других баз данных, следует относиться с особой тщательностью, рассматривая возможности для проектирования схемы. С одной стороны, между разработчиками при выборе показателей не должно быть никаких трений, с другой — прозрение может наступить намного позже.

Показатели доставки из приложения Spring Boot

Найти размещенные версии многих таких баз данных временны́х рядов не составляет труда. Одной из них, легко поддающейся интеграции, является Hosted Graphite (https://www.hostedgraphite.com/). В ней предварительно конфигурируются Graphite, Graphite Composer и Grafana. Две последние позволяют создавать графики на основе собранных показателей. Можно, конечно же, запустить собственный экземпляр Graphite, но при этом следует иметь в виду, что целью, как всегда, является получение конечного результата как можно быстрее. Поэтому мы склоняемся к применению сервисов на основе облачных технологий (программ в виде сервисов — Software as a Service, SaaS); нам не нравится запускать ПО, если его нельзя продать. Есть несколько вполне подходящих вариантов подключения

к Graphite, доступных разработчику программ в среде Spring Boot. Можно воспользоваться Spring Boot-компонентом `StatsdMetricWriter`, который понимает протокол StatsD и работает со множеством вышеупомянутых серверных приложений. Если же вы, к примеру, помимо StatsD, предпочитаете применять собственный протокол, то можете задействовать огромное количество обычных реализаций генераторов отчетов Dropwizard Metrics. Чтобы организовать обмен данными с Graphite в исходном формате, мы, как показано в примере 13.10, именно этим здесь и займемся.

Пример 13.10. Конфигурирование экземпляра Dropwizard Metrics GraphiteReporter

```
package demo.metrics;

import com.codahale.metrics.MetricRegistry;
import com.codahale.metrics.graphite.Graphite;
import com.codahale.metrics.graphite.GraphiteReporter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.annotation.PostConstruct;
import java.util.concurrent.TimeUnit;

@Configuration
class ActuatorConfiguration {

 ActuatorConfiguration() {
  java.security.Security.setProperty("networkaddress.cache.ttl", "60");  ❶
 }

 @Bean
 GraphiteReporter graphiteWriter(
  @Value("${hostedGraphite.apiKey}") String apiKey,
  @Value("${hostedGraphite.url}") String host,
  @Value("${hostedGraphite.port}") int port, MetricRegistry registry) {

  GraphiteReporter reporter = GraphiteReporter.forRegistry(registry)
   .prefixedWith(apiKey)  ❷
   .build(new Graphite(host, port));
  reporter.start(1, TimeUnit.SECONDS);
  return reporter;
 }

}
```

❶ Предотвращение DNS-кэширования, связанное с возможностью перемещения узлов `HostedGraphite.com` и их отображения на новый DNS-маршрут.

❷ Сервис HostedGraphite.com (https://www.hostedgraphite.com/) вводит аутентификацию, используя API-ключ, и ожидает его отправки в качестве части поля `prefix`. Следует признать, что этот прием не слишком привлекателен, но другого очевидного места для помещения информации по аутентификации просто нет!

Направьте трафик к `CustomerRestController` или `MeterCustomerRestController` — и увидите отображение этого трафика на графике, который можно создать на HostedGraphite.com, в интерфейсе Grafana либо в интерфейсе-компоновщике Graphite, чьи изображения показаны на рис. 13.1 и 13.2.

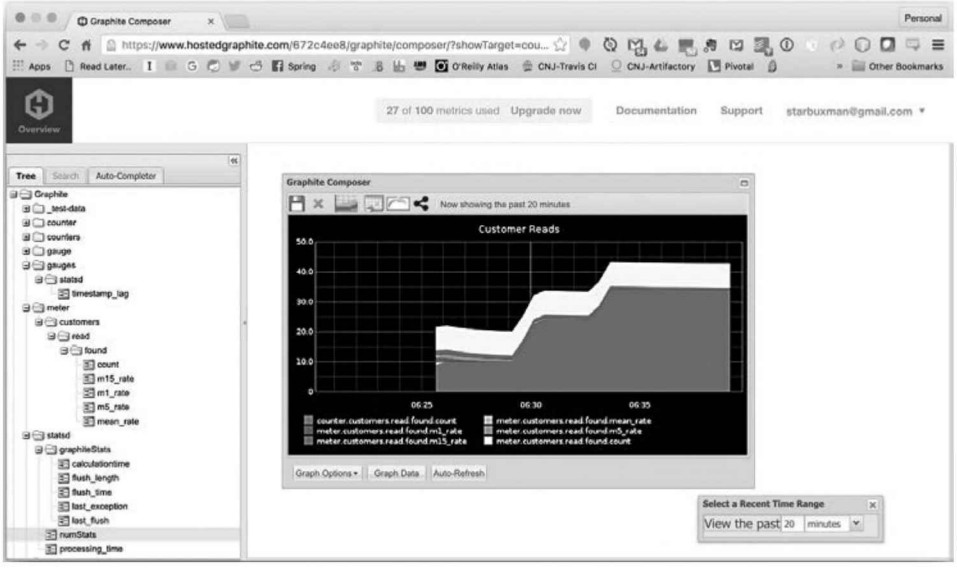


Рис. 13.1. Информационная панель Graphite Composer

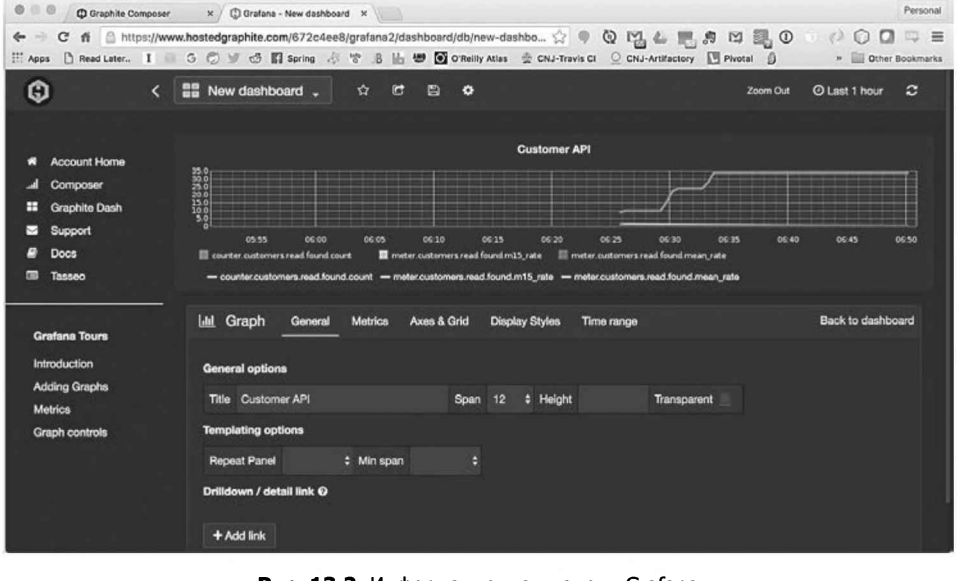


Рис. 13.2. Информационная панель Grafana

Идентификация вашего сервиса с помощью конечной точки /info

В идеале разработка кода ведется в процессе непрерывной доставки. В данном процессе каждое подтверждение изменения кода может приводить к продвижению в производство. В идеальном случае это происходит много раз за день. При сбоях в работе в первую очередь хотелось бы узнать, какая версия кода запущена. Вашему сервису нужно придать возможность самоидентификации с помощью конечной точки `/info`.

Эта конечная точка намеренно оставлена пустой. Она является вполне естественным местом для помещения информации о самом сервисе. Как называется сервис? Какое подтверждение изменения кода в git-системе запустит сборку, которая в конечном итоге приведет к продвижению в производство? Какова версия сервиса?

Пользовательские свойства можно предоставить через обычные каналы (`application.properties`, `application.yml` и т. д.), указав перед этими свойствами префикс `info.`. Можно также собрать информацию о состоянии git-хранилища исходного кода относительно времени сборки проекта, добавив дополнительный модуль Maven путем указания строки `pl.project13.maven : git-commit-id-plugin`. Этот модуль предварительно сконфигурирован для пользователей среды Maven в родительской Maven-сборке в Spring Boot. Он создает файл `git.properties`, содержащий свойства `git.branch` и `git.commit`. Конечная точка `/info` будет знать о необходимости поиска этого файла при условии его доступности (пример 13.11).

Пример 13.11. Ветвь Git-хранилища, идентификатор подтверждения изменения кода и время, предоставленные конечной точкой /info

```
{
  git: {
    branch: "master",
    commit: {
     id: "407359e",
     time: "2016-03-23T00:47:09+0100"
    }
  }
  ...
}
```

Добавить пользовательские свойства довольно просто. Любое свойство среды окружения, снабженное префиксом `info.`, будет добавлено к выходным данным этой конечной точки. К примеру, имеющаяся в Spring Boot исходная конфигурация модуля Maven уже настроена на фильтрацию ресурсов среды Maven. Фильтрацию ресурсов Maven можно применить для выдачи пользовательских свойств, зафиксированных на момент сборки, таких как имеющиеся в среде Maven `project.artifactId` и `project.version` (пример 13.12).

Пример 13.12. Фиксация пользовательской информации на время сборки, то есть artifactId и версии проекта, с помощью конечной точки /info за счет предоставления свойств в ходе сборки с применением фильтрации ресурсов среды Maven

```
info.project.version=@project.version@
info.project.artifactId=@project.artifactId@
```

Как только это будет сделано, обратитесь к своей конечной точке `/info` и выясните, что произошло (пример 13.13).

Пример 13.13. Фиксация пользовательской информации на время сборки, то есть получение artifactId и версии проекта с помощью конечной точки /info

```
{
  ...
  project: {
    artifactId: "actuator",
    version: "1.0.0-SNAPSHOT"
  }
}
```

Проверки работоспособности

Приложениям нужен способ инициативной проверки своей работоспособности в той или иной инфраструктуре. Качественная проверка показывает совокупное состояние, в котором суммируются состояния, представленные в отчетах об отдельных задействованных компонентах. Проверки работоспособности часто используют балансировщики нагрузки для определения жизнеспособности того или иного узла. Балансировщики могут исключать узлы на основе возвращенного кода состояния HTTP. В конечной точке `org.springframework.boot.actuate.endpoint.HealthEndpoint` происходит сбор показаний всех реализаций индикаторов работоспособности `org.springframework.boot.actuate.health.HealthIndicator` в контексте приложения и их предоставление. В примере 13.14 показаны выходные данные исходной конечной точки `/health` в нашем приложении-образце.

Пример 13.14. Выходные данные исходной конечной точки /health для нашего приложения-образца

```
{
  status: "UP",
  diskSpace: {
    status: "UP",
    total: 999334871040,
    free: 735556071424,
    threshold: 10485760
  },
  redis: {
    status: "UP",
```

```
    version: "3.0.7"
  },
  db: {
    status: "UP",
    database: "H2",
    hello: 1
  }
}
```

В среде Spring Boot на основе различных автоматических заданий конфигурации для JavaMail, MongoDB, Cassandra, JDBC, SOLR, Redis, ElasticSearch, файловой системы и др. выполняется автоматическая регистрация обычных реализаций индикаторов работоспособности `HealthIndicator`. Эти индикаторы относятся ко всему тому, что при взаимодействии с вашим сервисом может дать сбой независимо от него самого. В показанном выше примере для вас уже автоматически учтены Redis, файловая система и наш источник данных JDBC `DataSource`.

Предоставим пользовательский индикатор работоспособности `HealthIndicator`. Соглашение для него будет простым: при запросе будет возвращаться экземпляр `Health` с соответствующим состоянием. Другие присутствующие в системе компоненты должны иметь возможность оказывать влияние на возвращаемый объект `Health`. Внедрить соответствующий `HealthIndicator` можно напрямую, после чего он сможет управлять своим состоянием в каждом компоненте, способном воздействовать на это состояние. Однако данная комбинация приведет к привязке весьма существенного объема кода приложения к решению вторичного вопроса, который заключается в выводе состояния работоспособности. В качестве альтернативы можно применить имеющуюся в среде Spring шину событий `ApplicationContext`, чтобы публиковать события в компонентах и управлять `HealthIndicator` на основе уведомительных событий.

В следующих блоках кода мы введем эмоциональный индикатор работоспособности (пример 13.15), который будет рад (`UP`) или опечален (`DOWN`) при получении события `SadEvent` (пример 13.16) или события `HappyEvent` (пример 13.17) соответственно.

Пример 13.15. Эмоциональный HealthIndicator

```
package demo.health;

import org.springframework.boot.actuate.health.AbstractHealthIndicator;
import org.springframework.boot.actuate.health.Health;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.Optional;

@Component
```

```
class EmotionalHealthIndicator extends AbstractHealthIndicator {

 private EmotionalEvent event;

 private Date when;

 ❶
 @EventListener
 public void onHealthEvent(EmotionalEvent event) {
  this.event = event;
  this.when = new Date();
 }

 ❷
 @Override
 protected void doHealthCheck(Health.Builder builder) throws Exception {
 //@formatter:off
  Optional
    .ofNullable(this.event)
    .ifPresent(
     evt -> {
      Class<? extends EmotionalEvent> eventClass = this.event.getClass();
      Health.Builder healthBuilder = eventClass
              .isAssignableFrom(SadEvent.class) ? builder
       .down() : builder.up();
      String eventTimeAsString = this.when.toInstant().toString();
      healthBuilder.withDetail("class", eventClass).withDetail("when",
       eventTimeAsString);
      });
 //@formatter:off
 }

}
```

❶ Мы подключим этот метод-отслеживатель к событиям `ApplicationContext`, используя аннотацию `@EventListener`.

❷ Метод `doHealthCheck` использует `Health.Builder` для переключения состояния индикатора работоспособности на основе последних знаний, записанных `EmotionalEvent`.

Пример 13.16. SadEvent

```
package demo.health;

public class SadEvent extends EmotionalEvent {
}
```

Пример 13.17. HappyEvent

```
package demo.health;

public class HappyEvent extends EmotionalEvent {
}
```

Теперь любому компоненту в `ApplicationContext` нужно лишь опубликовать подходящее событие, чтобы инициировать изменения соответствующего состояния (пример 13.18).

Пример 13.18. Эмоциональная конечная точка REST

```
package demo.health;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
class EmotionalRestController {

 private final ApplicationEventPublisher publisher; ❶

 @Autowired
 EmotionalRestController(ApplicationEventPublisher publisher) {
  this.publisher = publisher;
 }

 @RequestMapping("/event/happy")
 void eventHappy() {
  this.publisher.publishEvent(new HappyEvent()); ❷
 }

 @RequestMapping("/event/sad")
 void eventSad() {
  this.publisher.publishEvent(new SadEvent());
 }
}
```

❶ Среда Spring автоматически предоставляет реализацию интерфейса `ApplicationEventPublisher` для использования в коде компонента. Более того, вполне возможно, что предоставляемый этой средой контекст приложения `ApplicationContextSpring`, который используется для запуска вашего приложения, уже *является* интерфейсом `ApplicationEventPublisher`.

❷ Оттуда обычно выполняется диспетчеризация события между компонентами.

События упрощают поддержку распространения оперативной информации, не замедляя прохождение запроса бизнес-логики, для которой, к примеру, совершенно неинтересна конечная точка, связанная с проверкой работоспособности.

Контрольные события

События представляют собой великолепный способ фиксации практически всего происходящего в системе. С помощью контрольных событий среда Spring Boots поддерживает аудит, при котором события в приложении привязываются

к инициировавшим их аутентифицированным пользователям. Рассмотрим конечную точку REST, у которой в пути к классам тоже имеется Spring Security (`org.springframework.boot : spring-boot-starter-security`). Чтобы заставить работать обычную демонстрацию, мы сконфигурировали пользовательскую реализацию `UserDetailsService` (пример 13.19).

Пример 13.19. Несколько конкретно заданных пользователей

```
package com.example;

import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import java.util.Arrays;
import java.util.Optional;
import java.util.Set;
import java.util.concurrent.ConcurrentSkipListSet;

@Service
class SimpleUserDetailsService implements UserDetailsService {

 private final Set<String> users = new ConcurrentSkipListSet<>();

 SimpleUserDetailsService() {
  ❶
  this.users.addAll(Arrays.asList("pwebb", "dsyer", "mbhave", "snicoll",
   "awilkinson"));
 }

 @Override
 public UserDetails loadUserByUsername(String s)
  throws UsernameNotFoundException {
  ❷
  return Optional
   .ofNullable(this.users.contains(s) ? s : null)
   .map(x -> new User(x, "pw",
        AuthorityUtils.createAuthorityList("ROLE_USER")))
   .orElseThrow(() -> new UsernameNotFoundException("couldn't find " +
                s + "!"));
 }
}
```

❶ Мы конкретно задали список пользователей (`dsyer`, `pwebb` и т. д.) и...

❷ ...паролей (`pw` для каждого пользователя, но не вздумайте пробовать этот же прием на практике!) с фиксированной ролью (`ROLE_USER`).

По умолчанию автоконфигурирование среды Spring Boot блокирует конечные точки HTTP с помощью аутентификации HTTP BASIC. Среда Spring Security

выдаст события, относящиеся к аутентификации и авторизации: пройдена ли кем-либо аутентификация (или же попытка была безуспешной), вышел ли кто-либо из системы и т. д. Можно также создать собственные контрольные события. Рассмотрим обычный пример конечной точки HTTP, зависящей от факта предоставления средой Spring Security уже аутентифицированному принципалу `java.security.Principal` возможности ввода в методы-обработчики Spring MVC (пример 13.20).

Пример 13.20. Обычная (но безопасная) конечная точка HTTP

```
package com.example;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.audit.AuditEvent;
import org.springframework.boot.actuate.audit.AuditEventRepository;
import org.springframework.boot.actuate.audit.listener.AuditApplicationEvent;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.security.Principal;
import java.util.Collections;

@RestController
class GreetingsRestController {

 public static final String GREETING_EVENT = "greeting_event".toUpperCase();

 private final ApplicationEventPublisher appEventPublisher;

 @Autowired
 GreetingsRestController(ApplicationEventPublisher appEventPublisher) {
  this.appEventPublisher = appEventPublisher;
 }

 @GetMapping("/hi")
 String greet(Principal p) {  ❶
  String msg = "hello, " + p.getName() + "!";

  AuditEvent auditEvent = new AuditEvent(p.getName(),  ❷
   GREETING_EVENT,  ❸
   Collections.singletonMap("greeting", msg));  ❹

  this.appEventPublisher.publishEvent(  ❺
   new AuditApplicationEvent(auditEvent));

  return msg;
 }

}
```

1. Ввод уже аутентифицированного принципала `Principal` и...
2. ...его использование для создания контрольного события `AuditEvent`, выполнения разыменования аутентифицированного принципала `Principal`...
3. ...наряду с указанием имени события (в данном случае произвольным, но для вашей системы нужно задействовать какое-нибудь значимое имя) и...
4. ...любых дополнительных метаданных, которые вам хотелось бы включить в регистрационную запись.
5. И наконец, применение механизма события контекста Spring-приложения для диспетчеризации `AuditEvent` с объектом-адаптером `AuditApplicationEvent`.

Вызвать защищенную конечную точку можно после аутентификации с помощью HTTP BASIC. Мы воспользуемся подходящим клиентом `httpie`, ну а вы можете применить, что захотите (пример 13.21).

Пример 13.21. Выходные данные актуатора Actuator конечной точки HTTP /auditevents

```
{
  "events" : [
     {
        "timestamp" : "2017-04-26T14:01:10+0000",
        "principal" : "dsyer",
        "type" : "AUTHENTICATION_SUCCESS",
        "data" : {
           "details" : {
              "sessionId" : null,
              "remoteAddress" : "127.0.0.1"
           }
        }
     },
     {
        "timestamp" : "2017-04-26T14:01:10+0000",
        "data" : {
           "greeting" : "hello, dsyer!"
        },
        "type" : "GREETING_EVENT",
        "principal" : "dsyer"
     }
  ]
}
```

Механизм контрольных событий упрощает получение информации о пользователях, вошедших в систему. В среде Spring Boot имеется компонент `AuditListener`, который занимается отслеживанием событий и их записью с помощью реализации хранилища `AuditEventRepository` (пример 13.22). Изначально оно находится в оперативной памяти, но его несложно будет реализовать, задействовав какую-либо постоянную внешнюю память. Предоставьте собственную реализацию, и среда Spring Boot не преминет ею воспользоваться взамен исходного хранилища.

Пример 13.22. Простой отслеживатель AuditApplicationEvent

```
package com.example;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.actuate.audit.AuditEvent;
import org.springframework.boot.actuate.audit.listener.AbstractAuditListener;
import org.springframework.boot.actuate.audit.listener.AuditApplicationEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
class SimpleAuditEventListener {

 private Log log = LogFactory.getLog(getClass());

 @EventListener(AuditApplicationEvent.class)
 public void onAuditEvent(AuditApplicationEvent event) {
  this.log.info("audit-event: " + event.toString());
 }
}
```

Можно также отслеживать контрольные события (и реагировать на них) в собственном коде, используя для этого точно такие же механизмы отслеживания, как и для любых других событий среды Spring.

Ведение журнала приложения

Мы уже готовы встретить будущее! У нас уже есть машины без водителей и дома, разговаривающие с нами. Мы, моргнув глазом, можем запустить тысячи серверов! И тем не менее почтенный регистрационный файл все еще остается одним из лучших имеющихся у нас способов, позволяющих разобраться в поведении того или иного узла или системы. В регистрационных записях отображается ритм процесса, его активность. Ведение журнала требует добавления в код внедряемых в него инструкций, но сами регистрационные файлы являются одним из самых обособленных инструментов. Извлекать полезные данные из регистрационных записей призваны целые специально разработанные экосистемы, инструментальные наборы, языки и платформы больших данных.

При ведении журнала принимаются два решения.

- *О выводе информации из журнала.* Где будут появляться данные, выводимые из журнала? В файле? В консоли? В сервисе `SyslogD`?
- *Об уровнях регистрирования.* Какая степень детализации вывода вам необходима? Нужно ли выводить на экран каждую малейшую подвижку или же этого должны удостаиваться только угрозы мирового масштаба?

Определение характера выходных регистрационных данных

По умолчанию регистрационные записи появляются в консоли приложения Spring Boot. Дополнительно можно настроить вывод в файл или какой-либо другой объект накопления регистрационных данных, но по умолчанию целесообразнее всего выводить их в консоль.

В процессе разработки вам понадобится изучать регистрационные записи по мере их появления, а если приложение запускается в облачной среде, то не нужно проявлять заботу о том, куда будут направлены регистрационные данные. Решение этого вопроса сообразуется с одним из принципов манифеста 12-факторного приложения (https://12factor.net/logs):

> На 12-факторное приложение никогда не возлагается задача направления или сохранения его выходных потоков данных. Оно не должно предпринимать попыток ведения записей в регистрационные файлы или управления этими файлами. Каждый выполняемый процесс записывает свой поток событий без всякой буферизации на стандартное устройство вывода (stdout). В ходе локальной разработки занимающийся этим специалист будет просматривать данный поток на видном месте своего терминала, что позволит ему отслеживать поведение приложения.

Сборщики или концентраторы регистрационных записей наподобие имеющихся в среде Cloud Foundry Loggregator (https://github.com/cloudfoundry/loggregator) или Logstash получают создаваемые регистрационные записи из разрозненных процессов и объединяют их в единый поток, вероятно направляя его далее в какое-либо место, где его можно проанализировать. Чтобы содействовать анализу, регистрируемые данные должны быть максимально структурированы с помощью соответствующих разделителей, определения групп и поддержки чего-то наподобие логической *схемы*. Регистрационные записи должны рассматриваться в качестве потоков событий. Они раскрывают историю поведения системы. Регистрируемая информация может быть *выводом* одного процесса и *вводом* другого, нижестоящего процесса, который выполняет аналитическую работу. Один из весьма популярных концентраторов Logstash предоставляет множество дополнительных модулей, позволяющих создавать каналы регистрации из нескольких входных источников, и подключать получаемые регистрационные данные к центральной аналитической системе, подобной Elasticsearch, являющейся полнотекстовым поисковым механизмом, поддерживаемым библиотекой Lucene.

Еще один концентратор регистрационных данных, Loggregator, собирает и переправляет их в вашу консоль, используя команду `cf logs $YOUR_APP_NAME`, или на любой сервис, совместимый с протоколом SyslogD, включая предустановленные или размещенные сервисы, такие как ElasticSearch (через Logstash), Papertrail (https://papertrailapp.com/), Splunk (https://www.splunk.com/), Splunk Storm, SumoLogic

или, конечно же, сам SyslogD (http://www.gnu.org/software/libc/manual/html_node/Overview-of-Syslog.html). Настройте направление регистрируемых данных, как и любой предоставляемый пользователем сервис, указав ключ `-l` для обозначения того, что это будет направление данных, помещаемых в журнал (пример 13.23).

Пример 13.23. Создание сервиса, предоставляемого пользователем

```
cf cups my-logs -l syslog://logs.papertrailapp.com:PORT
```

Теперь это просто сервис, доступный, как показано в примере 13.24, для привязки любого приложения.

Пример 13.24. Привязка сервиса к приложению

```
cf bind-service my-app my-logs && cf restart my-app
```

Кроме того, Loggregator публикует регистрационные сообщения по протоколу websocket, поэтому организовать программное *отслеживание* регистрационных данных, приходящих от любых приложений Cloud Foundry, совсем не трудно. Мы используем Java и простую интеграцию канала websocket с Java-клиентом Cloud Foundry.

Среда Cloud Foundry компании Pivotal также предоставляет *взаимосвязанные журнальные записи* (как показано на рис. 13.3), то есть отображает показатели на шкале времени, касающиеся запросов, а затем показывает ваши регистрационные записи за интересующий период времени, отображаемый на шкале.

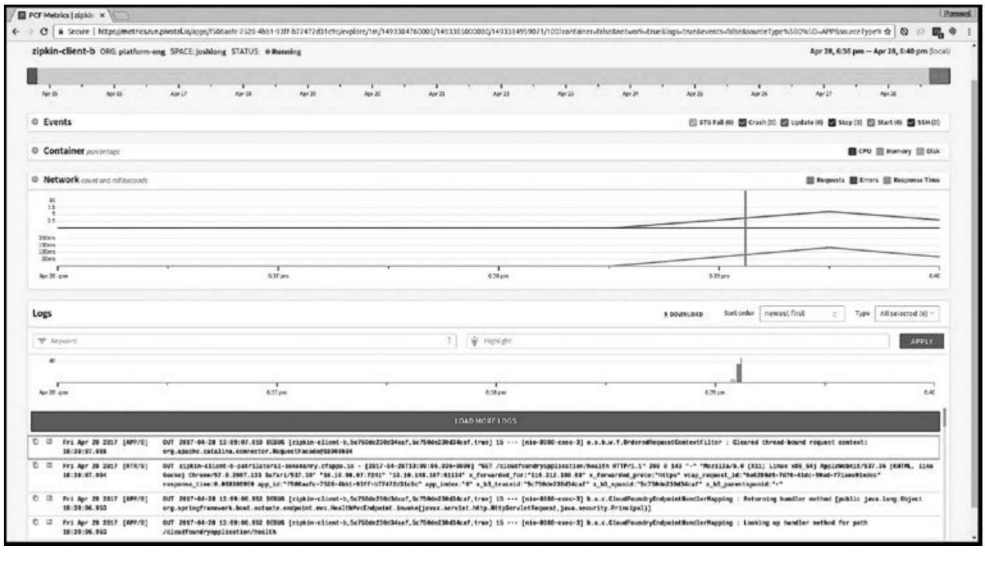


Рис. 13.3. Взаимосвязанные регистрационные записи в средстве AppsManager компании Pivotal

Определение уровней регистрации

Регистрационные журналы стали настолько естественным расширением состояния приложений, что даже приступить к выбору применяемой технологии ведения регистрационных записей весьма непросто. При использовании Spring Boot, наверное, вполне пригодны исходные установки. Для ведения всех внутренних регистрационных записей в среде Spring Boot служит Commons Logging, но реализация базового регистрационного журнала остается открытой. Исходные настройки предоставляются для поддержки ведения журнала средств разработки Java-программ (JDK), Log4J, Log4J2 и Logback. В каждом случае у регистраторов есть предварительная настройка на использование консольного вывода, но наряду с этим имеется и дополнительный вывод в файл.

По умолчанию в среде Spring Boot будет использоваться средство Logback, которое, в свою очередь, может фиксировать и переправлять регистрационные записи, полученные благодаря применению других технологий ведения журналов, таких как Apache Commons Logging, Log4j, Log4J2 и т. д. О чем это говорит? Все вероятные зависимости, указанные в путях к классам и связанные с созданием регистрационных записей, будут вполне успешно работать без всяких переделок, а их выход станет попадать в консоль во вполне узнаваемом настроенном формате, включающем дату и время, уровень регистрации, идентификатор процесса, имя потока, название регистратора и само регистрируемое сообщение. При просмотре регистрационных записей в консоли будут даже включены цветовые коды!

Среду Spring Boot можно использовать в целях управления уровнями регистрации, для чего в общем случае нужно будет указать уровни в среде окружения Spring (как свойство в `application.properties` или в вашем экземпляре Spring Cloud Config Server). Среда Spring разбирается в следующих уровнях регистрации, которые она будет соответственно отображать на используемого поставщика регистратора: `ERROR`, `WARN`, `INFO`, `DEBUG` или `TRACE`. Чтобы заставить весь вывод замолчать, можно также указать уровень `OFF`. Уровни регистрации выстраиваются по приоритету; потенциально и неизменно опасные сообщения, зарегистрированные на уровне `ERROR`, куда более важны, чем предназначенные для разработочных целей утверждения на уровне `DEBUG`. Если для журнала установить уровень `ERROR`, то ничего из имеющегося пакета, за исключением сообщений с уровнем `ERROR`, видно не будет. После установки для журнала уровня `TRACE` станут отображаться все регистрационные сообщения пакета. Вывод определяется с указанного уровня и с любого уровня, находящегося под ним. Рассмотрим пример.

У уровней регистрационных записей есть иерархия: если для пакета `a` установлен уровень `WARN`, то он же будет установлен также для пакета `a.b`. Между настроенными уровнями регистрационных сообщений и реальными уровнями прослеживается разница. В примере 13.25 показана конфигурация для изменения уровня регистрационных записей на `ERROR` для всего кода в пакете `demo`.

Пример 13.25. Указание произвольного уровня регистрационных записей в среде окружения Spring

```
logging.level.demo=error
```

Если запустить приложение, то можно увидеть, что даже при многократной выдаче одного и того же сообщения на различных уровнях регистрационных записей в консоли будет выведено только одно сообщение. Попробуйте это на практике, сделав HTTP-запрос `GET` по адресу http://localhost:8080/log, а затем измените уровень регистрационных записей на `TRACE` и перезапустите приложение. Воспользовавшись кодом примера 13.26, вы увидите одно и то же сообщение, зарегистрированное несколько раз, причем в разных цветах, если, конечно, ваш терминал поддерживает выделения цветом.

Пример 13.26. Приложение на Java, регистрирующее три сообщения с разными уровнями регистрационных записей

```
package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import javax.annotation.PostConstruct;
import java.util.Optional;

@SpringBootApplication
@RestController
public class LoggingApplication {

 private Log log = LogFactory.getLog(getClass());

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

 LoggingApplication() {
  triggerLog(Optional.empty());
 }

 @GetMapping("/log")
 public void triggerLog(@RequestParam Optional<String> name) {
  String greeting = "Hello, " + name.orElse("World") + "!";
  this.log.warn("WARN: " + greeting);  ❶
```

```
  this.log.info("INFO: " + greeting);
  this.log.debug("DEBUG: " + greeting);
  this.log.error("ERROR: " + greeting);
 }
}
```

❶ В зависимости от указанного уровня регистрационных записей сообщения вообще не появятся, появятся одно сообщение или все зарегистрированные.

Итак, чтобы увидеть обновленные уровни регистрационных сообщений, мы перезапустили процесс, но можно выполнить запрос и в *динамическом режиме* в ходе процесса, переконфигурировав уровни регистрационных записей с помощью конечной точки `/loggers` среды Spring Boot Actuator. Если воспользоваться HTTP-запросом `GET`, то конечная точка покажет все уровни регистрационных записей, сконфигурированные в нашем приложении (пример 13.27).

Пример 13.27. Перечисление всех уровней регистрационных записей

```
{
   "loggers" : {
      ... ❶
      "org.springframework.boot.actuate.endpoint" : {
         "effectiveLevel" : "INFO",
         "configuredLevel" : null
      },
      "demo" : {
         "effectiveLevel" : "ERROR",
         "configuredLevel" : "ERROR"
      }
   },
   "levels" : [
      "OFF",
      "ERROR",
      "WARN",
      "INFO",
      "DEBUG",
      "TRACE"
   ]
}
```

❶ Здесь открыто только несколько строк из *тысяч* других строк конфигурации!

Подробности, касающиеся указанного регистратора, можно получить с помощью обращения к конечной точке `/loggers/{logger}`, где `{logger}` — это имя вашего пакета или имя в иерархии регистрационных записей. В нашем примере убедиться в настройках конкретного уровня можно, сделав вызов `/loggers/demo`. Чтобы найти корневой уровень регистрационных записей, позволяющий получить информацию обо всех других неуказанных и более конкретных уровнях регистрационных записей, можно сделать вызов `/loggers/ROOT`.

Можно также *обновить* сконфигурированные уровни регистрационных записей, воспользовавшись HTTP-запросом `POST` к соответствующей конечной точке `loggers`.

В примере 13.28 показано обновление сконфигурированного уровня для пакета `demo`.

Пример 13.28. Обновление уровня регистрационных записей

```
curl -i -X POST -H 'Content-Type: application/json' \
  -d '{"configuredLevel": "TRACE"}' \
  http://localhost:8080/loggers/demo
```

Польза от этого, конечно, есть, но применяется данный прием только для одного экземпляра. Если выполнение ведется в облачной среде и имеется сразу несколько одновременно запущенных экземпляров, то полезнее будет снабдить регистратор механизмом пошагового повышения и понижения уровней регистрационных записей всего развертываемого приложения. При использовании Pivotal Web Services или Pivotal Cloud Foundry эта задача решается довольно просто. На рис. 13.4 и 13.5 мы просмотрим регистрационные записи приложения на информационной панели Pivotal AppsManager, а затем переконфигурируем уровни регистрационных записей для Spring Boot-приложения.

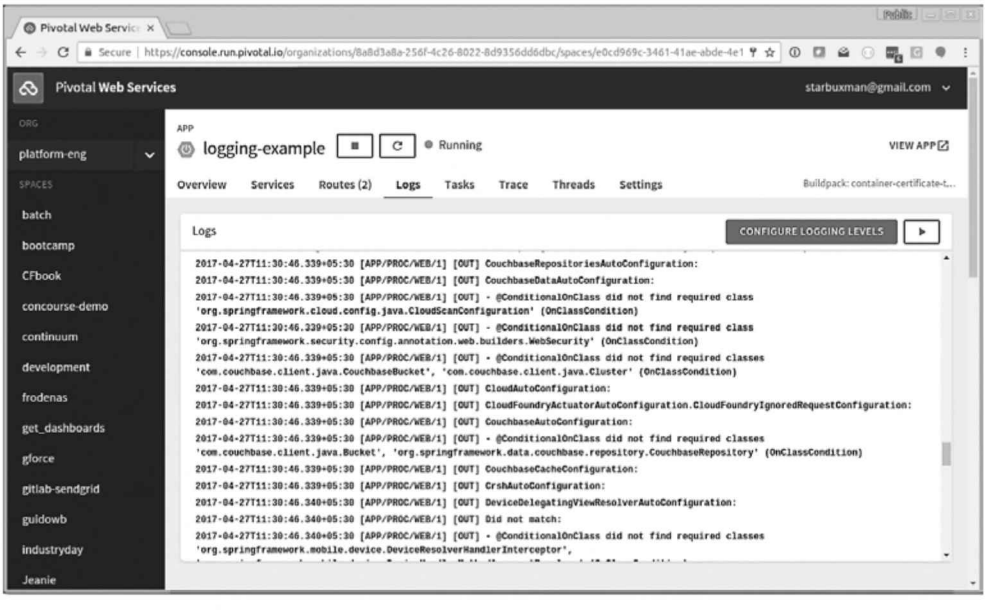


Рис. 13.4. Просмотр регистрационных записей приложения на информационной панели AppsManager

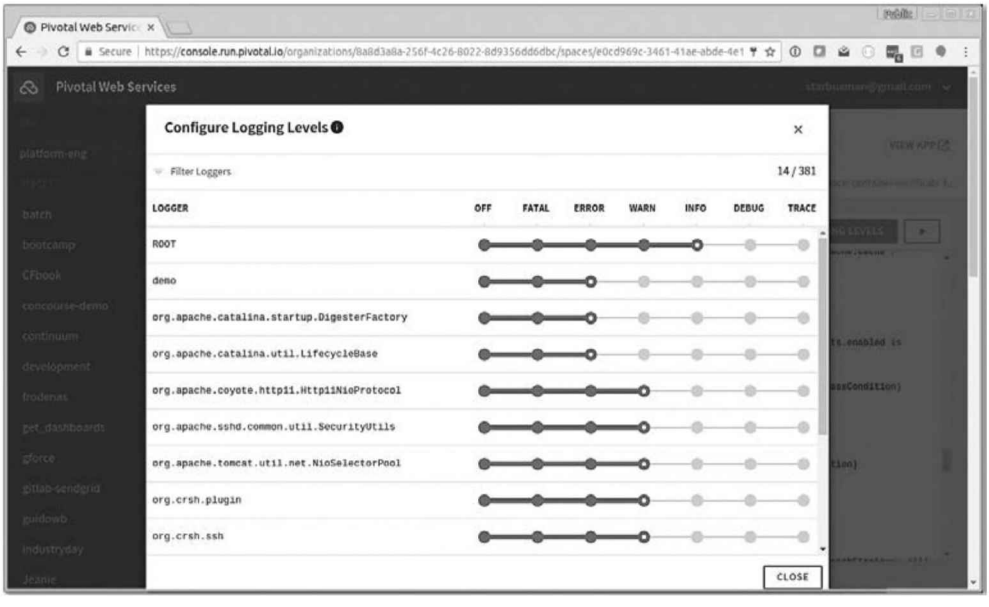


Рис. 13.5. Переконфигурирование уровней регистрационных записей для Spring Boot-приложения на Pivotal Cloud Foundry или Pivotal Web Services

Распределенная трассировка

Вариантов поддержки понимания профиля вашего приложения и его производительности довольно много. Существуют инструментальные технологии на основе агентов, среди которых можно отметить New Relic (проходящую интеграцию со средой Pivotal Cloud Foundry (https://newrelic.com/partner/pivotal) без проблем) и App Dynamics (обладающую таким же свойством (https://docs.pivotal.io/partners/appdynamics/index.html)), где Java-агенты и автоматическое оснащение используются для передачи низкоуровневого представления о характере производительности приложения. На эти инструменты стоит обратить внимание, поскольку они в ходе работы позволяют увидеть перспективы в производительности приложения. APM-инструментарий может дать межъязыковую и инструментальную высокотехнологичную информационную панель, которая отображает сквозное поведение приложения с помощью направляемых вниз HTTP-запросов по доступу к низкоуровневым источникам данных.

Достижения в технологии и облачных вычислениях упростили развертывание и ввод в строй сервисов. Такие вычисления позволяют автоматизировать все сложные моменты и сократить сроки (с дней или недель и до минут), связанные с вводом в строй новых сервисов. В свою очередь, это увеличение скорости позволяет нам

проявлять бóльшую гибкость, рассматривая возможность создания более мелких пакетов из независимо развертываемых сервисов. Быстрое распространение новых сервисов усложняет рассуждение об общесистемных и специфичных для запросов характеристиках производительности.

Когда все функции приложения собраны в *монолите* (так называются приложения, созданные в виде одной большой, неразделимой и развертываемой программы наподобие тех, которые имеют расширения `.war` или `.ear`), рассуждать о местах сбоев намного проще. Утечка памяти? В монолите. Компонент неправильно обрабатывает запрос? В монолите. Теряются сообщения? Тоже, вероятно, в монолите. Распределенное вычисление меняет всю картину.

Под нагрузкой и при масштабировании системы ведут себя по-разному. Описание поведения системы зачастую отличается от ее фактического поведения, которое в различных контекстах может быть разным. При прохождении запросов через систему их важно контекстуализировать. Кроме того, значимой является возможность говорить о природе конкретного запроса и разбираться в том, насколько его поведение имеет отношение к общему поведению подобных запросов в последнюю минуту, час, день или в другой интервал времени, предоставляющий статистически важную выборку. Контекст помогает установить неправильность запроса и то, заслуживает ли он внимания. Отследить ошибки в системе невозможно, пока не будут установлены исходные условия для определения *нормального* запроса. Какая продолжительность будет слишком велика? Для некоторых систем это могут быть микросекунды, а для других — секунды или минуты.

В данном разделе мы рассмотрим, насколько полезной может оказаться среда Spring Cloud Sleuth, поддерживающая распределенную трассировку, в установке нужного контекста и насколько быстрее она позволяет разобраться в текущем поведении системы, именно в *проявляющемся*, а не в указанном поведении.

Поиск разгадок с помощью Spring Cloud Sleuth

Теоретически трассировка не представляет особой сложности. При отправке запроса от одного компонента системы к другому через точки выдачи и поступления *трассировщики* добавляют логику, то есть соответствующий инструментарий, который позволяет сохранить уникальный *идентификатор трассировки* (trace ID), создаваемый при выдаче первого запроса. При поступлении запроса компоненту, находящемуся на пути, присваивается новый *идентификатор промежутка* (span ID), добавляемый к трассе. Трасса представляет собой весь путь запроса, а промежуток — каждый отдельно взятый транзитный шлюз, или запрос, встречающийся на пути. Промежутки могут включать *теги*, иначе метаданные, которые можно использовать для последующей контекстуализации запроса и возможного сопоставления запроса с конкретной транзакцией. Промежутки обычно содержат теги общего характера, такие как стартовые и стоповые отметки времени,

хотя с промежутком можно легко связать семантически уместные теги наподобие идентификатора бизнес-объекта.

Предположим, имеются два сервиса, `service-a` и `service-b`. Если HTTP-запрос поступает на `service-a`, который, в свою очередь, отправляет сообщение через Apache Kafka к `service-b`, то получится один идентификатор трассировки, но два промежутка. У каждого промежутка будут теги, специфичные для запросов. У первого промежутка могут иметься подробности HTTP-запроса. А у второго — подробности сообщения, отправленного поставщику сообщений Apache Kafka.

Среда Spring Cloud Sleuth (http://cloud.spring.io/spring-cloud-sleuth/) (`org.springframework.cloud : spring-cloud-starter-sleuth`) автоматически оснащает инструментами общие каналы связи:

- запросы по технологии рассылки сообщений, использующие Apache Kafka (https://spring.io/blog/2015/04/15/using-apache-kafka-for-integration-and-data-processing-pipelines-with-spring) или RabbitMQ (или любую другую систему рассылки сообщений, для которой имеется связующий компонент Spring Cloud Stream (http://cloud.spring.io/spring-cloud-stream/));
- HTTP-заголовки, полученные контроллерами Spring MVC;
- запросы, проходящие через микропрокси-серверы Netflix Zuul;
- запросы, выполняемые с помощью `RestTemplate` и т. п.;
- запросы, выполняемые через клиент Netflix Feign REST;
- ...и конечно же, для большинства запросов и ответов другого типа, которые могут встретиться в обычном приложении Spring-экосистемы, среда Spring Cloud Sleuth устанавливает за вас подходящее форматирование регистрационных записей, в которых регистрируются идентификатор трассировки и идентификатор промежутка.

При условии запуска разрешающего использование Spring Cloud Sleuth кода, который находится в микросервисе, чьим именем `spring.application.name` будет `my-service-id`, вы увидите в регистрационных записях для вашего микросервиса нечто подобное тому, что показано в примере 13.29.

Пример 13.29. Регистрационные записи, исходящие из приложения, снабженного инструментарием средой Spring Cloud Sleuth

```
2016-02-11 17:12:45.404 INFO [my-service-id,73b62c0f90d11e06,73b6etydf90d11e06,false]
  85184 --- [nio-8080-exec-1] com.example.MySimpleComponentMakingARequest : ...
```

В этом примере `my-service-id` является `spring.application.name`, `73b62c0f90d11e06` выступает идентификатором трассировки, а `73b6etydf90d11e06` — идентификатором промежутка. Данная информация весьма полезна и позволяет применить имеющийся в вашем распоряжении аналитический инструмент регистрационных записей для ее извлечения. При наличии в одном месте, доступном для запроса

и анализа всех регистрационных записей и трассировочной информации, можно увидеть проход запросов через различные сервисы.

Инструментарий, предоставляемый средой Spring Cloud Sleuth, обычно состоит из двух компонентов: объекта, осуществляющего *трассировку* подсистемы, и конкретного экземпляра `SpanInjector<T>` для этой подсистемы. Трассировщиком обычно является некая разновидность перехватчика, отслеживателя, фильтра и т. д., который можно вставить в поток запроса для трассируемого компонента. Если по каким-то причинам не получается применить готовое средство трассировки для нужного компонента, то можно создать и применить собственное.

Какого объема данных будет достаточно?

Какой из запросов должен быть подвергнут трассировке? В идеале для отображения тенденций, показывающих живой, оперативный трафик, понадобится достаточный объем данных. Но перегружать инфраструктуру регистрации и анализа нежелательно. В некоторых организациях могут запрашиваться данные только о каждых тысячных запросах, или о каждых десятых, или о каждых миллионных! По умолчанию порог составляет 10 %, или 0,1, но его можно переопределить, настроив процент выборки (пример 13.30).

Пример 13.30. Изменение порога процента выборки

```
spring.sleuth.sampler.percentage = 0.2
```

В качестве альтернативы подойдет регистрация определения собственного bean-компонента `Sampler` и принятия решения о том, какие запросы должны быть выбраны. Можно сделать более разумный выбор предмета трассировки, например игнорируя успешно выполненные запросы и, скажем, проверяя, находится ли некий компонент в состоянии ошибки либо случилось ли что-нибудь еще.

Определение `Sampler` показано в примере 13.31.

Пример 13.31. Spring Cloud-интерфейс Sampler

```
package org.springframework.cloud.sleuth;

import org.springframework.cloud.sleuth.Span;

public interface Sampler {
    boolean isSampled(Span s);
}
```

Компонент `Span` дает аргумент, представляющий промежуток для текущего выполняемого запроса в большой трассировке. При необходимости можно делать интересующие выборки по конкретным типам запросов: к примеру, задать выборку только запросов с кодом состояния 500 HTTP.

Убедитесь в сформированности набора реалистичных ожиданий для вашего приложения и инфраструктуры. Вполне вероятно, что схемы использования вашего приложения для выявления тренда или схем потребуют чего-то более или менее точного. Это должна быть онлайн-телеметрия. Во многих организациях такие данные хранятся не более нескольких дней или в крайнем случае не более недели.

OpenZipkin: графическое представление стоит тысячи трассировок

Со сбора данных все только начинается, но цель — *разобраться* в них, а не собрать. Чтобы оценить общую картину, нужно выйти за рамки отдельно взятых событий. Мы воспользуемся проектом OpenZipkin (https://github.com/openzipkin). Это версия проекта Zipkin с открытым кодом (рис. 13.6), появившаяся в Twitter в 2010 году и основанная на работах по Google Dapper (https://ai.google/research/pubs/pub36356).

Рис. 13.6. OpenZipkin — версия Zipkin с открытым кодом

Прежде темп развития версии Zipkin с открытым кодом отличался от темпа развития версии, используемой внутри Twitter. OpenZipkin представляет собой синхронизацию усилий: OpenZipkin (https://github.com/openzipkin) — это Zipkin и, когда здесь говорится о Zipkin, речь идет о той версии, которая воплощена в OpenZipkin.

В Zipkin предоставляется REST API, с которым клиенты общаются напрямую. Указанный REST API написан с помощью Spring MVC и Spring Boot. В Zipkin даже поддерживается реализация этого REST API на основе Spring Boot. Применять его не сложнее, чем непосредственно задействовать имеющуюся в Zipkin аннотацию `@EnableZipkinServer`. Сервер Zipkin делегирует операции записи уровню поддержки постоянного хранения данных через `SpanStore`. На данный момент имеется поддержка использования MySQL или готового хранилища `SpanStore`, работающего в оперативной памяти.

Кроме непосредственного обращения к Zipkin REST API, можно также публиковать сообщения для сервера Zipkin через такую привязку к Spring Cloud Stream, как RabbitMQ или Apache Kafka, что и показано в примере 13.32. Создайте новое приложение Spring Boot, добавьте к пути к классам запись `org.springframework.cloud : spring-cloud-sleuth-zipkin-stream`, а затем добавьте к приложению Spring Boot аннотацию `@EnableZipkinStreamServer`, чтобы принимать и адаптировать поступающие Sleuth `Span`-экземпляры на основе Spring Cloud Stream к `Span`-типу Zipkin. Далее они будут сохранены на постоянной основе с помощью сконфигурированного хранилища `SpanStore`. Можете воспользоваться той привязкой к Spring Cloud Stream, которая вам больше нравится, но в данном случае мы применили средство Spring Cloud Stream RabbitMQ (`org.springframework.cloud : spring-cloud-starter-stream-rabbitmq`).

Пример 13.32. Код сервера Zipkin

```
package demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.sleuth.zipkin.stream.EnableZipkinStreamServer;

❶
@EnableZipkinStreamServer
@SpringBootApplication
public class ZipkinApplication {

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

❶ Это предписание серверу Zipkin на отслеживание поступающих промежутков.

Для визуализации запросов добавьте компонент пользовательского интерфейса Zipkin UI (`io.zipkin : zipkin-ui`) к путям к классам сервера Zipkin Stream. Запустите этот компонент (по адресу http://localhost:9411, в том же месте, где находится сервер потока) — и увидите все последние трассировки, если таковые существуют. При их отсутствии их можно создать.

К запущенному и работающему серверу можно подключить пару клиентов и сделать несколько запросов. Рассмотрим два весьма обычных сервиса с выдуманными названиями `zipkin-client-a` и `zipkin-client-b`. У обоих сервисов есть в путях к классам требуемая привязка (`org.springframe work.cloud : spring-cloud-starter-stream-rabbit`) и клиент Spring Cloud Sleuth Stream (`org.springframework.cloud : spring-cloud-sleuth-stream`).

Клиент `zipkin-client-a` настроен на запуск на порте 8082. Сообщить клиенту о местонахождении сервиса поможет имеющееся свойство `message-service`. Можно было легко воспользоваться регистрацией сервиса и обнаружить его в этом

месте. Клиент делает запрос к нижестоящему сервису, применяя bean-компонент `RestTemplate`, определенный в основном классе. Важно, чтобы клиент, в данном случае `RestTemplate`, был bean-компонентом Spring. Из конфигурации Spring Cloud Sleuth должно быть понятно, где искать bean-компонент, если он сможет сконфигурировать поддерживающий Sleuth перехватчик для всех запросов, которые проходят через него (пример 13.33).

Пример 13.33. Клиент сообщений

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.util.Map;

@RestController
class MessageClientRestController {

 @Autowired
 private RestTemplate restTemplate;

 @Value("${message-service}")
 private String host;

 @RequestMapping("/")
 Map<String, String> message() {

  //@formatter:off
  ParameterizedTypeReference<Map<String, String>> ptr =
          new ParameterizedTypeReference<Map<String, String>>() { };
  //@formatter:on

  return this.restTemplate.exchange(this.host, HttpMethod.GET, null, ptr)
   .getBody();
 }
}
```

Клиент `zipkin-client-b` настроен на запуск на порте 8081. Он получает все трассировочные заголовки от входящего запроса и наряду с сообщением включает их в свои ответы (пример 13.34).

Пример 13.34. Сервис сообщений

```
package demo;

import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpServletRequest;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@RestController
class MessageServiceRestController {

 @RequestMapping("/")
 Map<String, String> message(HttpServletRequest httpRequest) {

  List<String> traceHeaders = Collections.list(httpRequest.getHeaderNames())
   .stream().filter(h -> h.toLowerCase().startsWith("x-"))
   .collect(Collectors.toList()); (1)

  Map<String, String> response = new HashMap<>();
  response.put("message", "Hi, @ " + System.currentTimeMillis());
  traceHeaders.forEach(h -> response.put(h, httpRequest.getHeader(h)));
  return response;
 }
}
```

(1) Сбор всех заголовков, предоставленных Spring Cloud Sleuth (тех, что начинаются с `x-`), из исходящего запроса от `zipkin-client-a` и включение их в созданный JSON-ответ вместе с оригинальным сообщением.

Сделайте несколько запросов по адресу http://localhost:8082. Будут получены ответы, подобные показанным в примере 13.35.

Пример 13.35. Ответ, полученный на отслеживаемый запрос

```
{
   "x-b3-parentspanid" : "9aa83c71878b6cd4",
   "x-b3-sampled" : "1",
   "message" : "Hi, 1493358280026",
   "x-b3-traceid" : "9aa83c71878b6cd4",
   "x-span-name" : "http:",
   "x-b3-spanid" : "668b8e088a35f1db"
}
```

Теперь можно изучить запросы на сервере Zipkin по адресу http://localhost:9411. Чтобы получить более детальный контроль над результатами, их можно отсортировать по новизне, длине и т. д. На рис. 13.7 показаны результаты поиска трассировок в сервере Zipkin.

На рис. 13.8 представлена возможность изучения подробностей трассировок.

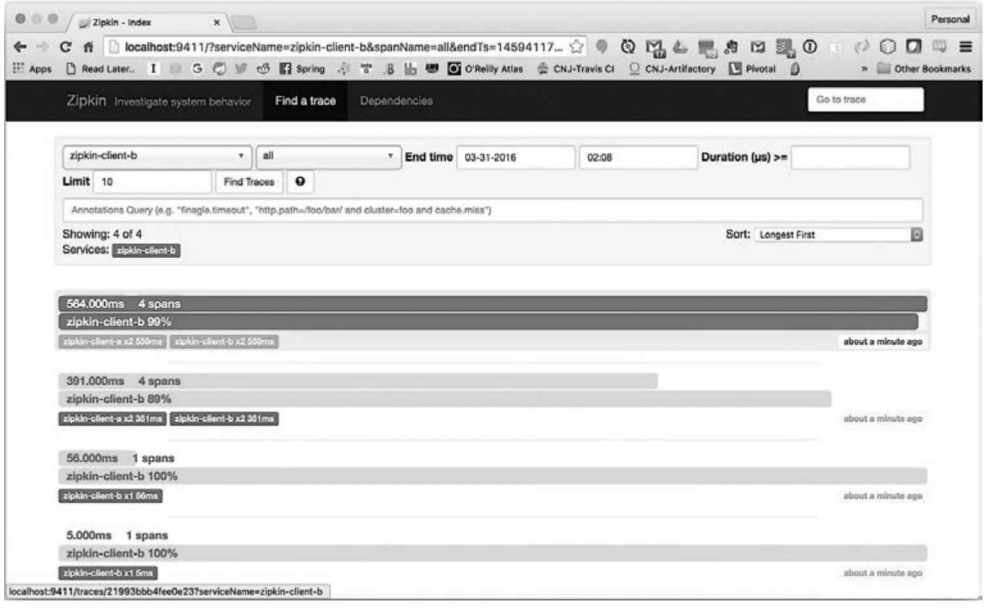


Рис. 13.7. Результаты поиска трассировок на главной странице Zipkin

Каждый отдельно взятый промежуток также несет с собой информацию (теги) о конкретном, связанном с ним запросе. На рис. 13.9 показано, что эти подробности можно рассмотреть, щелкнув кнопкой мыши на конкретном промежутке.

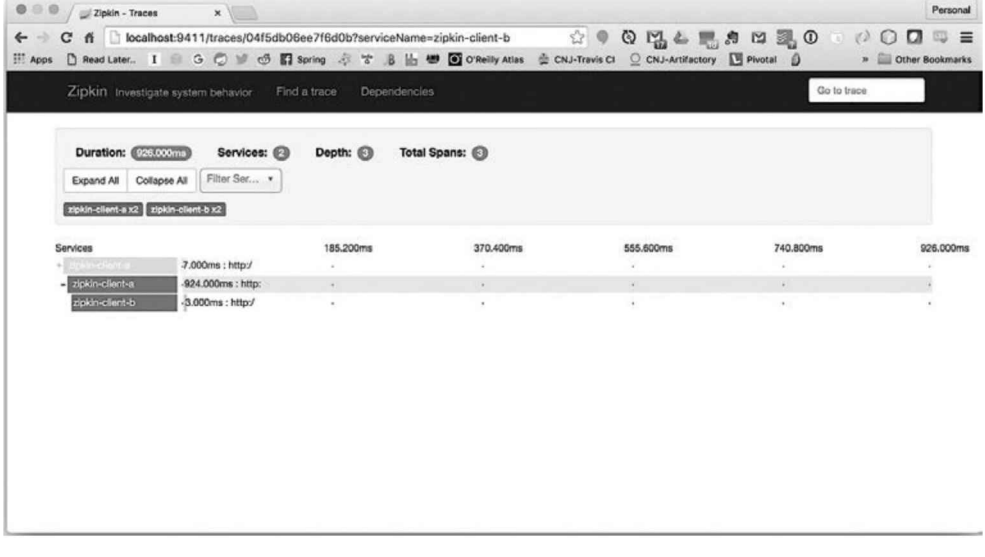


Рис. 13.8. Страница подробностей, касающихся отдельных промежутков для одной трассировки

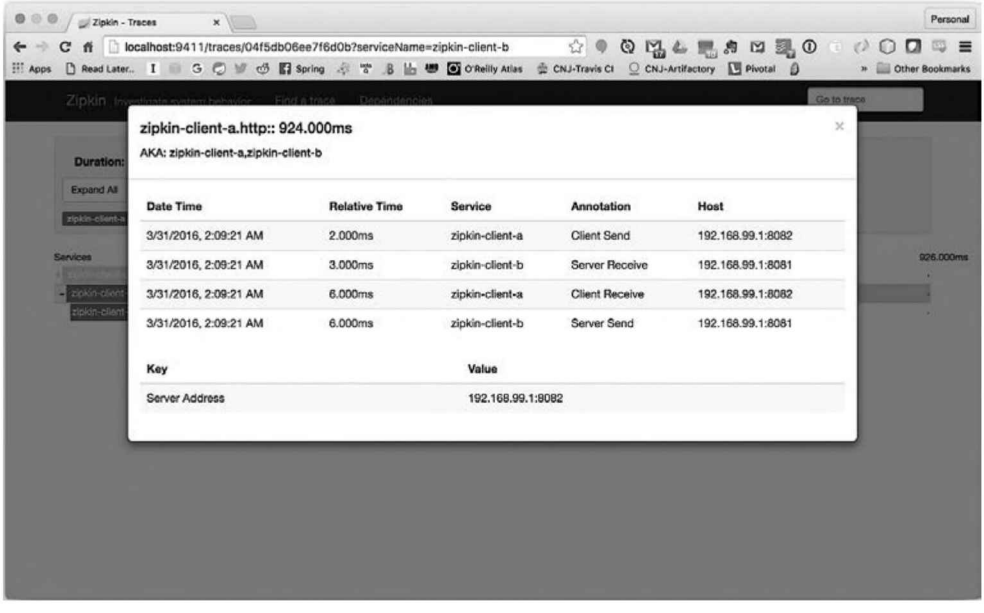


Рис. 13.9. Панель подробностей, на которой показывается информация, соответствующая заданному промежутку, и связанные с ним теги

У сервера Zipkin весьма завидная позиция: ему известно, как сервисы взаимодействуют друг с другом. Он знает топологию вашей системы. Он, как показано на рис. 13.10, даже создает удобную визуализацию этой топологии, если вы щелкаете на вкладке Dependencies (Зависимости).

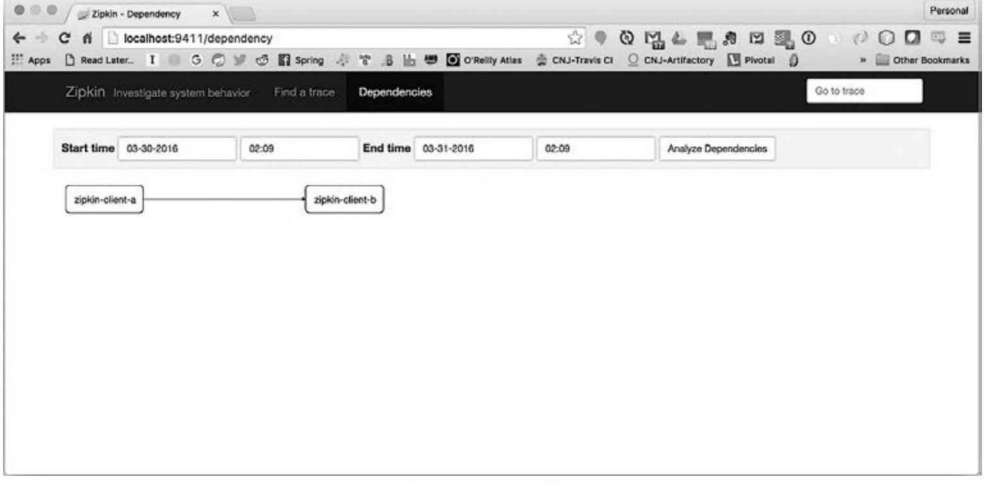


Рис. 13.10. Визуализация топологии ваших сервисов

Каждый элемент визуализации может дать дополнительную информацию, включая то, какой компонент его использует и сколько отслеженных вызовов было сделано. Эта возможность показана на рис. 13.11.

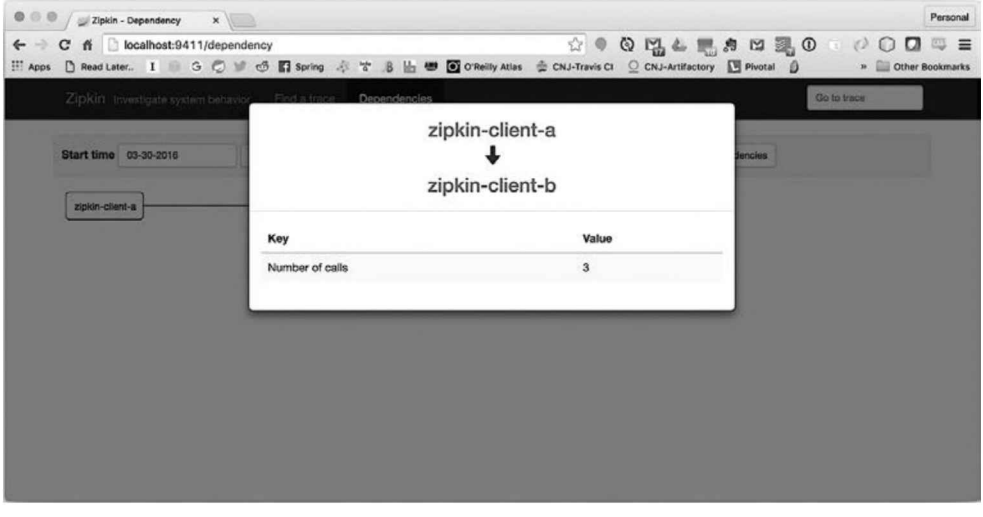


Рис. 13.11. Подробности каждого сервиса в визуализации зависимостей

Если переместить приложение на облачную платформу, такую как Cloud Foundry, то инфраструктура маршрутизации также должна будет справляться с созданием или сохранением заголовков трассировки. И в Cloud Foundry это делается: как только запросы поступают в облачный маршрутизатор системы, заголовки добавляются к вашему запущенному приложению или сохраняются в нем (как в вашем Spring-приложении).

Отслеживание других платформ и технологий

Для задач, решаемых на основе использования среды Spring, распределенная трассировка не может быть проще, но по своей природе она является вопросом, охватывающим все сервисы, независимо от технологий, на основе которых они реализованы. Работа, направленная на стандартизацию словаря и понятий современной трассировки для нескольких языков и платформ, вылилась в инициативу OpenTracing (https://github.com/opentracing). API OpenTracing поддерживается *множеством* весьма крупных организаций.

Эта работа определяет языковые привязки. Уже имеются реализации для JavaScript, Python, Go и т. д. Команда Spring будет поддерживать и отслеживать концептуальную совместимость Spring Cloud Sleuth с этой работой. Ожидается, но не подразумевается, что привязки будут, как правило, иметь Zipkin в качестве своего серверного приложения.

Работа по OpenTracing находится в относительно зачаточном состоянии, поэтому, возможно, будет проще воспользоваться клиентом OpenZipkin, привязанным к другому языку, а не реализацией на основе OpenTracing.

Информационные панели

До сих пор мы занимались в основном поиском способов получения информации по каждому узлу и ее настройке. Но польза от такой информации проявится только при получении возможности подключить ее к более общей картине о более крупной системе. К примеру, Actuator публикует информацию о любом заданном узле, но предполагает наличие некой инфраструктуры, занимающейся ее потреблением и сведением в общее представление, подобно тому как Google управляет сервисами с помощью своего средства Borg Monitoring (Borgmon). Последнее является централизованным решением по управлению и отслеживанию, которое используется внутри Google, но полагается на каждый узел, отображающий служебную информацию. Сервисы, совместимые с Borgmon, публикуют информацию через конечные точки HTTP, даже если отслеживаемые сервисы сами по себе не выступают сервисами HTTP. В данной главе, помимо узловых конечных точек, предоставляемых Actuator, будут рассмотрены несколько вариантов централизации и визуализации самой системы.

В данном разделе мы изучим несколько полезных инструментов, поддерживающих вывод на панель весьма важной информации, позволяющей оценить как общее функционирование, так и решение бизнес-задач. Информационные панели зачастую создаются на основе уже рассмотренных выше инструментов, выдавая соответствующую информацию в едином представлении. Эти средства в вопросах регистрации и обнаружения сервисов в системе полагаются на сервисы, выполняющие именно эти задачи, после чего собирают исходящую от них информацию. Подробности работы с такими реестрами сервисов, как Eureka компании Netflix, можно найти в главе 7, в разделе, посвященном маршрутизации. Здесь же мы будем полагаться на доступность этого реестра, чтобы наши информационные панели могли обнаруживать и отслеживать сервисы, развернутые в системе. В качестве альтернативы можно воспользоваться Hashicorp Consul, или Apache Zookeeper, или любым другим реестром, для которого доступна реализация абстракции Spring Cloud `DiscoveryClient`.

Отслеживание нижестоящих сервисов с помощью Hystrix Dashboard

Добавить инструментальное оснащение в код других команд мы не можем, как и настоять на том, чтобы они создавали приложения с помощью лучших в своем классе технологий, таких как Cloud Foundry и Spring Cloud. Обычно мы не можем заставить другие команды и другие организации делать что-либо. Лучшее из того, что нам доступно, — это защитить самих себя от потенциальных сбоев в нижестоящем

коде. Один из способов такой защиты выражается в заключении потенциально ненадежных вызовов типа «сервис-сервис» в автомат защиты.

В среде Spring Cloud поддерживается простая интеграция с автоматом защиты Netflix Hystrix. Мы познакомились с ним в главе 12, в разделе «Изоляция сбоев и постепенное снижение качественных характеристик». Как напоминание рассмотрим простой пример, произвольно вставляющий сбой при выдаче вызовов либо к http://google.com, либо к http://yahoo.com (пример 13.36).

Пример 13.36. Использование автомата защиты Hystrix (предполагая где-либо наличие аннотации @EnableCircuitBreaker)

```
package com.example;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.net.URI;
import java.util.Random;

@RestController
class ShakyRestController {

 @Autowired
 private RestTemplate restTemplate;

 ❶
 public ResponseEntity<String> fallback() {
  return ResponseEntity.ok("ONOES");
 }

 ❷
 @HystrixCommand(fallbackMethod = "fallback")
 @RequestMapping(method = RequestMethod.GET, value = "/google")
 public ResponseEntity<String> google() {
  return this.proxy(URI.create("http://www.google.com/"));
 }

 @HystrixCommand(fallbackMethod = "fallback")
 @RequestMapping(method = RequestMethod.GET, value = "/yahoo")
 public ResponseEntity<String> yahoo() {
  return this.proxy(URI.create("http://www.yahoo.com"));
 }

 private ResponseEntity<String> proxy(URI url) {

  if (new Random().nextInt(100) > 50) {
```

```
    throw new RuntimeException("tripping circuit breaker!");
   }

 ResponseEntity<String> responseEntity = this.restTemplate.getForEntity(url,
  String.class);

 return ResponseEntity.ok()
  .contentType(responseEntity.getHeaders().getContentType())
  .body(responseEntity.getBody());
 }

}
```

❶ Обеспечивает откат, вызываемый при выдаче автоматом защиты исключения. В данном случае возвращается бессмысленное значение типа `String`.

❷ Мы декорировали разнообразные REST-вызовы автоматом защиты таким образом, чтобы безопасно обрабатывать те вызовы нижестоящих сервисов, которые могут завершиться сбоем.

Каждый узел, использующий автомат защиты Hystrix, также выдает контрольный поток событий, периодически посылаемых сервером (server-sent event, SSE) для каждого узла, содержащего автомат защиты. SSE-поток доступен с адреса http://localhost:8000/hystrix.stream при условии сохранения исходной конфигурации и отслеживания показанного выше кода на порте 8000. Этот поток постоянно обновляется. В нем содержится информация о потоке трафика через автомат защиты, включая количество выполненных запросов и то, открыт автомат (в таком случае запросы являются сбойными и направляются в обработчик *отката*) или закрыт (и выданные запросы стремятся к успешному достижению нижестоящего сервиса), а также статистика самого трафика. Хотя мы не в состоянии снабдить своим оборудованием чужие сервисы, мы можем отслеживать поток запросов через автомат защиты в качестве своеобразного виртуального монитора нижестоящих сервисов. Если автомат открыт и запросы через него не проходят, то, вполне возможно, это является признаком сбоя нижестоящего сервиса.

Мы можем отслеживать автоматы защиты. Поток автомата не такой интенсивный, чтобы его нельзя было изучать напрямую, но этот поток можно взять и передать другому компоненту, Hystrix Dashboard, который сможет выполнить визуализацию потока запросов, проходящего через автомат защиты к конкретному узлу.

Создайте новое приложение Spring Boot и добавьте к путям к классам запись `org.springframework.cloud : spring-cloud-starter-hystrix-dashboard`. Добавьте к классу `@Configuration` аннотацию `@EnableHystrixDashboard`, а затем запустите приложение. Пользовательский интерфейс Hystrix Dashboard UI доступен по адресу /hystrix.html.

Неплохо! Но это по-прежнему всего лишь один узел. Данный код бесполезен при масштабировании, когда имеется более одного экземпляра одного и того же сервиса или, кроме того, еще несколько сервисов. Информационная панель Hystrix Dashboard, распознающая только один узел, вряд ли принесет много пользы;

придется подключать поодиночке каждый адрес /hystrix.stream. Для объединения всех потоков со всех узлов в один поток можно применить Spring Cloud Turbine. Затем получившийся объединенный поток подключить к информационной панели Hystrix Dashboard. Spring Cloud Turbine может объединять сервисы, используя их регистрацию и обнаружение (через имеющуюся в Spring Cloud абстракцию реестра сервисов `DiscoveryClient`) или с помощью таких предоставляемых поставщиков сообщений, как RabbitMQ и Apache Kafka (путем имеющейся в Spring Cloud абстракции рассылки сообщений `Stream`).

Добавьте к новому приложения Spring Boot записи `org.springframework.boot : spring-boot-starter-web`, `org.springframe work.cloud : spring-cloud-starter-stream-rabbit` и `org.springframe work.cloud : spring-cloud-starter-turbine-stream` (пример 13.37).

Пример 13.37. Использование Spring Cloud Turbine для объединения контрольного потока событий, периодически посылаемых сервером, исходящего от нескольких автоматов защиты, в один поток

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.turbine.stream.EnableTurbineStream;

❶
@EnableTurbineStream
@SpringBootApplication
public class TurbineApplication {

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

❶ Сбор потоков в один поток средством Turbine, работающим на основе Spring Cloud Stream.

При запуске сервис Spring Cloud Turbine станет обслуживать поток по адресу http://localhost:8989/hystrix.stream, где 8989 является портом по умолчанию. Этот порт можно переопределить, указав значение для `turbine.stream.port`. Для примеров в данной книге мы указали порт 8010.

От всех клиентов, имеющих у себя автомат защиты, потребуется небольшое обновление для поддержки применения Spring Cloud Turbine. Добавьте привязку Spring Cloud Stream (мы используем `spring-cloud-starter-stream-rabbit`), а затем `org.springframework.cloud : spring-cloud-netflix-hystrix-stream`. Эта последняя зависимость адаптирует обновление состояния автомата защиты к отправке сообщений через конкретно выбранную вами привязку к Spring Cloud Stream. Как часть всего этого, средству Spring Cloud Turbine потребуется информация о локальном узле и кластере. Проще всего данную информацию можно предоставить,

воспользовавшись регистрацией и обнаружением сервисов. Добавьте также к путям к классам реализацию абстракции `DiscoveryClient` (мы применяем `org.springframework.cloud : spring-cloud-starter-eureka`).

Это, конечно же, потребует запуска где-либо сервиса регистрации Eureka компании Netflix. Подробности можно найти в материале по регистрации и обнаружению сервисов, изложенном в главе 7. Мы собираемся использовать реестр сервисов довольно часто, поскольку рассматриваем способы наблюдения за системами, а не просто за отдельными узлами. Вы также можете запустить один из реестров и не выключать его, если хотите следовать всему, что будет делаться с этого момента.

Перезапустите клиент, а затем снова зайдите на информационную панель автомата защиты. Подключите конечную точку `hystrix.stream` из Spring Cloud Turbine (http://localhost:8010/hystrix.stream, если используете наш код). На рис. 13.12 показан результат просмотра потока Hystrix.

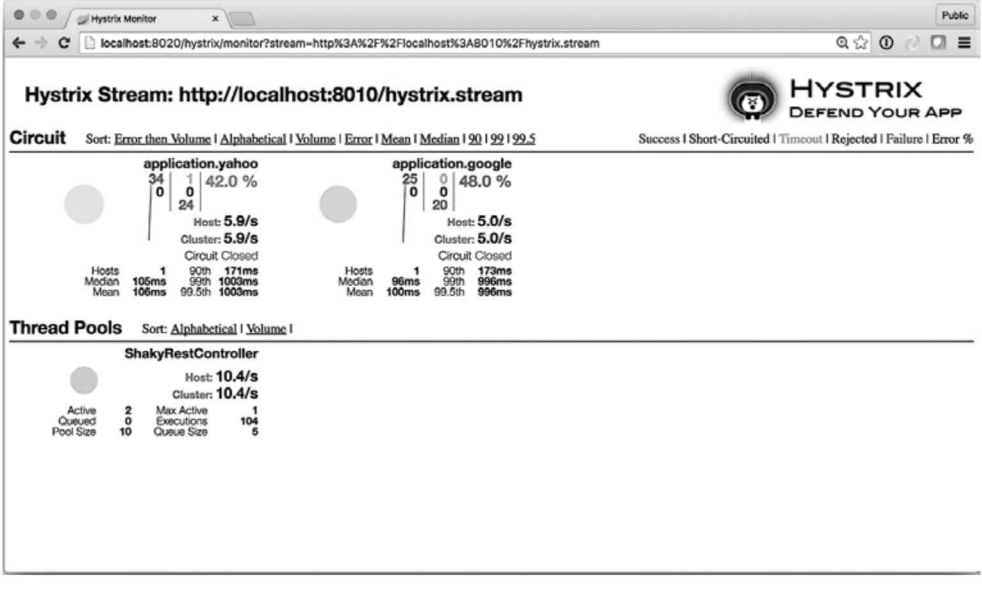


Рис. 13.12. Информационная панель Hystrix

Технологии, подобные Hystrix Dashboard, играют важную роль, но повышают эксплуатационные расходы. В идеале этой работой должна заниматься платформа, причем в автоматическом режиме. Использование среды Cloud Foundry подразумевает наличие в каталоге сервисов опорного сервиса Hystrix Dashboard, который уже подключен к применению готового агрегатора потоков Spring Cloud Turbine.

Spring Boot Admin от команды Codecentric

Spring Boot Admin (https://github.com/codecentric/spring-boot-admin) — это проект от специалистов Codecentric. Он предоставляет объединенный взгляд на сервисы и поддерживает движение вниз, в сторону сервисов на основе Spring Boot с Actuator, открывающим конечные точки (регистрационные записи, среду окружения JMX, регистрации запросов и т. д.).

Чтобы задействовать это средство, нужно установить реестр сервисов (мы применим рассмотренный ранее экземпляр Netflix Eureka). Установите новое приложение Spring Boot и добавьте к путям к классам следующие зависимости: `de.codecentric : spring-boot-admin-server` и `de.codecentric : spring-boot-admin-server-ui`. Версии, конечно, не будут стоять на месте, поэтому обратитесь к Git-хранилищу и к предпочитаемому вами хранилищу Maven. В данном примере использовалась версия `1.5.0` (пример 13.38).

Пример 13.38. Установка экземпляра Spring Boot Admin

```
package com.example;

import de.codecentric.boot.admin.config.EnableAdminServer;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@EnableDiscoveryClient
@EnableAdminServer
@SpringBootApplication
public class SpringBootAdminApplication {

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

Подключение клиентов к серверу будет осуществляться с поддержкой Spring Cloud-абстракции `DiscoveryClient`. Добавьте к сервису Spring Boot Admin и ко всем клиентам зависимость `org.springframework.cloud : spring-cloud-starter-eureka` и аннотацию `@EnableDiscoveryClient`. За счет регистрации и обнаружения сервиса Spring Boot Admin мы избегаем явного оповещения своих клиентов о его наличии. Кроме того, можно воспользоваться клиентской зависимостью Spring Boot Admin, `de.codecentric : spring-boot-admin-starter-client`. В случае ее применения нужно определить свойство `spring.boot.admin.url`, указывающее клиентам на экземпляр сервера Spring Boot Admin.

Вашему клиенту также понадобится Spring Boot Actuator. В Spring Boot 1.5 и более старших версиях конечные точки Actuator закрыты и требуют аутентификации. Самый простой способ обойти это препятствие — отключить аутентификацию (`management.security.enabled = false`) для самих конечных точек управления;

в противном случае некоторые функциональные возможности Spring Boot Admin не смогут проявиться.

Запустите клиент и зайдите на страницу Spring Boot Admin (показанную на рис. 13.13).

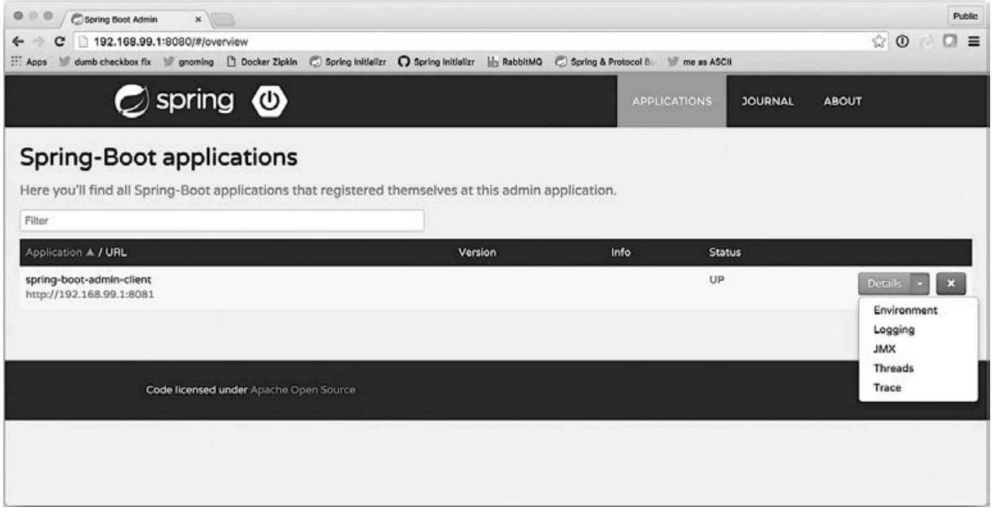


Рис. 13.13. Центральный экран Spring Boot Admin со списком зарегистрированных сервисов

В нашем примере, как показано на рис. 13.14 и 13.15, запуск произведен на порте 8080.

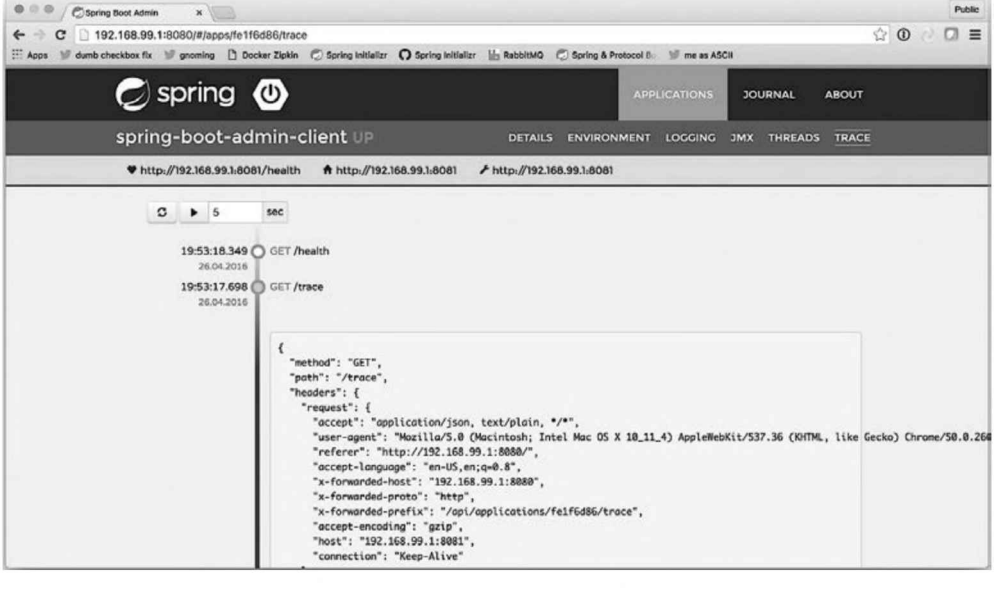


Рис. 13.14. Перечисление запросов (/trace) в Spring Boot Admin

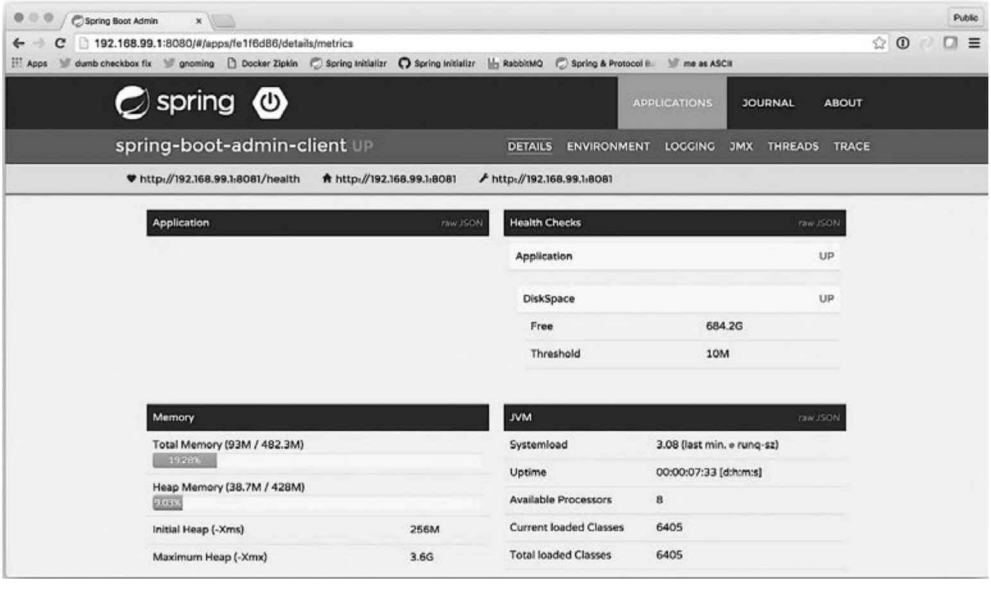


Рис. 13.15. Экран детализации в Spring Boot Admin

Spring Boot Admin предоставляет еще один способ увидеть объединение сервисов в системе и позволяет углубиться в их состояние.

Информационная панель Ordina Microservices Dashboard

Подразделение JWorks компании Ordina разработало еще одну информационную панель, предоставляющую очень удобное визуальное перечисление зарегистрированных в системе сервисов. Она также выполняет обнаружение сервисов с помощью поддержки абстракции `DiscoveryClient`, принадлежащей Spring Cloud. Следовательно, для этого потребуется установить ранее упомянутый реестр сервисов и реализовать клиентские пути к классам (пример 13.39).

Пример 13.39. Установка экземпляра информационной панели Microservices Dashboard

```
package com.example;

import be.ordina.msdashboard.EnableMicroservicesDashboardServer;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
@EnableMicroservicesDashboardServer
public class MicroservicesDashboardServerApplication {

 public static void main(String[] args) {
```

```
  SpringApplication.run(MicroservicesDashboardServerApplication.class, args);
 }
}
```

Информационная панель Microservices Dashboard (рис. 13.16) предоставляет визуализацию порядка соединения сервисов. В ней имеются четыре разные полосы, обозначающие отображаемые уровни компонентов системы:

- *UI Components* обозначают записи, касающиеся компонентов пользовательского интерфейса (например, директивы Angular);
- *Resources* (ресурсы) могут представлять информацию, извлеченную из конечной точки `/mappings` Spring Boot Actuator с исключенными по умолчанию spring-отображениями или гипермедиассылками, которые отображаются на индексный ресурс через индексный контроллер;
- *Microservices* (микросервисы) отображают обнаруженные сервисы (Spring Boot или другие), используя принадлежащую Spring Cloud абстракцию `DiscoveryClient`;
- *Backends* (внутренние интерфейсы) отображают любые индикаторы работоспособности `HealthIndicators`, найденные в обнаруженных микросервисах, то есть компоненты, от которых зависят сервисы и которые способны дать сбой.

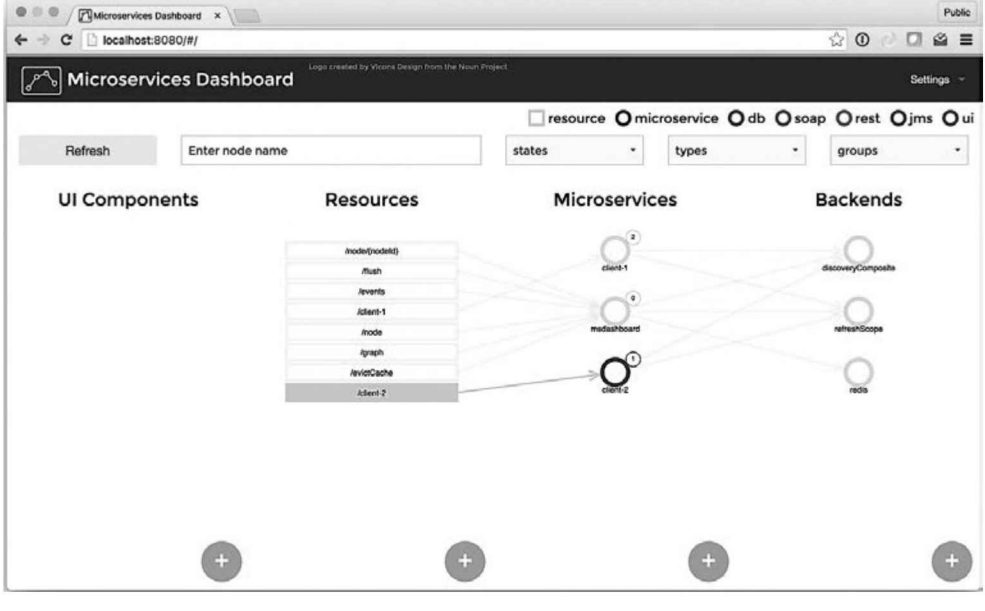


Рис. 13.16. Информационная панель Microservices Dashboard, на которой перечислены сервисы и представлена информация об интересующих компонентах в этих сервисах

Информационная панель Microservices Dashboard поддерживает углубление в состояние узлов, имена, типы и группы. Можно добавить виртуальные узлы, которые не обнаруживаются в автоматическом режиме, но о которых желательно сообщить

панели Microservices Dashboard. По меньшей мере эти узлы могут послужить заместителями того, что со временем планируется создать на их месте.

AppsManager платформы Pivotal Cloud Foundry

Две только что рассмотренные информационные панели в вопросах извлечения информации о процессах Java зависят от принадлежащего Spring Boot актуатора Actuator. Но со временем у вас возникнут планы запуска кода на платформе, и на ней будут и другие активные части, подобные контейнеру (в свою очередь, имеющему собственные отслеживаемые состояния работоспособности), в котором запускаются собственные прикладные процессы, опорные сервисы и т. д. В случае использования чего-то подобного Cloud Foundry не будет препятствий для запуска .NET- или Python-приложений или программ, созданных с применением других предпочитаемых вами технологий, и у них также будут их собственные состояния. Вполне возможно, что оптимальная информационная панель будет централизованно видна во всех приложениях системы. Если доступна более подробная диагностика, как имеющаяся в Spring Boot Actuator, то и такая степень разрешения тоже должна быть видна. Именно это и предоставляет диспетчер приложений AppsManager платформы Pivotal Cloud Foundry. Мы уже встречались с ним в данной главе, поэтому здесь представим лишь его обзор.

Можно просмотреть все приложения для заданного пользователя, принадлежащего конкретной организации и конкретному пространству, что и показано на рис. 13.17.

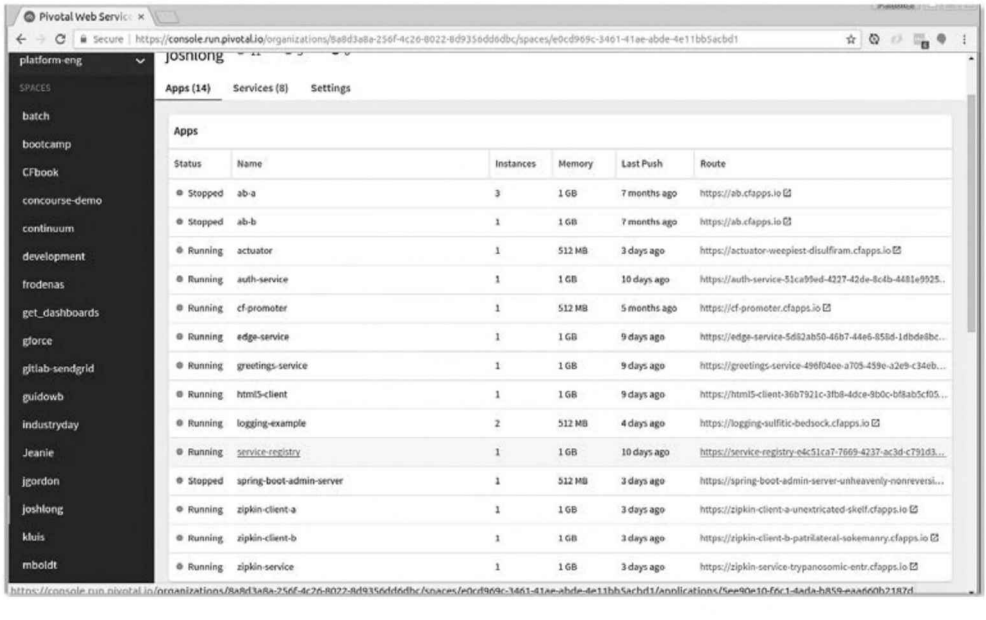


Рис. 13.17. Все приложения в конкретном пространстве Cloud Foundry, которое, в свою очередь, является частью организации

Как показано на рис. 13.18, можно просмотреть и подробности отдельно взятого приложения.

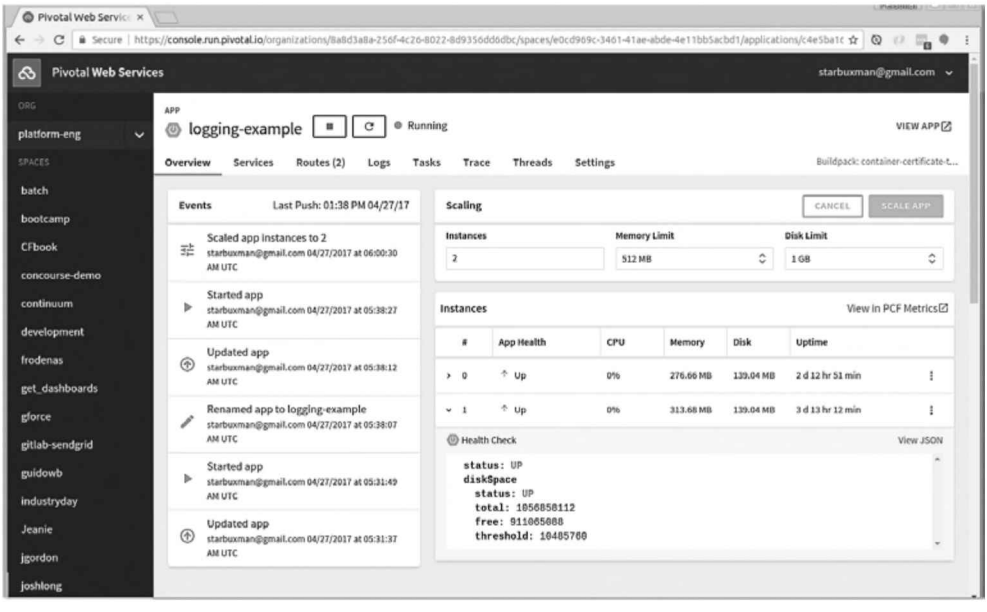


Рис. 13.18. Подробности конкретного приложения Cloud Foundry. Кроме того, видна информация, полученная из Spring Boot Actuator

Восстановление работоспособности

До сих пор основное внимание уделялось сбору информации о состоянии системы, а также совершенствованию ее визуализации. А куда употребить эти знания? В статической дооблачной среде усовершенствованная визуализация может использоваться для вывода оповещений, которые потом (мы надеемся) превратятся в вызов оператора по средствам связи или в получение кем-нибудь сообщения по электронной почте. Ко времени подачи оповещения реакция на негативное развитие ситуации уже может запоздать и у кого-то могут возникнуть проблемы с использованием системы. Облачные вычисления (поддерживающие работу с API) изменяют такую динамику: проблемы приложений можно решать с помощью программных способов. Если, к примеру, системе требуется дополнительный объем памяти, то не требуется подавать заявку в ITIL; достаточно послать API-запрос. Можно поддерживать *автоматическое восстановление работоспособности*.

Распределенные вычисления изменяют способ наших представлений о доступности экземпляра приложения. В адекватно распределенной системе допустить наличие всего одного экземпляра, доступного 100 % времени, практически не-

возможно. Вместо этого основное внимание нужно уделить созданию системы, в которой сервис способен каким-то образом восстанавливать свою деятельность. Следует оптимизировать время восстановления работоспособности. Если это время составляет ноль секунд, то, значит, сервис обладает *эффективной* стопроцентной доступностью, но такое изменение в подходе повлечет коренные изменения в порядке проектирования системы. Этот эффект достижим только при возможности программирования самой платформы.

Платформа даже в состоянии предоставить некоторые базовые возможности по автоматическому восстановлению работоспособности. Большинство облачных платформ позволяют отслеживать состояние работоспособности. При запросе у Cloud Foundry установки десяти экземпляров приложения данная платформа обеспечит запуск как минимум десяти экземпляров. Даже если экземпляр выйдет из строя, Cloud Foundry его перезапустит.

Большинство облачных платформ, включая Pivotal Web Services и Pivotal Cloud Foundry, поддерживает *автомасштабирование*. Его механизм отслеживает информацию на уровне контейнеров, касающуюся оперативной памяти и ресурсов центрального процессора, и в случае необходимости добавляет объем путем запуска новых экземпляров приложения. На платформе Pivotal Web Services нужно лишь создать экземпляр сервиса типа `appautoscaler`, а затем связать его с приложением. Его настройку необходимо провести на панели управления, имеющейся в консоли Pivotal Web Services Console (пример 13.40).

Пример 13.40. Создание сервиса autoscaler на платформе Pivotal Web Services

```
cf marketplace
Getting services from marketplace in org platform-eng / space joshlong as ..
OK

service         plans         description
..
app-autoscaler bronze, gold Scales bound applications in response to load (beta)
..
```

Но при этом еще есть резервы для применения более сложных средств восстановления работоспособности. Можно привести ряд весьма интересных примеров: Winston от Netflix (https://medium.com/netflix-techblog/introducing-winston-event-driven-diagnostic-and-remediation-platform-46ce39aa81cc), Nurse компании LinkedIn (https://engineering.linkedin.com/sre/introducing-nurse-auto-remediation-linkedin), FBAR от Facebook (https://www.facebook.com/notes/facebook-engineering/making-facebook-self-healing/10150275248698920) и инструмент с открытым исходным кодом StackStorm (https://stackstorm.com/). Эти средства упрощают определение каналов, составляемых из неделимых функциональных единиц, сочетание которых позволяет решить вашу задачу. Средства работают в понятиях хорошо известных входных событий, инициируемых агентами отслеживания либо сенсорами или другими развернутыми

в системе индикаторами. В традиционной архитектуре эти события будут инициировать полезные оповещения, но для ряда классификаций проблем можно также инициировать потоки автоматического восстановления работоспособности.

Мы рекомендуем изучить некоторые из этих подходов. И все же большинство из них основаны на средах, не имеющих преимуществ по уровням гарантий, которые предоставляют платформы, подобные Cloud Foundry. В нашем случае не нужно решать такие проблемы, как перезапуск сервисов или обеспечение сбалансированности нагрузки. Не придется и задумываться об оповещениях о низкоуровневых действиях вроде обнаружения периодической выдачи контрольных событий; со всем перечисленным платформа справится самостоятельно.

Но в нашей визуализации все же остаются некоторые пробелы, касающиеся известных только нам искомых особенностей, специфичных для нашей архитектуры. Найти ее компоненты, информирующие об объеме ресурсов системы, не составляет особого труда. Они предоставляют информацию, доступную для нашего отслеживания или для проявления соответствующей реакции. Среда Spring Integration упрощает обработку событий от различных источников и объединение компонентов с помощью таких технологий рассылки сообщений, как RabbitMQ или Apache Kafka. Среда Spring Cloud Data Flow, построенная на основе Spring Integration, предоставляет DSL-язык, похожий на тот, что используется в оболочке Bash, который позволяет составлять произвольные потоки обработки, а затем управлять ими в распределенной структуре обработки, подобной YARN или Cloud Foundry. Более полные сведения можно получить в главе 10, где рассматривалась среда Spring Integration, и в главе 12, раскрывающей особенности среды Spring Cloud Data Flow. Последняя выступает идеальным набором инструментов для сборки каналов обработки, отвечающих за анализ данных, которые поступают из наших пользовательских источников событий.

Существует множество механизмов, пригодных к рассмотрению в качестве приглашений к действиям. Нашему положению можно позавидовать — у всех необходимых нам рычагов воздействия, запускающих решение многих классов проблем, имеются API. Для приведения в норму сбойного приложения можно использовать программный код. Предположим, у нас есть потребитель, затрачивающий существенное время на ответ на сообщение. Можно посмотреть на пропускную способность очереди в поставщике сообщений, чтобы решить вопрос с добавлением дополнительных потребителей и помочь тем самым более эффективному опустошению очереди и выдерживанию условий соглашения об уровне обслуживания (service-level agreement, SLA).

В исходном коде для этой главы есть несколько весьма полезных *источников* Spring Cloud Data Flow (компонентов, выдающих события в виде сообщений) и *приемников* (компонентов, выполняющих действия в ответ на события). Источник `rabbit-queue-metrics` отслеживает заданную очередь RabbitMQ и публикует информацию о нем: глубину очереди (сколько сообщений еще не обработано), счетчик потребителей (количество клиентов, отслеживающих сообщения на другой

стороне) и имя самой очереди. Приемник `cloudfoundry-autoscaler` реагирует на входящие сообщения (которые должны быть в виде числа) и добавляет или убавляет экземпляры заданного приложения, пока это число не окажется в пределах установленного диапазона. Вы определяете диапазон и необходимое количество. При условии регистрации пользовательских источников и приемников не составляет труда связать эти два понятия путем получения выходной информации от источника, извлечения из нее размера очереди, а затем отправки данного размера приемнику автомасштабирования `autoscaler` (пример 13.41).

Пример 13.41. Использование Spring Cloud Data Flow для автоматического масштабирования приложений на основе глубины очереди

```
rabbit-queue-metrics
  --rabbitmq.metrics.queueName=remediation-demo.remediation-demo -group |
 transform --expression=headers['queue-size'] |
 cloudfoundry-autoscaler --applicationName=remediation-rabbitmq-consumer
   --instance CountMaximum=10 --thresholdMaximum=5
```

Чтобы понять принцип поддержки средств восстановления работоспособности отказавшей системы, нужно разобраться с большим количеством входящих переменных. Если забрать соответствующие события, очистить их от всего ненужного и подключить к автоматическим обработчикам, то будут получены основы автоматизированной системы реагирования на инциденты для конкретных простых классов проблем. Среда Spring Cloud Data Flow создавалась с прицелом на эту категорию специализированных подходов к отслеживанию событий и реагированию на них. В исходный код для данной главы также включен компонент источника Spring Cloud Data для отслеживания таких показателей приложения Cloud Foundry, как степень загруженности оперативной памяти и использования жесткого диска. Подобные потоки восстановления работоспособности могут быть созданы вами на основе других показателей функционирования приложения.

Резюме

Мы пока так и остались на начальной стадии рассмотрения возможностей средств, упоминаемых в текущей главе, и если вы почувствовали усталость, то, значит, все в порядке! В системе, изначально предназначенной для работы в облачной среде, данный предмет приобретает *особую важность*, а архитектурные недочеты в возможностях отслеживания состояния влекут крупные неприятности. Их часто называют «проблемами второго дня», которые связаны с такими вещами, необходимость которых начинаешь осознавать только при переходе ПО к промышленному применению. И этот «второй день» будет переживаться куда менее болезненно, если об этих вещах хорошенько подумать в «первый день». Spring Boot, Spring Cloud и Cloud Foundry специально создавались для быстрого и простого интегрирования и поддержки требований, связанных с отслеживаемостью, с минимумом различных церемоний.

Вы, наверное, заметили, что многое из представленного в данной главе относилось к клиентским средствам с открытым кодом, но часть упомянутых опорных сервисов размещаются в качестве SaaS-предложений, за которые, как правило, нужно платить. Но это все откладывается на потом, а, как уже говорилось ранее, наша цель — никогда не запускать ПО, которое нельзя будет продать. Все же лучше удовлетворение этих требований, не свойственных прямому назначению нашего ПО, возложить на используемые платформы и разработки третьих сторон, для которых данные вопросы входят в их прямую компетенцию.

Займитесь вопросом извлечения всех этих сконцентрированных на решении производственных проблем требований в отдельную автоматическую конфигурацию Spring Boot, на основе которой выполняется сборка микросервисов в вашей организации. Можно даже создать собственные начальные зависимости и метааннотации. Тогда вам придется всего лишь однократно сконфигурировать порядок своей обработки регистрационных записей, показателей, трассировки и т. п., а затем просто добавлять к путям к классам соответствующее автоматическое конфигурирование. При использовании такой платформы, как Cloud Foundry, не составит особого труда установить нужные опорные сервисы для поддержки вашего приложения. Сочетание такой среды выполнения приложений, как Spring Boot (поддерживающей 12-факторные приложения), и платформы, подобной Cloud Foundry (поддерживающей 12-факторные операции), поможет вам быстро и безопасно перейти к эксплуатации собственного ПО.

14 Сервис-брокеры

Опорным называется сервис (базы данных, очереди сообщений, почтовые сервисы и т. д.), которым приложение пользуется по сети во время выполнения. Приложения Cloud Foundry задействуют такие сервисы, находя их указатели местоположения и регистрационные данные в переменной среды окружения по имени `VCAP_SERVICES`. Особенностью этого подхода является простота: любой язык может забрать эту переменную из среды и провести анализ встроенных данных в формате JSON, чтобы извлечь информацию о хостах, портах и регистрационных данных. Код приложения не должен вносить различий между локально предоставляемым, предлагаемым и управляемым платформой опорными сервисами. Подобная развязка упрощает миграцию приложений из одной среды в другую: следует просто переопределить значение переменной среды окружения, адаптировав его под нужную среду, и перезапустить приложение.

В среде Cloud Foundry приложения общаются со *всеми* сервисами через такую вот опосредованность. Cloud Foundry управляет наборами опорных сервисов и приложений. Оператор должен указать, какие приложения могут видеть контактную информацию конкретного опорного сервиса. Это называется привязкой сервиса. Одно приложение может быть связано с большим количеством опорных сервисов, и многие приложения — с одним опорным сервисом. В Cloud Foundry существует два способа предоставления сервисов, которые должны быть связаны с приложениями: применение сервис-брокеров и сервисов, определенных пользователем. В конечном итоге оба способа сводятся к упоминанию сервисов в качестве объектов в переменной среды `VCAP_SERVICES`.

Оператор может создать сервисы, определенные пользователем, если нужно, чтобы приложение общалось с посреднической программой или с каким-либо другим ресурсом за пределами платформы. Это особенно выгодно при подключении к имеющимся в организации устаревшим ресурсам наподобие экземпляра Ye Olde Sybase Instance, или универсального компьютера старого образца, или, конечно же, других REST API, представляющих простые приложения, запущенные где-либо на иных платформах.

При частом одновременном использовании ресурса несколькими пользователями и поддержке с его стороны работы в таком режиме имеет смысл вместо сервисов, определяемых пользователем, обратиться к услугам сервис-брокеров. Сами по себе последние являются приложениями, отвечающими за динамическое предоставление промежуточных программ и контактной информации платформе, которая затем привязывает ее к приложению. Сервис-брокер — это средство, используемое платформой для выставления по мере требования нового экземпляра сервиса. Сервис-брокер должен также управлять жизненным циклом экземпляра отдельно взятого сервиса, его созданием и уничтожением по мере необходимости. Платформа общается с сервис-брокером с помощью унифицированного REST API, который должен поддерживаться всеми сервис-брокерами.

Жизнь опорных сервисов

Торговая площадка сервисов в Cloud Foundry формируется *каталогом* сервиса, предлагаемого пользователям платформы. Оператор может согласованным образом выдать из каталога любой сервис. Приложения и их сервисы предоставляются платформой по единому принципу.

Для создания нового экземпляра сервиса служит команда `cf create-service` (или `cf cs`).

Пример 14.1. Создание нового экземпляра сервиса с помощью команды CF из арсенала CLI

```
NAME:
   create-service - Create a service instance
USAGE:
   cf create-service SERVICE PLAN SERVICE_INSTANCE [-c PARAMETERS_AS_JSON] [-t
TAGS]
```

В примере 14.1 можно увидеть, что для создания нового экземпляра сервиса нужны входные параметры. В качестве первого из них служит имя *сервиса*, предлагаемого каталогом торговой площадки. Команда `cf marketplace` приводит к извлечению списка сервисов, имеющих доступные планы для учетной записи данного пользователя.

Запустите эту команду на платформе Cloud Foundry, организованной Pivotal *(Pivotal Web Services)*, и увидите, что в каталоге торговой площадки имеется сервис `rediscloud` с разнообразными планами. Этот сервис, имеющийся на торговой площадке, позволяет создавать экземпляр сервиса, предоставляющий развернутую базу данных Redis, к которой можно привязывать свои приложения. Посмотрите внимательнее на каждый из планов применения сервиса для этой БД, которые предлагаются при использовании команды `cf marketplace -s rediscloud` (пример 14.2).

Пример 14.2. Список планов использования сервиса для развернутой базы данных Redis

```
$ cf marketplace -s rediscloud

Getting service plan information for service rediscloud for user...
OK

service plan   description          free or paid
100mb          Basic                paid
250mb          Mini                 paid
500mb          Small                paid
1gb            Standard             paid
2-5gb          Medium               paid
5gb            Large                paid
10gb           X-Large              paid
50gb           XX-Large             paid
30mb           Lifetime Free Tier   free
```

Здесь показаны различные платные (paid) уровни ресурсов и план `30mb`, предоставляемый в качестве бесплатного уровня. Создадим новый экземпляр сервиса Redis, использующий этот бесплатный уровень `30mb` (пример 14.3).

Пример 14.3. Создание нового экземпляра сервиса Redis

```
cf create-service rediscloud 30mb my-redis
```

После создания нового экземпляра сервиса (`my-redis`) можно привязать к нему существующее приложение. Представим, что у нас есть приложение по имени `user-api`, в котором содержится приложение Spring Boot, применяющее Redis для кэширования пользователей, хранящихся в базе данных MySQL. После привязки экземпляра сервиса Redis к нашему приложению Spring Boot мы можем подключиться к экземпляру этого сервиса в автоматическом режиме. Команда привязки к экземпляру сервиса имеет вид `cf bind-service` (пример 14.4).

Пример 14.4. Привязка экземпляра сервиса к приложению

```
NAME:
   bind-service - Bind a service instance to an app
USAGE:
   cf bind-service APP_NAME SERVICE_INSTANCE [-c PARAMETERS_AS_JSON]
```

Поскольку нам известно, что приложение, которое нужно привязать, называется `user-api`, а имя экземпляра сервиса — `my-redis`, то теперь мы можем привязать пользовательское приложение к нашему только что созданному экземпляру базы данных Redis (пример 14.5).

Пример 14.5. Привязка экземпляра сервиса my-redis к приложению user-api

```
cf bind-service user-api my-redis
```

При привязке приложения к экземпляру сервиса в среде Cloud Foundry сервис-брокер предоставит первому набор переменных среды окружения с описанием экземпляра сервиса. Результат для нашей привязки `my-redis` можно увидеть при просмотре переменных среды окружения приложения `user-api` с помощью команды `cf env user-api` (пример 14.6).

Пример 14.6. Показ переменных среды окружения для приложения user-api

```
$ cf env user-api

Getting env variables for app user-api in org user-org...
OK

System-Provided:
{
 "VCAP_SERVICES": {  ❶
  "rediscloud": [
   {
    "credentials": {  ❷
     "hostname": "pub-redis-9809.us-east-1-3.7.ec2.redislabs.com",
     "password": "not-a-real-password",
     "port": "17709"
    },
    "label": "rediscloud",
    "name": "my-redis",  ❸
    "plan": "30mb",  ❹
    "tags": [
     "Data Stores",
     "Data Store",
     "Caching",
     "Messaging and Queuing",
     "key-value",
     "caching",
     "redis"
    ]
   }
  ]
 }
}
```

❶ `VCAP_SERVICES` дает описание всех экземпляров сервисов, привязанных к приложению.

❷ Регистрационные данные, предоставленные для привязанного экземпляра сервиса.

❸ Имя экземпляра сервиса, созданного как `my-redis` из `rediscloud`.

❹ План, выбранный при создании `my-redis` с экземпляром бесплатного уровня.

Итак, мы предоставили новый экземпляр сервиса, для чего использовали сервис-брокер `rediscloud`, и этот экземпляр привязан к нашему приложению, что вполне согласуется с идеей опорного сервиса в рамках манифеста 12-факторного приложения (https://12factor.net/backing-services). Мы исследовали способ создания экземпляров сервиса и их привязки с помощью каталога сервиса на торговой площадке в среде Cloud Foundry. Мы также увидели, как сервис-брокеры вставляют подробности подключения в развертывание приложения, используя предоставленную системой переменную среды окружения `VCAP_SERVICES`.

Вид со стороны платформы

Теперь исследуем механизм, стоящий за платформой Cloud Foundry, который позволит разрабатывать собственные пользовательские сервис-брокеры с помощью Spring Boot. Платформа состоит из нескольких модулей веб-сервисов, общающихся через HTTP, которые походят на ряд архитектур распределенных систем, рассмотренных в данной книге. Композиция этих сервисов представлена в качестве одного, цельного REST API, Cloud Controller API. Он является первичным интерфейсом для клиентов, служащим для инициирования управления виртуальной архитектурой, предоставленной поставщиком используемого облака. Это первичное соглашение, в котором описываются функции платформы Cloud Foundry.

Cloud Controller также обслуживает базу данных для множества ресурсов предметной области, которые применяются при взаимодействии с Cloud Foundry, таких как организации, пространства, роли пользователя, определения сервиса, экземпляры сервиса, приложения и др. Поскольку Cloud Controller владеет как определениями сервиса, так и его экземплярами, то является тем самым модулем Cloud Foundry, который позволяет управлять жизненным циклом внешних сервис-брокеров и может приобретать форму базы данных и других промежуточных сервисов.

API сервис-брокера по своей сути — REST API, имеющий описание ожиданий Cloud Controller, которые касаются сторонних приложений сервис-брокера. Как правило, сервис-брокеры не являются частью выпуска Cloud Foundry. Брокеры всего лишь добавляются после выпуска Cloud Foundry. Они прикрепляются в качестве опорных сервисов, которые должны быть зарегистрированы с помощью Cloud Controller API.

В сервис-брокере дается описание каталога сервисов, которые он предоставляет. Это показывается в торговой площадке Cloud Foundry, которую можно увидеть, воспользовавшись командой `cf marketplace`. Каталог содержит описание одного или нескольких сервисов и их планов. План сервиса характеризуется различными уровнями сервиса, зачастую связанными с разными ценами. Оператор может

принять решение о создании нового экземпляра сервиса, указав его имя и связанный с ним план. Для этого используется команда `cf create-service`. Созданный сервис должен представлять запущенную виртуальную машину или же инициализированный ресурс. Операторы дополнительно могут привязать экземпляр сервиса к своему приложению с помощью команды `cf bind-service`. Здесь сервис-брокеру нужно предоставить регистрационные данные, которые приложение сможет использовать для подключения к выделенному сервису. Сервис-брокер должен также управлять удалением экземпляров сервиса и их привязками.

Конечные точки, наличие которых ожидает сервис-брокер, показаны в табл. 14.1.

Таблица 14.1. Маршруты сервис-брокера

Описание	Маршрут	Метод
Чтение каталога с описанием планов, которые можно получить от сервис-брокера	/v2/catalog	GET
Удаление привязки сервис-брокера к приложению	/v2/service_instances/{instanceId}/service_bindings/{bindingId}	DELETE
Создание нового экземпляра сервиса с привязкой приложения к сервису	/v2/service_instances/{instanceId}/service_bindings/{bindingId}	PUT
Извлечение состояния последней операции, выполненной сервис-брокером	/v2/service_instances/{instanceId}/last_operation	GET
Извлечение экземпляра сервиса	/v2/service_instances/{instanceId}	GET
Создание нового экземпляра сервиса	/v2/service_instances/{instanceId}	PUT
Удаление экземпляра сервиса	/v2/service_instances/{instanceId}	DELETE
Обновление экземпляра сервиса	/v2/service_instances/{instanceId}	PATCH

Реализация сервис-брокера с помощью Cloud Foundry Service Broker

В данном разделе мы извлечем уроки из всего изученного в этой книге, чтобы создать сервис-брокер с помощью Spring Boot. Создание его — важная составляющая расширения и подгонки под свои нужды платформы Cloud Foundry. Существуют программы с открытым кодом на множестве различных языков и платформ, которые демонстрируют способы создания сервис-брокера. Мы же воспользуемся имеющимся в среде Spring Cloud проектом Cloud Foundry Service Broker, чтобы проделать небольшую работу по выполнению несущественных требований, которые касаются создания сервис-брокера, а именно по реализации жизненного цикла экземпляров сервиса и их привязкам.

Простой сервис-брокер Amazon S3

Среда Amazon Web Services (AWS) предоставляет весьма богатую экосистему сервисов, которые может задействовать наше приложение, даже если мы запустим его на другой платформе. Amazon S3 отлично подходит для сохранения данных. Основной принцип использования облачных приложений заключается в отказе от применения состояний в их экземплярах. Мы не можем быть уверенными в том, что экземпляр виртуальной машины, на котором запущено наше приложение, будет всегда одним и тем же, и не можем обеспечить постоянное наличие той файловой системы, которая использовалась для нашего экземпляра приложения. S3 предоставляет способ прикрепления к нашим приложениям хранилища данных в качестве опорного сервиса. Amazon S3 не является файловой системой, как может быть воспринято вашей операционной системой, поэтому не станет работать в качестве прямой замены `java.io.File` из арсенала JDK. Это средство имеет собственный API. Он служит тем же целям, поэтому облегчение взаимодействия с Amazon S3 принесет пользу при миграции существующих приложений в облако. Сервис-брокер предоставляет превосходный способ интегрирования ранее созданных (но иногда просто необходимых) сервисов наподобие FTP, файловых систем и сервисов электронной почты или же, конечно, любого имеющегося в вашей организации сервиса, способного обслуживать сразу несколько клиентов, у которого пока нет поддержки со стороны платформы.

Реализация сервис-брокера состоит из решения двух задач: предоставления ресурсов, предлагаемых сервисом, и реализации REST API сервис-брокера. Имеющийся в среде Spring Cloud проект Cloud Foundry Service Broker обеспечит REST API, если мы обеспечим наличие трех ключевых компонентов:

- экземпляра bean-компонента `Catalog`, дающего облачному контроллеру описание доступных сервисов, предоставляемых этим сервис-брокером;
- реализации сервиса `ServiceInstanceService` для управления созданием и удалением его экземпляров;
- реализации сервиса `ServiceInstanceBindingService` для создания и удаления привязок его экземпляра.

Каталог сервисов

Первый шаг к реализации сервис-брокера — создание *каталога сервисов*, дающего описание того, чем этот брокер будет заниматься. Каталог также выдаст коллекцию планов сервисов, которую можно будет использовать для описания различных аспектов порядка создания и работы опорных сервисов. Возьмем, к примеру, базу данных, предоставляемую в качестве сервиса через сервис-брокер. Наш брокер Amazon S3 выдаст лишь один план, являющийся участком памяти (или бакетом,

bucket) S3 с логически бесконечным (но, возможно, не с финансовой точки зрения!) файловым хранилищем (пример 14.7).

Пример 14.7. Соглашение по интерфейсу для каталога сервисов, принадлежащего брокеру

```
package cnj;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.servicebroker.model.Catalog;
import org.springframework.cloud.servicebroker.model.Plan;
import org.springframework.cloud.servicebroker.model.ServiceDefinition;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.Collections;
import java.util.List;
import java.util.Map;

@Configuration
class CatalogConfiguration {

 private final String spacePrefix;

 @Autowired
 CatalogConfiguration(  ❶
  @Value("${vcap.application.space_id:${USER:}}") String spaceId) {
   this.spacePrefix = spaceId + '-';
 }

 @Bean
 Catalog catalog() {
  Plan basic = buildBasicPlan();  ❷
  ServiceDefinition definition = buildS3ServiceDefinition(basic);  ❸
  return new Catalog(Collections.singletonList(definition));
 }

 private ServiceDefinition buildS3ServiceDefinition(Plan basic) {
  String description = "Provides AWS S3 access";
  String id = this.spacePrefix + "1489291412183";
  String name = "s3-service-broker";
  boolean bindable = true;
  List<Plan> plans = Collections.singletonList(basic);
  return new ServiceDefinition(id, name, description, bindable, plans);
 }

 private Plan buildBasicPlan() {
  String planName = "basic";
```

```
  String planDescription = "Amazon S3 bucket with unlimited storage";
  boolean free = true;
  boolean bindable = true;
  String planId = this.spacePrefix + "249722552510577";
  Map<String, Object> metadata = Collections.singletonMap("costs",
   Collections.singletonMap("free", true));
  return new Plan(planId, planName, planDescription, metadata, free,
   bindable);
 }
}
```

1. У сервиса брокера и у плана должен быть уникальный идентификатор, имеющий глобальный характер для всего экземпляра `CloudFoundry`, в котором происходит развертывание. Поэтому здесь мы применяем логику ввода в качестве префикса указателя на пространство Cloud Foundry (или как минимум на имя пользователя) для стабильных статичных в иных случаях идентификаторов. Значения являются произвольными и выдаются заранее. Если предположить, что в каждом отдельно взятом пространстве развертывается только один сервис-брокер (это весьма допустимо), то каталог сервисов должен быть уникальным. При необходимости обновить сервис-брокер будет полезно воспользоваться стабильными идентификаторами сервисов. В противном случае можно просто применить универсальный уникальный идентификатор (UUID).
2. Создание простого плана экземпляра (бесплатного).
3. Регистрация плана в качестве части нашего определения одного сервиса. Ничто не препятствует размещению в одном сервис-брокере нескольких определений сервисов (к примеру, по одному для каждого интересующего сервиса в каталоге Amazon Web Services!).

Конфигурация каталога жестко задана в приложении из соображений краткости, хотя нет никаких препятствий для его обслуживания во внешней конфигурации, возможно, в сочетании с Spring Cloud Config Server.

Управление экземплярами сервиса

Сервис-брокер Cloud Foundry Service Broker, имеющийся в среде Spring Cloud, ожидает реализацию интерфейса `ServiceInstanceService` (пример 14.8).

Пример 14.8. Определение ServiceInstanceService

```
public interface ServiceInstanceService {
  CreateServiceInstanceResponse createServiceInstance(
    CreateServiceInstanceRequest r);
  UpdateServiceInstanceResponse updateServiceInstance(
    UpdateServiceInstanceRequest r);
```

```
  DeleteServiceInstanceResponse deleteServiceInstance(
    DeleteServiceInstanceRequest r);
  GetLastServiceOperationResponse getLastOperation(
    GetLastServiceOperationRequest r);
}
```

Большинство из этих методов должны быть весьма простыми, за исключением `GetLastServiceOperationRequest`. Он используется для возвращения состояния *асинхронной* операции. По умолчанию сервис-брокеры работают в синхронном режиме: сначала возвращается ответ на вызов к конечной точке REST, а затем работа считается завершенной. Но так бывает не всегда: представьте сервис-брокер, предоставляющий кластеры Hadoop либо Spark, или сервис-брокер, пытающийся демонтировать такой кластер в вызове удаления. Эта работа могла бы занять гораздо больше времени, чем разумный установленный максимальный период ожидания, разрешенный сервис-брокером. В нашем случае можно ответить, что на вызов сервиса всегда возвращается успешный ответ; вызовы в адрес Amazon Web Services незатратны по времени.

Наша реализация `ServiceInstanceService` будет реагировать на эти запросы и сохранять соответствующую информацию в базе данных SQL (конкретнее, в MySQL), чтобы можно было сопоставить запросы к фактическому экземпляру сервиса с ресурсами, управляемыми сервис-брокером (пример 14.9).

Пример 14.9. Определение ServiceInstance

```
package cnj;

import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;
import lombok.ToString;

import org.springframework.cloud.servicebroker.model.CreateServiceInstanceRequest;
import org.springframework.cloud.servicebroker.model.DeleteServiceInstanceRequest;
import org.springframework.cloud.servicebroker.model.UpdateServiceInstanceRequest;

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

@Entity
❶
@Data
❷
@NoArgsConstructor
@ToString
@EqualsAndHashCode
class ServiceInstance {
```

```
 @Id

 private String id;

 private String serviceDefinitionId;

 private String planId;

 private String organizationGuid;

 private String spaceGuid;

 private String dashboardUrl;

 private String username, accessKeyId, secretAccessKey;

 ❸
 public ServiceInstance(CreateServiceInstanceRequest request) {
  this.serviceDefinitionId = request.getServiceDefinitionId();
  this.planId = request.getPlanId();
  this.organizationGuid = request.getOrganizationGuid();
  this.spaceGuid = request.getSpaceGuid();
  this.id = request.getServiceInstanceId();
 }

 public ServiceInstance(DeleteServiceInstanceRequest request) {
  this.id = request.getServiceInstanceId();
  this.planId = request.getPlanId();
  this.serviceDefinitionId = request.getServiceDefinitionId();
 }

 public ServiceInstance(UpdateServiceInstanceRequest request) {
  this.id = request.getServiceInstanceId();
  this.planId = request.getPlanId();
 }
}
```

❶ `ServiceInstance` является объектом JPA...

❷ ...который использует принадлежащий проекту Lombok конструктор `@Data`, чтобы убрать большой объем работы по созданию однообразного кода для геттеров и сеттеров, конструктор без аргументов, метод `toString` и пару `equals`/`hashCode`...

❸ ...с разнообразными копиями конструкторов, поддерживающих создание, удаление и обновление запросов.

Необходимо сохранить подробности в базе данных SQL (в текущем проекте — в MySQL). Хранилище Spring Data JPA упрощает работу, позволяя двигаться дальше (пример 14.10).

Пример 14.10. Определение ServiceInstanceRepository

```
package cnj;

import org.springframework.data.jpa.repository.JpaRepository;

interface ServiceInstanceRepository extends
 JpaRepository<ServiceInstance, String> {
}
```

При создании нового экземпляра сервиса следует объединиться с Amazon Web Services, задействуя предоставляемый этой платформой Java SDK для управления созданием как IAM-пользователя, так и бакета S3. Это любимое занятие нашего брокера. Основная часть данной логики находится в реализации пользовательского сервиса в кодовой базе `S3Service`. Мы не станем здесь перепечатывать этот код; посмотрите исходный код, прилагаемый к книге, и найдете в нем множество однообразных логических построений, завязанных на Amazon Web Services Java SDK. Единственное, в чем стоит разобраться, — Amazon Web Services Java SDK требует авторизации, которая конфигурируется следующим образом (пример 14.11).

Пример 14.11. Класс конфигурации, содержащий определения bean-компонентов для аутентификации клиента AmazonS3

```
package cnj.s3;

import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.identitymanagement.
AmazonIdentityManagementClient;
import com.amazonaws.services.s3.AmazonS3Client;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
class S3Configuration {

 @Bean
 AmazonS3Client amazonS3Client(BasicAWSCredentials credentials) {
  return new AmazonS3Client(credentials);
 }

 @Bean
 AmazonIdentityManagementClient identityManagementClient(
  BasicAWSCredentials awsCredentials) {
  return new AmazonIdentityManagementClient(awsCredentials);
 }

 ❶
 @Bean
 BasicAWSCredentials awsCredentials(
  @Value("${aws.access-key-id}") String awsAccessKeyId,
  @Value("${aws.secret-access-key}") String awsSecretAccessKey) {
```

```
  return new BasicAWSCredentials(awsAccessKeyId, awsSecretAccessKey);
 }

 @Bean
 S3Service s3Service(AmazonIdentityManagementClient awsId, AmazonS3Client s3)
 {
  return new S3Service(awsId, s3);
 }
}
```

❶ Чтобы описанное заработало, нужно сконфигурировать токен доступа Amazon Web Services и секретный ключ доступа к Amazon Web Services. В наших средах непрерывной интеграции это обеспечивается в виде переменной среды окружения.

С помощью нашего `S3Service` реализация `ServiceInstanceService` выполняется довольно быстро. В случае использования S3 нам практически нечего предоставлять — в среде Amazon Web Services уже есть инициированный, запущенный и готовый к применению бакет S3. Поэтому основная часть кода экземпляра сервиса занимается созданием пользователя S3, чья идентичность была сохранена. Позже, при привязке приложения к предоставленному экземпляру, мы вспомним сохраненную информацию и выдадим ее контроллеру облака, чтобы она была доступна в среде запущенного приложения. Это значит, что все привязанные приложения будут иметь одни и те же регистрационные данные. Указанный факт может совпадать или не совпадать с вашими желаниями, но такое поведение вполне разумно при наличии доверия к сторонам, разрабатывающим приложение (пример 14.12).

Пример 14.12. Определение DefaultServiceInstanceService

```
package cnj;

import cnj.s3.S3Service;
import cnj.s3.S3User;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.servicebroker.exception.
ServiceBrokerException;
import org.springframework.cloud.servicebroker.model.*;
import org.springframework.cloud.servicebroker.service.ServiceInstanceService;
import org.springframework.stereotype.Service;

@Service
class DefaultServiceInstanceService implements ServiceInstanceService {

 private final S3Service s3Service;

 private final ServiceInstanceRepository instanceRepository;

 private Log log = LogFactory.getLog(getClass());

 @Autowired
```

```
DefaultServiceInstanceService(S3Service s3Service,
 ServiceInstanceRepository instanceRepository) {
 this.s3Service = s3Service;
 this.instanceRepository = instanceRepository;
}

@Override
public CreateServiceInstanceResponse createServiceInstance(
 CreateServiceInstanceRequest request) {
 if (!this.exists(request.getServiceInstanceId())) {
  ServiceInstance si = new ServiceInstance(request);
  S3User user = s3Service.createS3UserAndBucket(si.getId());  (1)
  si.setSecretAccessKey(user.getAccessKeySecret());
  si.setUsername(user.getUsername());
  si.setAccessKeyId(user.getAccessKeyId());
  this.instanceRepository.save(si);
 }
 else {
  this.error("could not create serviceInstance "
   + request.getServiceInstanceId());
 }
 return new CreateServiceInstanceResponse();
}

@Override
public DeleteServiceInstanceResponse deleteServiceInstance(
 DeleteServiceInstanceRequest request) {
 String sid = request.getServiceInstanceId();
 if (this.exists(sid)) {
  ServiceInstance si = this.instanceRepository.findOne(sid);
  (2)
  boolean deleteSucceeded = this.s3Service.deleteBucket(si.getId(),
   si.getAccessKeyId(), si.getUsername());
  if (!deleteSucceeded) {
   log.error("could not delete the S3 bucket for service instance " + sid);
  }
  this.instanceRepository.delete(si.getId());
 }
 else {
  this.error("could not delete the S3 service instance " + sid);
 }
 return new DeleteServiceInstanceResponse();
}

(3)
@Override
public UpdateServiceInstanceResponse updateServiceInstance(
 UpdateServiceInstanceRequest request) {
 String sid = request.getServiceInstanceId();
```

```
  if (this.exists(sid)) {
   ServiceInstance instance = this.instanceRepository.findOne(sid);
   this.instanceRepository.delete(instance);
   this.instanceRepository.save(new ServiceInstance(request));
  }
  else {
   this.error("could not update serviceInstance " + sid);
  }
  return new UpdateServiceInstanceResponse();
 }

 @Override
 public GetLastServiceOperationResponse getLastOperation(
  GetLastServiceOperationRequest request) {
  return new GetLastServiceOperationResponse()
   .withOperationState(OperationState.SUCCEEDED);
 }

 private void error(String msg) {
  throw new ServiceBrokerException(msg);
 }

 private boolean exists(String serviceInstanceId) {
  return instanceRepository.exists(serviceInstanceId);
 }
}
```

❶ Основная часть данного метода относится к служебным действиям, за исключением этой строки, где выполняется взаимодействие с Amazon Web Services API с помощью нашего `S3Service` в целях создания бакета S3 и пользователя `S3User`, имеющего доступ к этому бакету.

❷ Операция удаления отменяет создание, удаляя пользователя и бакет.

❸ Операция обновления ничего не делает для фактически созданного бакета S3 или пользователя, она просто обновляет сохраненные метаданные о самом экземпляре сервиса.

Результат работы опорного сервиса — вставка в контейнер приложения, имеющего вид переменных среды окружения, всей необходимой информации о порядке его использования. Это именно тот процесс, который в Cloud Foundry мы называли *привязкой к экземпляру сервиса*. Результатом ее является то, что контактная информация о способах применения сервиса вставляется в контейнер приложения. Когда приложение помещается в контейнер, а затем запускается, у него появляется возможность определить местонахождение переменных среды окружения в целях получения информации о подключении к экземпляру сервиса. Затем приложение будет использовать эти данные для нахождения запущенного экземпляра сервиса и подключения к нему. Посмотрим, как наш экземпляр сервиса привязывается к приложению поддержки.

Привязки сервисов

Зачастую довольно трудно провести разделительную линию между предложением функции в качестве сервиса с помощью платформы и ее реализацией в каждом приложении. Существует устоявшееся простое правило для определения того, в каких случаях следует делегировать функцию в вашем приложении опорному сервису благодаря платформе.

При необходимости реализовать одну и ту же функциональную возможность во всех ваших приложениях ее следует предоставлять в *качестве сервиса* с помощью платформы.

Цель создания настоящего облачного приложения — сокращение времени, затрачиваемого на любую разработку, которая не относится к бизнес-логике приложения. У платформы более широкие задачи, чем быть оперативным средством запуска приложений в облаке, — это механизм, удаляющий код, который превращает приложение в устаревшее, вместо этого предоставляя его в виде заменяемых сервисов. Например, в архитектуре многих приложений может понадобиться сохранение файлов. Предоставление сервис-брокера, способного снабдить приложения возможностью подключаться к бакету Amazon S3, поможет при создании каждого приложения не тратить время на заботы по реализации пользовательского кода, позволяющего сохранять файлы. Единственное, что нужно будет сделать разработчикам, — привязать экземпляр сервиса и приступить к сохранению и извлечению файлов.

В Cloud Foundry Service Broker, имеющемся в среде Spring Cloud, такая функциональная возможность предоставляется путем реализации сервиса `ServiceInstanceBindingService` (пример 14.13).

Пример 14.13. Определение ServiceInstanceBindingService

```
public interface ServiceInstanceBindingService {

  CreateServiceInstanceBindingResponse createServiceInstanceBinding(
     CreateServiceInstanceBindingRequest r);

  void deleteServiceInstanceBinding(DeleteServiceInstanceBindingRequest r);
}
```

Что касается `ServiceInstanceBindingService`, то следует записать информацию о привязке приложения. Это делается с помощью еще одного объекта JPA по имени `ServiceInstanceBinding` (пример 14.14).

Пример 14.14. Определение ServiceInstanceBinding

```
package cnj;

import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
```

```
import javax.persistence.Id;

@Entity
@Data
@NoArgsConstructor
public class ServiceInstanceBinding {

 @Id
 private String id;

 private String serviceInstanceId, syslogDrainUrl, appGuid;

 public ServiceInstanceBinding(String id, String serviceInstanceId,
  String syslogDrainUrl, String appGuid) {
  this.id = id;
  this.serviceInstanceId = serviceInstanceId;
  this.syslogDrainUrl = syslogDrainUrl;
  this.appGuid = appGuid;
 }
}
```

Мы сохраним `ServiceInstanceBinding`, воспользовавшись еще одним хранилищем Spring Data JPA `ServiceInstanceBindingRepository` (пример 14.15).

Пример 14.15. Определение ServiceInstanceBindingRepository для управления экземплярами сервиса

```
package cnj;

import org.springframework.data.jpa.repository.JpaRepository;

public interface ServiceInstanceBindingRepository extends
 JpaRepository<ServiceInstanceBinding, String> {

}
```

Выполнить эту часть работы в нашей реализации `ServiceInstanceBindingService` помогут два хранилища. `DefaultServiceInstanceBindingService` нуждается в воссоздании в памяти сохраненных регистрационных данных S3, извлекаемых из БД, и в доставке их клиенту (пример 14.16).

Пример 14.16. Определение DefaultServiceInstanceBindingService

```
package cnj;

import org.springframework.beans.factory.annotation.Autowired;

//@formatter:off
import org.springframework.cloud.servicebroker.exception.
        ServiceInstanceBindingDoesNotExistException;
import org.springframework.cloud.servicebroker.exception.
        ServiceInstanceBindingExistsException;
import org.springframework.cloud.servicebroker.model
        .CreateServiceInstanceAppBindingResponse;
```

```
import org.springframework.cloud.servicebroker.model
        .CreateServiceInstanceBindingRequest;
import org.springframework.cloud.servicebroker.model
        .CreateServiceInstanceBindingResponse;
import org.springframework.cloud.servicebroker.model
        .DeleteServiceInstanceBindingRequest;
import org.springframework.cloud.servicebroker.service
        .ServiceInstanceBindingService;
//@formatter:on

import org.springframework.stereotype.Service;

import java.util.HashMap;
import java.util.Map;

@Service
class DefaultServiceInstanceBindingService implements
 ServiceInstanceBindingService {

 private final ServiceInstanceBindingRepository bindingRepository;

 private final ServiceInstanceRepository instanceRepository;

 @Autowired
 DefaultServiceInstanceBindingService(ServiceInstanceBindingRepository sibr,
  ServiceInstanceRepository sir) {
  this.bindingRepository = sibr;
  this.instanceRepository = sir;
 }

 @Override
 public CreateServiceInstanceBindingResponse createServiceInstanceBinding(
  CreateServiceInstanceBindingRequest request) {
  String bindingId = request.getBindingId();
  {
   ServiceInstanceBinding binding;
   if ((binding = this.bindingRepository.findOne(bindingId)) != null) {
    throw new ServiceInstanceBindingExistsException(
     binding.getServiceInstanceId(), binding.getId());
   }
  }

  ❶
  ServiceInstance serviceInstance = this.instanceRepository.findOne(request
   .getServiceInstanceId());

  String username = serviceInstance.getUsername();
  String secretAccessKey = serviceInstance.getSecretAccessKey();
  String accessKeyId = serviceInstance.getAccessKeyId();

  ❷
  Map<String, Object> credentials = new HashMap<>();
```

```
  credentials.put("bucket", username);
  credentials.put("accessKeyId", accessKeyId);
  credentials.put("accessKeySecret", secretAccessKey);

  Map<String, Object> resource = request.getBindResource();
  String appGuid = String.class.cast(resource.getOrDefault("app_guid",
   request.getAppGuid()));

  ❸
  ServiceInstanceBinding binding = new ServiceInstanceBinding(bindingId,
   request.getServiceInstanceId(), null, appGuid);

  this.bindingRepository.save(binding);

  ❹
  return new CreateServiceInstanceAppBindingResponse()
   .withCredentials(credentials);
 }

 @Override
 public void deleteServiceInstanceBinding(
  DeleteServiceInstanceBindingRequest request) {
  String bindingId = request.getBindingId();
  if (this.bindingRepository.findOne(bindingId) == null) {
   throw new ServiceInstanceBindingDoesNotExistException(bindingId);
  }
  this.bindingRepository.delete(bindingId);
 }
}
```

❶ Этот метод ведет поиск существующих экземпляров сервиса (созданных в реализации `ServiceInstanceService`)...

❷ ...и возвращает `Map<K,V>`, где находится информация, которая будет востребована привязанным приложением. Содержимое этого отображения носит абсолютно произвольный характер — сюда можно помещать что угодно. Данное содержимое превратится в JSON-структуру, которая будет прочитана потребляющим ее приложением.

❸ По готовности следует записать полученную информацию о привязке в базу данных...

❹ ...и возвратить `CreateServiceInstanceAppBindingResponse`, где содержатся регистрационные данные.

Важно помнить: как только мы создали на брокере ServiceInstanceBinding, Cloud Controller становится системой записи привязок, превращающей регистрационные данные в неизменяемый объект, который может быть изменен только путем повторного создания привязки.

Обеспечение безопасности сервис-брокера

Cloud Controller предполагает, что API сервис-брокера S3 защищено базовой аутентификацией HTTP. Каждый запрос, сделанный из Cloud Controller к этому брокеру, будет содержать заголовок `Authentication`, включающий закодированные имя пользователя и пароль, которые брокер должен проверить.

Приложение Spring Boot для брокера S3 задействует для базовой идентификации HTTP среду Spring Security. Эту конфигурацию можно полностью переопределить, но целесообразнее было применить удобные свойства `security.user.name` и `security.user.password` автоматического конфигурирования, имеющиеся в Spring Boot, чтобы жестко задать в качестве имени пользователя и пароля значение `admin`. Это, конечно же, можно переопределить с помощью переменных среды окружения, например `SECURITY_USER_PASSWORD` и `SECURITY_USER_NAME`.

В данном примере задействован диспетчер `AuthenticationManager`, обеспеченный представлением пользователей, находящимся в оперативной памяти. Это исходное поведение практически всех настраиваемых по умолчанию приложений Spring Boot с имеющейся в путях к классам записью `spring-boot-starter-security`. В более сложном примере сервис-брокера может потребоваться создание средства администрирования, позволяющего операторам платформы управлять установками брокера, возможно, сохраненными в базе данных.

При создании сервис-брокеров, расширяющих платформу, безопасность ставится во главу угла. Важно, чтобы доступ к API брокера ограничивался только обменом данными с Cloud Controller. Так как в последнем поддерживается только базовая аутентификация HTTP, если бы брокер должен был обеспечить доступность своих API через открытый хост или маршрут, то это стало бы серьезным изъяном безопасности, через который можно было бы раскрыть внутреннее содержимое базовой инфраструктуры, управляемой платформой.

Итак, все готово! Приложение, как и предусматривалось, является работоспособным сервис-брокером. Нужно только развернуть его и сообщить о нем среде Cloud Controller.

Развертывание

После создания сервис-брокера Amazon S3 можно приступить к тестированию работоспособности, развернув его в распределенном выполнении Cloud Foundry. Сервис-брокер можно сконфигурировать для запуска в любой дистрибуции Cloud Foundry.

Выпуск с помощью BOSH

Развертывать пользовательские сервис-брокеры Cloud Foundry можно несколькими способами. Популярный вариант выпуска заключается в применении такого средства, как BOSH (*внешняя оболочка* BOSH). Это инструмент выпуска, предназначенный для управления развертываниями. Для связи с платформами, которые представляют собой инфраструктуры в качестве сервисов (Infrastructure as a Service, IaaS), подобные Amazon Web Services, Google Compute Engine, OpenStack, vSphere и Microsoft Azure, оно может использовать множество разнообразных интерфейсов, предоставляемых облаком — *Cloud Provider Interfaces* (CPIs). Интерфейс CPI является выполняемым компонентом архитектуры BOSH, который служит в качестве уровня трансляции для обмена данными с API, предоставляемыми поставщиком IaaS. CPI получает собственные API IaaS и транслирует их сравнимые функции в общий язык предметной области, применяемый в BOSH для описания комплексных развертываний. BOSH управляет виртуальными ресурсами платформы IaaS и гарантирует, что фактическое состояние системы управляет описываемым состоянием. Этим BOSH отличается от средств управления конфигурацией, таких как Puppet или Chef, которые поддерживают конвергентную инфраструктуру. Эти средства предпринимают попытку получить существующие узлы и модифицировать их в соответствии с предписанным состоянием. BOSH поддерживает неизменяемую инфраструктуру — каждый раз воссоздает мир с нуля, начиная с образов виртуальных машин.

В основе BOSH заложена идея выпуска. Последний должен включать исходные файлы, файлы конфигурации, сценарии установки и т. д. Если вкратце, в нем содержится все необходимое для воспроизводимого выпуска заданного компонента. В выпуск Apache Zookeeper может входить исходный код для Apache Zookeeper, его настройки по умолчанию и сценарии инициализации.

Не все выпуски одинаковы. К примеру, в среде комплексного тестирования может быть оправданным наличие двух узлов Zookeeper, а в эксплуатационной среде — наличие десятков узлов. Параметризация BOSH-выпусков осуществляется в манифестах развертывания, где дается описание количества требуемых узлов, свойств конфигурации, которые должны передаваться каждому ресурсу узла, и т. д.

BOSH stemcells («стволовые клетки») — базовые образы операционной системы, поверх которых BOSH надстраивает уровни всего остального. На официальном сайте BOSH (http://bosh.io/docs/) содержится множество stemcells.

BOSH — идеальное средство управления развертыванием комплексных систем. Если нечто должно измениться, то обычно вносится изменение в манифест развертывания, создается новый выпуск, а затем блоку управления BOSH сообщается о новом выпуске. Он выполнит новое развертывание всего этого вычислительного мира в соответствии с данным выпуском.

А зачем применять BOSH для выпуска пользовательского сервис-брокера вместо развертывания приложения с помощью самой среды Cloud Foundry? К тому же разве не разумнее будет вести разработку приложений на платформе, чтобы иметь возможности выполнять на ней и развертывание? Если ответить вкратце, то все зависит от виртуализированных вычислительных ресурсов, которыми вы намереваетесь управлять, задействуя ваш пользовательский брокер.

Наш сервис-брокер с помощью REST API работает в целях непосредственного обмена данными с Amazon Web Services IaaS. Он отвечает не за установку самого экземпляра S3, а только за общение с ним.

Если нашему брокеру требуется справляться всего лишь с одним IaaS, то для управления его ресурсами будет вполне достаточно использовать API IaaS на основе HTTP. Это похоже на то, что мы делаем с нашим сервис-брокером Amazon S3, и, как следствие, не нужно создавать BOSH-выпуск. Польза применения BOSH заключается в том, что данное средство помогает управлять выпусками сервис-брокера, а им, в свою очередь, приходится управлять всем уровнем IaaS, на котором запущена дистрибуция Cloud Foundry.

Основная причина, по которой сервис-брокеры объединяются в BOSH-выпуске, кроется в использовании такой разновидности брокеров, которым необходимо управлять экземплярами сервисов, состоящих из больших распределенных систем. Эта разновидность сервис-брокеров будет нуждаться в более детализированном управлении устройствами хранения данных и ресурсами сети, предоставляемыми уровнем IaaS. Что же касается нашего брокера S3, то не нужно заботиться о создании BOSH-выпуска, поскольку нашему брокеру не требуется общение с IaaS-уровнем. Поэтому можно обходиться с брокером как с развертыванием обычного приложения, используя имеющийся в среде Cloud Foundry интерфейс Cloud Controller API.

Выпуск с помощью Cloud Foundry

Развернем наш сервис-брокер в Cloud Foundry. Ему понадобится экземпляр MySQL, привязанный в качестве `s3-service-broker-db`, поэтому, прежде чем приступить к развертыванию приложения, нужно предусмотреть наличие такого экземпляра. Если используется PCFDev или Pivotal Web Services, то для этих целей есть управляемая Pivotal база данных MySQL, которая называется `p-mysql`. Интересно, что спецификации планов Pivotal Web Services и PCFDev отличаются друг от друга. Если применяется первая, то должен сработать следующий код, но вам следует убедиться в этом с помощью команды `cf marketplace` (пример 14.17). На момент написания этих строк существовало также предложение Pivotal Web Services под названием ClearDB, которое предлагало MySQL.

Пример 14.17. Предоставление экземпляра p-mysql в Pivotal Web Services

```
cf create-service p-mysql 100mb s3-service-broker-db
```

Если скомпилировать код сервис-брокера, то останется только поместить его в среду Cloud Foundry, используя команду `cf push` в том же каталоге, в котором находится сам сервис-брокер. Вы найдете файл `manifest.yml`, содержащий описание всего необходимого для запуска этого приложения (пример 14.18).

Пример 14.18. Файл manifest.yml с описанием порядка развертывания сервис-брокера в Cloud Foundry

```
---
applications:
- name: s3-service-broker
  path: ./target/s3-service-broker.jar
  buildpack: https://github.com/cloudfoundry/java-buildpack.git
  memory: 1024M
  instances: 1
  timeout: 180
  host: s3-service-broker-${random-word} (1)
  env:
    SPRING_PROFILES_ACTIVE: cloud
    AWS_ACCESS_KEY_ID: replace
    AWS_SECRET_ACCESS_KEY: replace
  services:
    - s3-service-broker-db
```

(1) Токен `${random-word}` указывает Cloud Foundry на необходимость создания произвольной последовательности слов для предотвращения конфликтов URI. Это особенно удобно в многопользовательской среде, подобной Pivotal Web Services.

Файл `manifest.yml` нуждается в корректировке с предоставлением значений для Amazon WebServices `AWS_ACCESS_KEY_ID` и `AWS_SECRET_ACCESS_KEY`. Если жестко задавать эту информацию в `manifest.yml` нежелательно (мы не станем вас за это винить!), то среде Cloud Foundry следует сообщить, что не нужно запускать сервис-брокер при выполнении команды `cf push`; проведите ручную привязку переменных, а затем запустите приложение (пример 14.19).

Пример 14.19. Команды CLI для развертывания и конфигурирования сервис-брокера Amazon S3

```
cf push --no-start
cf set-env s3-service-broker AWS_ACCESS_KEY_ID _yourvalue_
cf set-env s3-service-broker AWS_SECRET_ACCESS_KEY _yourvalue_
cf start s3-service-broker
```

Вам необходимо извлечь регистрационные данные Amazon Web Services из Amazon Web Services Console. Самую свежую информацию по этому процессу можно найти в документации по Amazon Web Services (https://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/credentials.html).

Теперь наш сервис-брокер Amazon S3 *полностью работоспособен* на платформе. Мы готовы известить о его наличии другие приложения, зарегистрировав его и включив в работу.

Регистрация сервис-брокера Amazon S3

Сервис-брокер нужно зарегистрировать. Для него требуется URL. Это делается вызовом команды `cf apps` (пример 14.20).

Пример 14.20. Возвращение состояния приложений Cloud Foundry в текущем пространстве

```
→ cf apps
Getting apps in org cloud-native-java / space cnj as cnj@...
OK

name              requested state   instances memory  disk  urls
s3-service-broker started           1/1       512M    1G    s3-serv..cfapps.io
```

URL для приложения в нашем примере является http://s3-service-broker-speckled-swallow.cfapps.io. Вспомним, что сервис-брокер конфигурировался с исходным именем пользователя и исходным паролем, оба имеют одинаковое имя — `admin`. Осталось только разобраться с видимостью сервиса. При развертывании приложения на платформе Pivotal Web Services, то есть в многопользовательской среде, в которой у вас, скорее всего, нет прав администратора, ваш сервис-брокер будет виден только другим приложениям в том же самом пространстве. Именно на это было указано с помощью прикрепления ключа `--space-scoped` к вызову команды `cf createservice-broker`. Если же используется PCFDev, то прикреплять эту информацию не нужно. При наличии этих сведений создание сервис-брокера упрощается (пример 14.21).

Пример 14.21. Регистрация сервис-брокера с помощью Cloud Foundry

```
cf create-service-broker s3-broker admin admin \
  http://s3-service-broker-speckled-swallow.cfapps.io --space-scoped
```

Теперь сервис-брокер должен просматриваться на торговой площадке Cloud Foundry (пример 14.22).

Пример 14.22. Инспектирование торговой площадки Cloud Foundry

```
→ cf marketplace
Getting services from marketplace in org cloud-native-java / space cnj as
cnj@...
OK

service           plans   description
...
s3-service-broker basic   Provides AWS S3 access
...
```

Располагая сервис-брокером, вы получаете возможность предоставлять экземпляры сервиса с последующей их привязкой к приложению. При желании со временем удалить сервис-брокер данный узелок вам придется распутывать! Для начала нужно вызвать команду cf delete-service-broker, но в случае ее сбоя существуют две весьма полезные команды, о которых не следует забывать. Команда cf purge-service-offering рекурсивно удалит все экземпляры заданного сервиса (или конкретный план заданного сервиса), если вы не станете больше заставлять сервис-брокер заниматься этим делом. Команда cf purge-service-instance принудительно удалит экземпляр заданного сервиса.

Создание экземпляров сервиса Amazon S3

После запуска нашего брокера S3 и его регистрации в качестве сервиса на торговой площадке PCFDev можно приступать к созданию экземпляров сервиса, чтобы получить новые бакеты хранилища S3 (пример 14.23).

Пример 14.23. Создание первого экземпляра сервиса с помощью нового брокера S3

```
$ cf create-service amazon-s3 s3-basic s3-service

Creating service instance s3-service in org pcfdev-org / space pcfdev-space as
<?pdf-cr?>admin
OK
```

Если регистрационные данные Amazon Web Services, установленные нами в виде переменных среды окружения на приложении `s3-broker`, верны, то команда, показанная в примере 14.24, будет выполнена удачно. Успешное безошибочное ее завершение будет означать, что под регистрационными данными Amazon Web Services были созданы новый бакет Amazon S3 и соответствующий IAM-пользователь. Прежде чем получить возможность увидеть, что это за ресурсы, нужно создать привязку сервиса для нового развертывания приложения, который будет применять новый экземпляр сервиса Amazon S3.

Если после попытки создания нового экземпляра сервиса Amazon S3 вы столкнетесь с отказом, то причиной будут неверные регистрационные данные, предоставленные Amazon Web Services. Чтобы справиться с проблемой, нужно убедиться, что пользователь с регистрационными данными обладает корректными политиками ресурсов, прикрепленными для администрирования IAM-пользователей и управления бакетами Amazon S3. Сконфигурировать политики можно в консоли Amazon Web Services с помощью инструмента управления идентификацией и доступом (Identity and Access Management).

Использование экземпляров сервисов. Занимаясь расширением платформы новыми сервисами, нужно также не упустить из виду предоставление разработчикам простого способа использовать эти сервисы в приложении, созданном для выполнения

в облачной среде. Хотя главный упор в проектировании платформы делается на создание сервисов, не менее важно придать этим сервисам максимальную простоту применения. Чтобы добиться эффективности в решении данного вопроса, следует принять во внимание различные рабочие нагрузки во время выполнения того приложения, которое будет задействовать новый сервис.

В этом месте пролегает разделительная черта между реализацией функционала в приложении и предложением его в качестве сервиса. Здесь действует устоявшееся правило: если нужно реализовать одни и те же функциональные возможности в каждом приложении, то их должен предоставлять сервис. Ту же точку зрения следует применять и при создании клиентских библиотек для применения новых сервисов. Для основной массы сценариев интегрирования с экземплярами сервисов должна быть уже готовая клиентская библиотека или SDK-комплект, доступный для использования в приложении. То же самое касается и наших экземпляров сервиса Amazon S3, где для взаимодействия с API хранилища Amazon S3 мы будем применять Amazon Web Services Java SDK.

При разработке JVM-приложений, применяющих пользовательские сервисы на Cloud Foundry, создатели платформы должны иметь вполне определенный взгляд на то, как проектировщики приложений задействуют эти сервисы. Из четко выраженной позиции разработчика платформы извлекается дополнительная выгода: комиссии по контролю за архитектурой не приходится проверять исходный код приложения на предмет определения уровня соответствия установленным стандартам. Будучи уверенными в том, что инфраструктура приложения интегрируется с платформой, мы получаем возможность автоматизировать выполнение в нем основного объема требований по соответствию. С помощью среды Spring Boot мы можем придерживаться подхода, актуального и для разработчика среды приложения, за счет предоставления стартовых проектов Spring Boot, которые автоматически используют экземпляры сервисов, выданные путем расширения Cloud Foundry. Сюда идеально вписывается автоматическое конфигурирование, имеющееся в Spring Boot: мы знаем, что всем приложениям Spring Boot понадобится общение с нашим бакетом S3, следовательно, для этой интеграции можем описать все подвижные части однократно и в централизованном порядке в автоконфигурации и повсеместно пользоваться этим снова и снова.

Чтобы облегчить общение разработчиков приложений с нашим брокером Amazon S3, мы создали автоконфигурацию для имеющегося в среде Amazon Web Services SDK клиента `AmazonS3`.

Этот клиент зависит от типа `BasicSessionCredentials`, который, в свою очередь, требует значения от объекта `Credentials`. Если указать правильные свойства (`amazon.aws.access-key-id` и `amazon.aws.access-key-secret`), то все эти значения настраиваются за вас (с помощью сервиса безопасных токенов в AWS в целях сокращения области видимости заданного токена) (пример 14.24).

Пример 14.24. Создание первого экземпляра сервиса с помощью нового сервис-брокера S3

```
package amazon.s3;

import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.AWSCredentialsProvider;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.auth.STSSessionCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import com.amazonaws.services.securitytoken.AWSSecurityTokenService;
import com.amazonaws.services.securitytoken.AWSSecurityTokenServiceClient;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.boot.context.properties.
EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableConfigurationProperties(AmazonProperties.class)
public class S3AutoConfiguration {

 ❶
 @Bean
 @ConditionalOnMissingBean(AmazonS3.class)
 @ConditionalOnClass(AmazonS3.class)
 public AmazonS3 amazonS3(AmazonProperties awsProps) {  ❷

  String rootAwsAccessKeyId = awsProps.getAws().getAccessKeyId();
  String rootAwsAccessKeySecret = awsProps.getAws().getAccessKeySecret();

  AWSCredentials credentials = new BasicAWSCredentials(rootAwsAccessKeyId,
   rootAwsAccessKeySecret);
  AWSSecurityTokenService stsClient = new AWSSecurityTokenServiceClient(
   credentials);
  AWSCredentialsProvider credentialsProvider =
   new STSSessionCredentialsProvider(stsClient);
  return AmazonS3ClientBuilder.standard().withRegion(Regions.US_EAST_1)
   .withCredentials(credentialsProvider).build();
 }
}
```

❶ Автоматическое конфигурирование предоставляет bean-компонент исключительно при отсутствии в контексте приложения bean-компонента такого же типа.

❷ При решении вопроса с токеном доступа Amazon Web Services, секретным ключом и сроком действия токена конфигурация зависит от компонента пользовательских параметров конфигурации `AmazonProperties`.

Приложению, которое задействует сервис, нужно только добавить эту автоконфигурацию к путям к классам и предоставить значения для двух свойств: `amazon.aws.access-key-id` и `amazon.aws.access-key-secret`.

Клиентское приложение S3

Рассмотрим обычный REST API клиента S3. Это приложение Spring Boot, которое зависит только от нашей автоконфигурации и `spring-boot-starter-web`. Мы создадим экземпляр сервиса S3 (присвоив ему логическое имя `s3-service`) и привяжем к нашему клиенту S3 при его развертывании на платформе Cloud Foundry (пример 14.25).

Пример 14.25. Развертывание клиента S3

```
cf create-service s3-service-broker basic s3-service
cf push
```

Приложение является REST-сервисом с двумя конечными точками: `/s3/resources`, поддерживающей HTTP-запросы `GET`, и `/s3/resources/{name}`, поддерживающей HTTP-запросы `POST` с составными загрузками файлов. Загрузите файл, а затем проверьте его наличие. Код показан в примере 14.26.

Пример 14.26. S3RestController работает с предоставленным экземпляром сервиса S3

```
package com.example;

import amazon.s3.AmazonProperties;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.model.*;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.hateoas.Link;
import org.springframework.hateoas.Resource;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import java.net.URI;
import java.time.Instant;
import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping("/s3")
class S3RestController {

 private Log log = LogFactory.getLog(getClass());
```

```
private final AmazonS3 amazonS3Client;

private final String defaultBucket;

@Autowired
public S3RestController(AmazonProperties amazonProperties,
 AmazonS3 amazonS3Client) {
 this.amazonS3Client = amazonS3Client;
 this.defaultBucket = amazonProperties.getS3().getDefaultBucket();
 log.debug("defaultBucket = " + this.defaultBucket);
}

❶
@PutMapping("/resources/{name}")
ResponseEntity<?> upload(@PathVariable String name,
 @RequestParam MultipartFile file) throws Throwable {

 if (!file.isEmpty()) {
  ObjectMetadata objectMetadata = new ObjectMetadata();
  objectMetadata.setContentType(file.getContentType());
  PutObjectRequest request = new PutObjectRequest(this.defaultBucket, name,
   file.getInputStream(), objectMetadata)
   .withCannedAcl(CannedAccessControlList.PublicRead);
  PutObjectResult objectResult = this.amazonS3Client.putObject(request);
  URI location = URI.create(urlFor(this.defaultBucket, name));
  String str = String
   .format("uploaded %s at %s to %s", objectResult.getContentMd5(), Instant
    .now().toString(), location.toString());
  log.debug(str);
  return ResponseEntity.created(location).build();
 }
 return ResponseEntity.badRequest().build();
}

❷
@GetMapping("/resources")
List<Resource<S3ObjectSummary>> list() {

 ListObjectsRequest request = new ListObjectsRequest()
  .withBucketName(this.defaultBucket);

 ObjectListing listing = this.amazonS3Client.listObjects(request);

 return listing.getObjectSummaries().stream().map(this::from)
  .collect(Collectors.toList());
}

private String urlFor(String bucket, String file) {
 return String.format("https://s3.amazonaws.com/%s/%s", bucket, file);
}
```

```
 private Resource<S3ObjectSummary> from(S3ObjectSummary a) {
  Link link = new Link(this.urlFor(a.getBucketName(), a.getKey()))
   .withRel("location");

  return new Resource<>(a, link);
 }
}
```

❶ В этой конечной точке поддерживается загрузка файлов. Можно попробовать выполнить данное действие, воспользовавшись, к примеру, командой `curl` или дополнительным модулем браузера под названием PostIt.

❷ Эта конечная точка перечисляет все загруженные в экземпляр Amazon S3 ресурсы.

В `S3RestController` для общения с S3 используется сконфигурированный клиент `AmazonS3`. Автоматическое конфигурирование требует правильного секретного ключа и ключа доступа. Мы можем в свойствах конфигурации отобразить их на привязанный к среде опорный сервис S3.

Пример 14.27. Конфигурация для простого приложения с двумя профилями, один из которых предназначен для локальной разработки, а другой — для развертывания в Cloud Foundry с активным облачным профилем

```
spring.profiles.active: development
spring:
  http:
    multipart:
      file-size-threshold: 10MB
      max-file-size: 10MB
      max-request-size: 10MB
server:
  port: 8081
---
spring:
  profiles: cloud
  application:
    name: ${vcap.application.name:s3-sample}
amazon:
  aws:
   ❶
   access-key-id: ${vcap.services.s3-service.credentials.accessKeyId}
   access-key-secret: ${vcap.services.s3-service.credentials.accessKeySecret}
  s3:
   default-bucket: ${vcap.services.s3-service.credentials.bucket:bucket}
---
spring:
  profiles: development
  application:
    name: s3-sample-app
amazon:
```

```
aws:
  access-key-id: replace
  access-key-secret: replace
```

❶ Регистрационные данные опорного сервиса сохранены в JSON-структуре, находящейся в переменной среды окружения по имени `VCAP_SERVICES`. Spring Boot демонтирует JSON-структуру в ключи в Spring Environment, начинающиеся с `vcap.services.service-id`, где `service-id` — логическое имя сервиса, привязанного к приложению. В этом случае мы создали экземпляр сервиса S3 под названием `s3-service`. Данная конфигурация разыменовывает предоставленный секретный ключ и ключ доступа, а также выданную информацию о бакете с целью настройки клиента AmazonS3.

Посмотрим, что получилось

Имея в своем распоряжении сервис-брокер и развернутое клиентское приложение, выполним прогон этого решения. Сначала запишем в S3 файл, выдав `PUT`-запрос в отношении `/s3/resources/{file-name}`, где `file-name` — имя файла, к которому нужно будет обратиться впоследствии (пример 14.28).

Пример 14.28. Запись в Amazon S3 с именем test (мы сократили строку этого листинга, воспользовавшись многоточием, вместо которого нужно подставить HTTP URI вашего развернутого клиента Amazon S3)

```
curl -v -F file=@/path/to/an/img.png
  http://s3-sa...cfapps.io/s3/resources/test
```

Вы должны иметь возможность подтверждать результаты путем выдачи `GET`-запроса к конечной точке `/s3/resources` (пример 14.29).

Пример 14.29. Чтение из Amazon S3

```
curl -v http://s3-sample-app-...cfapps.io/s3/resources
```

Резюме

В данной главе были рассмотрены сервис-брокеры — основополагающая часть платформы Cloud Foundry. Они предлагают единый интерфейс предоставления и использования сервисов и упрощают работу операторов. Имеющееся в среде Spring Boot автоматическое конфигурирование является в данном случае ценным дополнением: оно обеспечивает подход к подключению предусмотренных сервисов, не вызывающий трений среди создателей приложений. Наиболее важный аспект как сервис-брокеров, так и автоматической конфигурации — они оптимизированы для согласованности, что, в свою очередь, ускоряет работу.

Сервис-брокеры — одна из наиболее эффективных парадигм, появившихся впервые в среде Cloud Foundry, но нам приходилось видеть, как она принималась и на других платформах. Инициатива Open Service Broker API (https://www.cloudfoundry.org/blog/open-service-broker-api-launches-as-industry-standard/), предназначенная для зарождения стандартизации доставки сервисов в рамках облачных предложений, внесла сервис-брокеры в еще одно предложение облачной среды. Эта инициатива, кроме всех прочих, включает участие таких поставщиков, как Google, Red Hat и IBM.

15 Непрерывная поставка

В данной книге основное внимание уделялось созданию ПО, пригодного для ввода в эксплуатацию. Особенно подробно мы говорили о разработке программ, готовых к поставке конечному потребителю. Кроме того, уделили пристальное внимание применению платформы Cloud Foundry, которая, если процитировать Эндрю Клей Шафера (Andrew Clay Shafer), является «вооруженной инфраструктурой», или инфраструктурой, открытой для разработчика, оптимизированной под развертывание и запуск приложений. Мы изучили вопросы получения в ней программ, создания приложений, получающих после развертывания все преимущества этой платформы.

В текущей главе мы собираемся рассмотреть организационные причины, побуждающие к использованию таких средств, как Cloud Foundry и архитектура микросервисов. Мы обратимся к практике *непрерывной поставки* (continuous delivery) и применению Concourse, технологии, поддерживающей конвейеры такой поставки.

Не только непрерывная интеграция

Вернемся на шаг назад и взглянем на то, что находится за пределами разработки программы, на ее развертывание и выпуск. Среда Spring Boot упрощает установку сервисов, пригодных для ввода в эксплуатацию. Она позволяет сократить интеллектуальные (и фактические) вложения в получение работоспособного ПО, дает организациям возможность быстрее добиться нужных результатов. Многие организации вполне благосклонно относятся также к применению непрерывной интеграции (continuous integration, CI), позволяющей разработчикам как можно раньше получить отзывы о своем коде и тестах. А есть ли польза от CI, Spring Boot, разработки на основе постоянного тестирования и применения методов гибкого проектирования ПО, если на получение предусмотренного оборудования и развернутой инфраструктуры уходят дни, недели или делаются бесчисленные заявки

в ITIL? Более того, что хорошего в оптимизации процесса создания кода, если работа, начиная от выработки концепции и заканчивая вводом в эксплуатацию, тормозится на любой промежуточной стадии: тестировании, проведении технологических операций и т. д.?

Никто не создает микросервисы, чтобы затем вручную заниматься их развертыванием на совмещенных экземплярах WebSphere, которые запущены на оборудовании, предоставленном по ITIL-запросу на использование аппаратных средств и инфраструктуры. Это в лучшем случае сервис-ориентированная архитектура (service-oriented architecture, SOA), а не микросервисы. В последних поддерживается гибкость. Они — часть потока создания ценностей, каждый шаг в котором способствует поставке ПО. Нельзя применять микросервисы, игнорируя этот более широкий контекст.

> Благодаря информационным технологиям программа поставляется быстрее, чем мы можем придумать в ней какие-либо изменения!
>
> *Это сказал точно не бизнесмен*

Поток создания ценностей для большинства организаций представляет собой примерно следующую картину: появляется бизнес-идея, проработку ее поручают отделу сбыта, отделу изучения спроса потребителей и отделу информационных технологий. Когда-нибудь, надеемся, не в самом отдаленном будущем, порожденный всем этим продукт оказывается в руках *клиента*, под которым подразумевается тот, кто спонсирует все эти действия и может относиться как к внутренним, так и к внешним пользователям, получающим продукт за плату. ПО в руках клиента должно представлять *ценность* для бизнеса или организации. Бизнесу, конечно же, понравилось бы, имей мы возможность упорядочить и ускорить весь поток создания ценностей, но специалисты по информационным технологиям традиционно придерживаются следующего мнения: при слишком быстром продвижении будут допущены ошибки. Речь здесь идет о том, что основная часть работы конвейера поставки выполняется вручную, и в этом заключается суть проблемы.

А у вас в конвейере поставки также применяется ручной труд? И десятки специалистов скрупулезно изучают код, тщательно выискивая ошибки? Требует ли ручное развертывание внесения изменений в производственную среду согласно выработанной документации? Насколько редко случаются такие развертывания: раз в месяц, или каждые три месяца, или дважды в год? Вызывает ли перспектива *дня развертывания* ужас у специалистов? Требует ли развертывание от специалистов особых усилий? Создаются ли оперативные центры, куда может обратиться любой специалист команды или другой привлеченный сотрудник? Если на какой-либо из перечисленных вопросов может быть дан положительный

ответ, то налицо явные симптомы неэффективной организации и раздробленного конвейера поставки.

Непрерывная поставка предлагает лучший путь развития. Она как погоня за совершенством, а не как предоставление абсолютно новой вещи. Ее идея проста: максимально автоматизировать поток создания ценностей. В частности, автоматизировать все после передачи кода, чтобы путь от передачи к производству всякий раз повторялся без вмешательства человека. Некоторые организации уже проделывают это или что-то близкое к нему на протяжении десятилетий.

Непрерывная поставка согласно сложившейся на сегодня практике многое заимствует у идей экономичного производства: она обеспечивает быструю поставку высококачественной продукции, уделяя особое внимание устранению убытков и снижению стоимости. В результате достигаются экономия средств и ресурсов, производство высококачественной продукции и более высокие темпы ее вывода на рынок.

Как бы парадоксально это ни выглядело, но устранение трудоемких ручных операций из процесса поставки *сократило* количество дефектов и *повысило* качество продукции. Такие операции, как развертывание ПО, носят высокотехнологичный характер, но все же трудоемки и весьма утомительны (скучны!). И это одно из самых коварных сочетаний, к которому люди просто не приспособлены. Без ошибок и неувязок здесь не обойтись. Устранить ошибки в процессе может только *полная автоматизация*. Она приводит к появлению процесса поставки, работающего одинаково как при обкатке, так и в условиях эксплуатации. Все ошибки в процессе можно устранить при обкатке, избавляя организацию от попыток выполнить отладку с целью устранить ошибки на этапе промышленного применения.

Автоматизация, облеченная в форму ПО, — контролируемая версия компонента, который способен оказать помощь в централизации и обобщении работы по развертыванию. Больше не нужно быть великим специалистом по развертыванию, любому под силу взять то, что находится в системе управления версиями, и запустить программу для получения надежной сборки версии приложения. По мере ускорения развертывания снижаются издержки на устранение дефектов, поскольку высока вероятность, что новый выпуск выйдет в ближайшее время. Скорость снижает риски, связанные со стоимостью изменений, что упрощает попытки их внесения. В изменениях кроется свойство выживаемости бизнеса.

При правильной организации поставка ПО должна сводиться к простому выбору версии программы из системы управления версиями и нажатию эквивалента большой зеленой кнопки с надписью «Развернуть!». В некоторых организациях пошли еще дальше: исключили из обихода такую кнопку и внедрили *непрерывную поставку*, где путь от успешного помещения кода в его систему хранения и контроля версий (`git push`) прокладывается прямо к стадии эксплуатации при условии, что все этапы проверки качества уже пройдены.

Работа Джона Оллспоу в Flickr, а затем в Etsy

Джон Оллспоу (John Allspaw) помог компании Flickr (и чуть позже компании Etsy) освоить непрерывную поставку и отправить объемное, монолитное приложение. История с компанией Etsy, подробно изложенная в статье в Network World (https://www.networkworld.com/article/2886672/software/how-etsy-makes-devops-work.html), за последнее десятилетие вдохновила многие компании. Оллспоу давно отстаивал непрерывную поставку и рассказывал о том, что в его организации сделано в области совершенствования культуры производства и технических приемов, чтобы заставить непрерывную поставку работать на благо компании. Обе компании — и Flickr, и Etsy — занимаются ПО в качестве сервисов (Software as a Service), поставляемых в режиме онлайн. Оллспоу много говорил (например, на конференции 2009 Velocity (https://www.youtube.com/watch?v=LdOe18KhtT4)) о том, как поставки веб-приложений (с которыми, как мы подозреваем, сталкивается большинство читателей данной книги) изменяют подходы к поставке в целом.

В SaaS-модели не существует прежней версии ПО. Обновите окно браузера — и получите самую последнюю установку кода. Вошедший в практику большинства организаций способ использования ветвлений (работы над возвратом к прежним версиям или над дефектами в упакованном и отправленном ПО) больше не применяется. В Etsy все изменения были отправлены в основную версию проекта. Имеющиеся в них функциональные свойства, которые необходимо было ввести в кодовую базу, находятся под флагами новизны (feature flags). Последние позволяют Etsy развертывать функциональные свойства для некой процентной доли получателей продукта. Эти свойства могут присутствовать в кодовой базе, собирая информацию и существуя в оперативном пути реальных запросов к сервису, где производство, кроме всего прочего, может контролировать загруженность системы. Данная информация используется при планировании возможностей и обсуждении вопросов, связанных с архитектурой. К моменту, когда бизнес объявит о существовании функциональной возможности и ее «выпуске», она становится вполне обыденным явлением; сервис уже работает и успешно справляется со своими задачами на протяжении недель или месяцев. При возникновении *неполадок* нужно просто отключить функциональное свойство, сняв флаг новизны и перенеся его использование на будущее.

В Etsy быстро научились автоматизировать перемещение кода на этап ввода в эксплуатацию, создавая механизмы защиты как в ПО, так и в самой организации для поддержки непрерывной поставки.

Они не задавались целью проводить несколько развертываний в день; частота данного этапа должна соответствовать задачам и безопасности производства.

> Если у вас по десять зависаний на день, проводить по десять развертываний в день просто невозможно.
>
> *Джон Оллспоу*

Работа Адриана Кокрофта в Netflix

Адриан Кокрофт (Adrian Cockroft) помог начавшемуся в 2009 году переходу компании Netflix от применения монолитной базы данных и архитектуры приложения к архитектуре, оптимизированной для выполнения в облаке. Среди многих идей (и всех потрясающих технологий, основанных на использовании открытого кода, которые вы, несомненно, применяли в процессе чтения этой книги!), возникших в результате вторжения компании Netflix в мир микросервисов, было мнение, что скорость итерации намного ценнее эффективности выполнения кода. Чем быстрее может прогрессировать организация, тем меньшее значение имеет то, что конкретный сервис работает медленнее, чем мог бы, поскольку это помогает сохранить конкурентное лидерство. В таком случае организация контролирует сферу деятельности. Адриан говорил об этом в многочисленных интервью, в том числе для Scale (https://medium.com/s-c-a-l-e/talking-microservices-with-the-man-who-made-netflix-s-cloud-famous-1032689afed3):

> Существует лимит по количественному составу команды, позволяющий проявлять гибкость. Данный лимит, вероятно, определяется текущей работой Etsy, поскольку их специалисты лучшие в запуске очень гибких монолитных приложений. Это великое достижение; однако многие говорят следующее: это потрясающе, но куда проще делать что-то более мелкими порциями, пусть менее эффективно, но мне важнее скорость, а не эффективность.
>
> *Адриан Кокрофт*

Ключевой фактор — скорость. Насколько быстро вы можете выполнить итерацию?

Непрерывная поставка в Amazon

В своей презентации DevOps at Amazon: a Look at Our Tools and Processes[1] на конференции 2015 AWS re:Invent Роб Бригам (Rob Brigham) рассказал захватывающую историю о переходе компании Amazon от монолитных приложений к микросервисам.

Бригам пояснил, что для Amazon идея DevOps стала дополнительным фактором повышения эффективности, сократившим простои в жизненном цикле поставки ПО. На первый взгляд, это может быть совсем не тем прозрением, связанным с DevOps, которое вы пытались отыскать. Но для Бригама только это одно утверждение изменило весь известный нам мир вычислений.

---

[1] DevOps at Amazon: A Look at Our Tools and Processes. 2015. https://www.youtube.com/watch?v=esEFaY0FDKc

Он сказал, что жизненный цикл поставки в компании Amazon имеет две стороны: разработчиков и клиентов. Чтобы функциональное свойство было получено клиентом от разработчика, каждое изменение должно пройти через фазы сборки, тестирования и выпуска. Как только сборка будет выпущена, разработчики могут отслеживать порядок взаимодействия клиентов с новыми функциональными свойствами. Этот цикл обратной связи играет весьма важную роль, поскольку дает проектировщикам понимание, столь необходимое им для определения того, над чем им нужно работать в дальнейшем.

Бригам сказал, что этот цикл обратной связи позволяет разработчикам компании Amazon реже строить догадки и поставлять более ценные функциональные свойства, воспринимаемые клиентами с большим удовольствием. Он также сказал, что в Amazon практикуется как можно более быстрое завершение отдельно взятого цикла обратной связи. Непрерывная поставка для данной компании зарождается из снижения затрат времени разработчиков либо на сборку и тестирование, либо на выпуск кода, поскольку клиенты не видят эту работу. При условии сведения к минимуму времени, затрачиваемого на фазу поставки, непрерывная поставка приводит к сокращению времени, необходимого для того, чтобы изменения стали видимы клиентам.

Конвейер

Когда речь заходит о непрерывной поставке, имеется в виду автоматизированный процесс, то есть *конвейер*, поставляющий ПО в производственную среду. Конвейер главным образом относится к тому путешествию, которое должно быть проделано программой от непрерывной интеграции до ввода в эксплуатацию. Мы слышали, что это еще называют обналичиванием кода.

Этот автоматизированный процесс называется конвейером поставки. Конвейер высвечивает каждый аспект поставки ценности клиенту; специалисты по бизнесу и информационным технологиям могут последовательно дискутировать о том, насколько далеко продвинулась работа в потоке создания ценностей. Конвейер выступает в роли единого канала, по которому должны пройти все изменения: оперативные исправления, новые функциональные свойства и регрессии. Подпитывать канал должны как разработчики, так и операторы. Крайне важно, чтобы никто не нарушал работу конвейера! Если он не отвечает требованиям всех вовлеченных сторон, то не может вызывать доверия и поэтому должен совершенствоваться. Конвейер — единственная визуализация, где все стороны прослеживают обратную связь, касающуюся выполняемой работы, что позволяет устранять любые ошибки на самой ранней стадии.

Каждый конвейер перед выполнением развертывания в производство будет иметь набор основных этапов, на которых берется изменение в исходном коде и в безопас-

ном режиме проводится его сборка и тестирование (подобно тому как это могло бы быть в традиционной непрерывной интеграции). Этапы в конвейере должны иметь описание в компоненте, доступном для системы управления версиями. Данный компонент может иметь вид дополнительного ПО, сценариев или конфигурации.

Ценность непрерывной интеграции заключается в том, что она вынуждает команды разрабатывать программы, работающие где-либо, кроме их машин. Если в запуске кода возникают какие-либо затруднения, то все несоответствия должны быть зафиксированы и добавлены к системе управления версиями и автоматизированного развертывания. Первым заданием новой команды должна стать установка конвейера непрерывной поставки на нулевую итерацию.

В непрерывной поставке изменения в ПО (обычно в форме отправки кода в основную ветвь разработки в системе управления версиями наподобие Git) прогоняются по конвейеру. Система управления должна иметь все необходимое для воспроизведения приложения и его среды. Все, за исключением производственных данных, должно присутствовать в управлении версиями, даже сами определения конвейера. Для современных Spring-приложений, запускаемых на платформе Cloud Foundry, это может касаться таких вещей, как сборка Maven или Gradle, файл `manifest.yml` платформы Cloud Foundry, и пересмотра любой конфигурации в Spring Cloud Config Server.

Большинство конвейеров поставки будут выполнять следующие и, возможно, еще и дополнительные этапы: сборка и блочное тестирование, комплексное тестирование, выпуск и развертывание.

- *Сборка и блочное тестирование*. Программный продукт собирается в пакет или компилируется в тот вид, которым могут воспользоваться последующие этапы. Важно отметить, что после этого потребность в исходном коде может быть весьма низкой (за исключением, вероятно, анализа качества кода). При перемещении приложения из одной среды в другую должен использоваться один и тот же двоичный код или пакет. Здесь для хранения успешно созданного продукта может служить хранилище компонентов, подобное Artifactory компании JFrog.
- *Интеграция*. Компонент из предыдущего этапа развертывается в среде эксплуатации, после чего выполняется. Cloud Foundry упрощает описание и развертывание идентичных и воспроизводимых сред, извлекаемых из краткого манифеста с расширением `.yml`. Здесь тесты могут носить более масштабный, сквозной характер. Кроме того, они способны проверять состояние зависимостей сервиса, запустив его в целях проверки отсутствия критических сбоев. Такое тестирование программы, называемое *дымовым*, направлено на проверку работоспособности наиболее важных функций приложения. Вполне возможно, что специалистам захочется провести и исследовательское тестирование. После прохождения данного этапа ПО передается на этап выпуска.

- *Выпуск*. Компонент получает статус выпускаемого. Но настоятельной необходимости в его выпуске нет. Он может просто пылиться в хранилище компонентов. Важно то, что он может быть без сомнений развернут в среде эксплуатации. Развертывание, происходящее автоматически, является непрерывным. Если автоматическое развертывание не предусматривается, то, значит, это непрерывная поставка.
- *Развертывание*. Выпускаемое ПО внедряется в производственную среду. Для онлайн-приложений SaaS часто используются шаблоны наподобие синезеленых развертываний (blue-green deploys), полностью исключающие какие-либо простои.

Тестирование

> Прекратите добиваться качества за счет контроля. Исключите необходимость в тотальном контроле, изначально встроив качество в продукт.
>
> *У. Эдвардс Деминг (W. Edwards Deming),*
> *14 Points for Total Quality management*
> *(«14 принципов всестороннего управления качеством»)*
> *(http://asq.org/learn-about-quality/total-quality-management/overview/deming-points.html)*

Быстрое продвижение имеет смысл, только если оно безопасно. Никто не поверит в план, в котором говорится: «Мы намерены ускориться, отказавшись от любого контроля качества! Только скорость! И никакого качества! И будь что будет!»

По мере прохождения по конвейеру изменение должно преодолеть все более всеохватывающие (и замедляющие продвижение) рубежи контроля качества. Если в какой-то момент изменение останавливает конвейер, срывая сборку, то первоочередной задачей вовлеченных в этот процесс команд становится восстановление сборки. И до оздоровления обстановки в конвейер не следует вносить никаких новых изменений. В компании Toyota настаивали на том, что любые производственные дефекты должны останавливать производственный конвейер до устранения дефекта. Нам тоже нужно остановить конвейер.

Непрерывная поставка основана на применении тестирования рубежей контроля качества. Применительно к приложениям среды Spring более обстоятельный разговор о тестировании велся в главе 4. В более широком понимании вопроса существует четыре достойные рассмотрения разновидности тестирования, и если вы переходите к непрерывной поставке, то нужно постараться воспользоваться их максимально возможным количеством. Сфера тестирования показана на рис. 15.1 *квадрантами тестирования* Брайана Марика (Brian Marick).

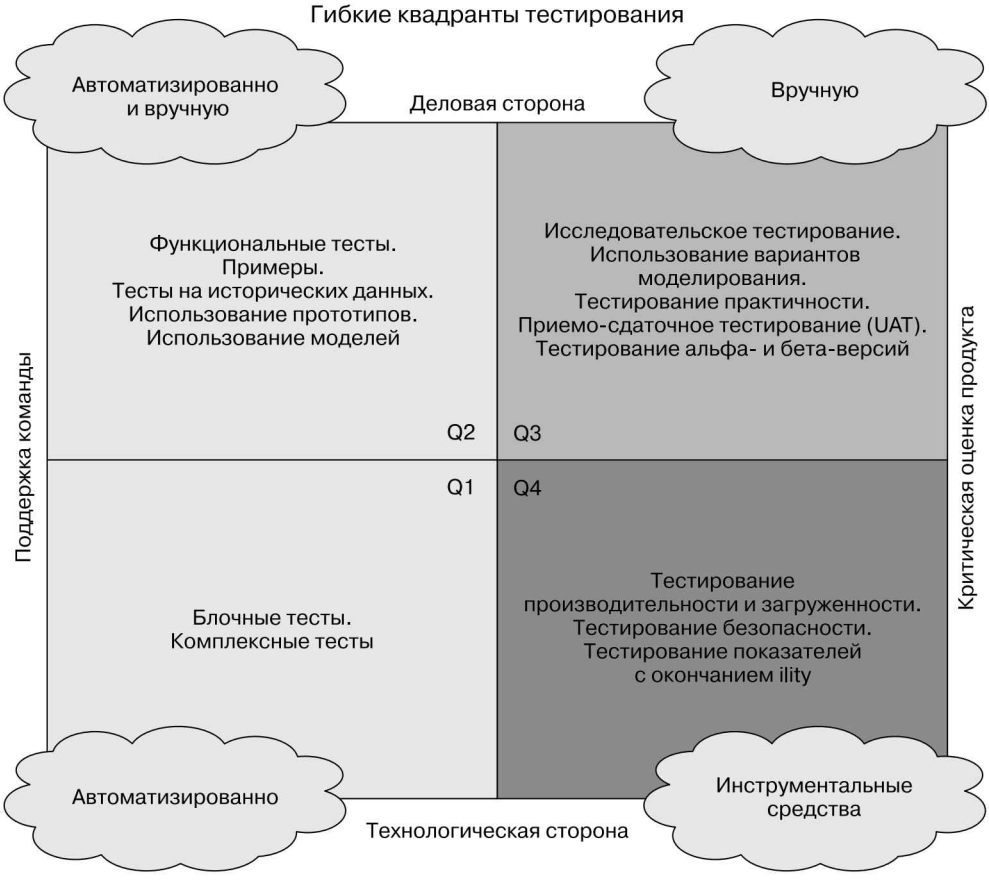


Рис. 15.1. Квадранты тестирования Брайана Марика

Непрерывная поставка для микросервисов

Одна из наиболее важных тем в разработке ПО — модульность. Если задумываться о ней в механическом смысле, то компоненты системы создаются в виде модулей, которые при чисто механическом сбое могут быть изменены. Возьмем, к примеру, автомобильный двигатель: его не нужно заменять целиком при поломке какой-нибудь одной его части. Однако в программе после одного незначительного изменения зачастую заменяется все приложение. Поэтому нам, считайте, повезло, что мы создаем программы, а не машины. Некоторые из сегодняшних наиболее влиятельных компаний были созданы с использованием битов и байтов, а не пластика и металла. Но, несмотря на эти преимущества, по-прежнему имеется ряд производящих автомобили компаний, способных выпускать машины значительно быстрее, чем собственное ПО.

Что легче сделать: заменить спустившее колесо на автомобиле или внести изменения в модуль программного продукта?

Модульность также способна дать разработчикам схему *обоснования* функциональных средств приложения. Возможность визуализации и переноса на схему сложных процессов, управляемых исходным кодом приложения, позволяет разработчикам как кода, так и архитектуры с хирургической точностью визуально определить место внесения изменения.

Исходный код приложения представляет собой постоянно развивающуюся систему связанных между собой битов и байтов — изменения следуют друг за другом. Но поскольку исходный код приобретает или теряет свой объем, то небольшие изменения требуют проведения сборки и развертывания целиком всех приложений. Чтобы внести одно небольшое изменение в производственную среду, приходится развертывать и все остальное, не подвергавшееся переменам.

Перенос изменений в производство на совместно используемом конвейере развертывания похож на покупку билета в один конец на автобус, отправляющийся в город под названием Производство. Посадка в автобус в большом городе — дешевая альтернатива управлению собственной машиной. Проблема в том, что вы не можете точно выбрать время, когда вас подберут на маршруте. То же самое имеет место при наличии одного конвейера развертывания.

А что, если вместо принудительного ожидания следующего автобуса можно было бы вызвать автобус ко времени вашей готовности к поездке? Эта идея не нова! Фактически она меняет способ использования технологии на путешествие в машине по требованию. Тогда как вы посмотрите на применение той же самой идеи к переносу изменений в производство?

Непрерывная поставка похожа на использование сервиса проката машин, предоставляемого по требованию с целью переносить ваши изменения в производство по мере надобности.

Когда команды совместно используют для приложения конвейер развертывания, они вынуждены составлять график применения, повлиять на который они не могут или же могут, но весьма и весьма ограниченно. По этой причине нововведения сдерживаются, поскольку люди вынуждены ждать следующего «автобуса», прежде чем смогут получить какую-либо ответную реакцию на свои изменения.

Результат внедрения микросервисов — неизменно возрастающее количество путей продвижения изменений в производство.

Инструменты

Есть несколько доступных инструментов, содействующих непрерывной поставке. В развертываниях на основе облачных технологий все еще широко используются такие средства, как Jenkins. Хотя эпоха облачных вычислений открыла множество

новых участников в управлении выпусками, многие чувствуют себя комфортнее, задействуя уже знакомые им инструменты. Новые средства характеризуются нацеленностью на различные фазы жизненного цикла поставки программных продуктов. В данной главе мы собираемся исследовать присущую облачной среде непрерывную поставку, осуществляемую с помощью инструмента под названием *Concourse*.

Concourse

Concourse — инструмент, «родившийся» готовым к приходу эры естественного применения облачной среды. Весь набор инструментов Concourse был создан компанией Pivotal. Он представляет собой средство непрерывной интеграции, позволяющее использовать Docker-контейнеры для составления простых или сложных конвейеров развертывания. Concourse отличается от таких инструментов, как Jenkins или Travis, двумя особенностями: контейнерами и конвейерами.

Контейнеры. Технология Docker революционизировала способ нашего осмысления упаковки и распространения приложений в облаке. На случай, если она вам еще не знакома, технология Docker зарождалась в виде инструмента упаковки с открытым кодом, позволяющего описывать и выстраивать среду выполнения для ваших приложений в виде Linux-контейнера. Последний лучше всего поддается описанию *в виде* виртуальной машины, но фактически это всего лишь файловая система, совместно использующая ядро Linux.

В Cloud Foundry наблюдается существенное абстрагирование от контейнеров с помощью сборочных пакетов (buildpacks), которые станут рационально расставлять ваши компоненты развертывания внутри контейнера, исключая необходимость описания среды выполнения. И хотя Cloud Foundry использует собственный инструментарий управления контейнерами, имеющий открытый код, Concourse позволяет составлять конвейеры, применяя Docker-контейнеры.

Docker — идеальный вариант для запуска задач в составном конвейере развертывания. Concourse позволяет создавать кратковременную среду сборки, тестирования и развертывания в виде конвейера задач, которые будут выполняться внутри Docker-контейнера.

Непрерывно поставляемые микросервисы

Мы собираемся создать два процесса непрерывной поставки, применяющих рассмотренные ранее этапы для перемещения кода из его источника в производство. Основной объем процесса непрерывной поставки приходится на тестирование. Поэтому мы собираемся еще раз воспользоваться примерами кода из главы 4, чтобы создавать, тестировать и развертывать микросервисы Spring Boot с помощью Concourse. В главе, посвященной тестированию, было два микросервиса: *Account Microservice* и *User Microservice* (рис. 15.2).

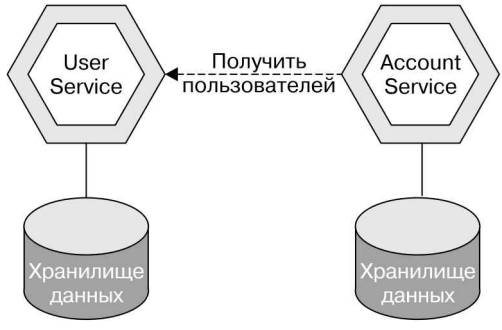


Рис. 15.2. Микросервис учетных записей зависит от микросервиса пользователей

Преимущество, получаемое от создания микросервисов, состоит в том, что команды могут независимо друг от друга развертывать функциональные средства в среде эксплуатации. Отсутствие единого конвейера развертывания — важное преимущество для такого типа архитектуры. Но за это приходится платить ответственностью. Команды должны обеспечить отсутствие возникновения аварийной ситуации у потребителя. Данный вопрос относится к непрерывной интеграции, при которой нам нужно протестировать наши микросервисы как можно раньше, перед переносом любых изменений в производство.

Нарушающее нормальную работу изменение может привести к чему угодно, от катастрофы (в случае возникновения непоследовательности данных) до небольшой неприятности, не оказывающей критического влияния на основную функциональность. Каждый потребитель микросервиса должен иметь возможность заявить о том, что он ожидает от сервиса, от которого зависит. Мы намереваемся разработать конвейер непрерывной поставки, выполняющий непрерывную интеграцию, чтобы обеспечить неизменность ожиданий потребителя от производителя, прежде чем существенное изменение сможет быть внедрено в производство.

Установка Concourse

Существует несколько способов установки Concourse, наиболее быстрый из которых предусматривает использование программы Vagrant. Самые последние инструкции по установке Concourse с помощью Vagrant можно найти в документации по Concourse.

Concourse имеет CLI-средство `fly` и интерфейс на веб-основе для визуализации состояния конвейера. При установленном и запущенном средстве Concourse и установленном средстве `fly` вы получите возможность указать то место, куда нужно установить локальный экземпляр Vagrant и начать создание конвейеров.

Чтобы приступить к использованию `fly`, нужно зарегистрироваться в вашем экземпляре Concourse. Если применяется Vagrant, то URL будет похож на следующий:

```
fly -t lite login -c http://192.168.100.4:8080
```

При успешной регистрации в вашем экземпляре Concourse с помощью `fly` появится возможность приступить к созданию конвейеров. Полезно будет узнать о двух простых командах: `set-pipeline` и `destroy-pipeline`. Команда `set-pipeline` показана в примере 15.1.

Пример 15.1. Создание нового конвейера с помощью set-pipeline

```
fly -t lite set-pipeline -p account-microservice -c pipeline.yml \
  -l .pipeline-config.yml
```

Здесь мы создаем новый конвейер для микросервиса учетных записей. Команда `set-pipeline` принимает несколько параметров, показанных в табл. 15.1.

Таблица 15.1. Параметры set-pipeline

Ключ	Параметр	Описание
-p	account-microservice	Имя создаваемого конвейера Concourse
-c	pipeline.yml	Путь к YAML-файлу, содержащему описание нового конвейера Concourse
-l	.pipeline-config.yml	Путь к YAML-файлу, включающему секретные данные, которые станут параметрами для вашего нового контейнера

В табл. 15.1 показаны параметры команды `set-pipeline` из примера 15.1. Обратите внимание: значения параметров берутся из исходного кода примера, сопровождающего эту главу. После выполнения данной команды вы получите возможность просматривать веб-интерфейс администратора Concourse и видеть готовый к запуску новый конвейер.

Еще одна полезная команда, о которой нужно знать, — `destroy-pipeline`. Конвейеры имеют неизменяемый характер, следовательно, есть настоятельная необходимость в их уничтожении и создании заново с приемлемыми качествами. Уничтожение конвейера выполняется в примере 15.2.

Пример 15.2. Уничтожение существующего конвейера с помощью destroy-pipeline

```
fly -t lite destroy-pipeline -p account-microservice
```

В примере 15.2 показана команда, которую можно использовать для уничтожения конвейера `account-microservice`, созданного в результате выполнения кода из примера 15.1.

Основная конструкция конвейера

Основная конструкция конвейера непрерывной поставки будет включать фазы сборки, тестирования и развертывания.

Определение конвейера Concourse для микросервиса учетных записей показано на рис. 15.3. Concourse состоит из трех элементов, используемых для создания конвейера, похожего на показанный выше, — ресурсов, задач и заданий (табл. 15.2).

Рис. 15.3. Конвейер для микросервиса учетных записей (Account Microservice)

Таблица 15.2. Конвейерные элементы Concourse

Тип	Функция
Ресурс (Resource)	Ресурс, являющийся вводом или выводом задачи
Задача (Task)	Задачи принадлежат заданию, создавая выходные данные из ресурсов внутри Docker-контейнера
Задание (Job)	Задания группируют набор задач и являются самым большим строительным блоком конвейера

Изучая рис. 15.3, можно заметить, что у изображенного на нем конвейера нет этапа интеграции. Пока у микросервиса учетных записей нет ни одного потребителя, но он зависит от сервиса пользователей. По этой причине мы начнем с разработки простого конвейера, имеющего этапы сборки, тестирования и выпуска.

В табл. 15.3 описано каждое задание в конвейере Concourse, принадлежащем сервису учетных записей. В Concourse поддерживается несколько типов ресурсов, определение которых будет дано в файле определений вашего конвейера. В учебном проекте, сопровождающем данную главу, имеется исходный код сервиса учетных записей, в котором есть каталог, содержащий полное определение конвейера Concourse.

Таблица 15.3. Concourse-задания для микросервиса учетных записей

Задание	Входные данные	Выходные данные	Описание
Сборка (Build)	Текущая версия, Git-источник	Выпуск компонента, увеличенный на единицу номер версии	При выполнении задания сборки будет клонирован и собран исходный код микросервиса учетных записей, и при условии прохождения блочных тестов будет подготовлен новый выпуск компонента с присвоенной ему версией
Выпуск (Release)	Увеличенный на единицу номер версии, Git-источник, выпуск компонента	Развертывание на платформе Cloud Foundry	Задание выпуска получает выпущенный компонент и развертывает новую версию на платформе Cloud Foundry

На рис. 15.4. показаны файлы, которые послужат входными данными для команды `set-pipeline`, создающей Concourse-конвейер для микросервиса учетных записей. Первым файлом, который мы исследуем, станет `pipeline.yml`, содержащий определение Concourse-конвейера.

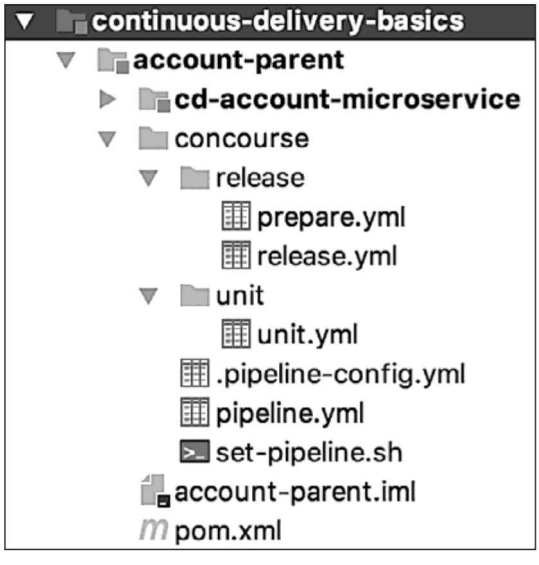


Рис. 15.4. Каталог исходного кода микросервиса учетных записей

В примере 15.3 показана первая часть определения Concourse-конвейера для микросервиса учетных записей. Это Concourse-ресурсы, которые послужат в качестве входных и выходных данных каждого задания. В следующей части `pipeline.yml` будут созданы описания нашего задания, использующие ресурсы, определенные в примере 15.3.

Пример 15.3. Ресурсы pipeline.yml для микросервиса учетных записей

```
# Concourse pipeline definition for the account service.
---
resource_types:
- name: maven-resource
  type: docker-image
  source:
    repository: patrickcrocker/maven-resource
    tag: latest
resources:
- name: account-microservice
  type: git
  source:
    uri:  (1)
    branch: master
```

```
    paths:
    - ./continuous-delivery-basics/account-parent/cd-account-microservice
- name: snapshot
  type: maven-resource
  source:
    url:
    artifact: cnj:cd-account-microservice:jar
    username:
    password:
- name: version
  type: semver
  source:
    driver: git
    initial_version: 1.0.0-RC.0
    uri:
    branch: version
    file: account-microservice
    private_key:
- name: cf-deploy-release
  type: cf
  source:
    api:
    username:
    password:
    organization:
    space:
    skip_cert_check: false
- name: pipeline-tasks
  type: git
  source:
    uri: https://github.com/Pivotal-Field-Engineering/pipeline-tasks
    branch: master
```

❶ Фигурные скобки служат признаком параметров и секретных данных, которые могут быть внедрены с помощью команды `fly set-pipeline`.

В примере 15.4 показаны определения заданий и задач для конвейера микросервиса учетных записей. Задание сборки (`build`) начнется с клонирования исходного кода микросервиса учетных записей и использования его результата в качестве ввода в задачу `unit`. Взглянем на эту задачу: в результате ее выполнения создается блочный тест `unit-test` микросервиса учетных записей с помощью исходного кода, чей клон был создан на основе кода из Git-хранилища (пример 15.5).

Пример 15.4. Задания и задачи pipeline.yml для микросервиса учетных записей

```
jobs:
- name: build
  max_in_flight: 1
  plan:
  - get: account-microservice
    trigger: true
```

```
  - task: unit
    file: account-microservice/concourse/unit/unit.yml
  - get: version
    params: { pre: RC }
  - put: snapshot
    params:
      file: release/cd-account-microservice.jar
      pom_file: account-microservice/source/pom.xml
      version_file: version/version
  - put: version
    params: { file: version/version }
- name: release
  serial: true
  plan:
  - get: snapshot
    trigger: true
    passed: [build]
  - get: account-microservice
    passed: [build]
  - get: version
    passed: [build]
  - get: pipeline-tasks
  - task: prepare-manifest
    file: account-microservice/concourse/release/prepare.yml
    params:
      MF_PATH: ../release-output/cd-account-microservice.jar
      MF_BUILDPACK: java_buildpack
  - task: prepare-release
    file: account-microservice/concourse/release/release.yml
  - put: cf-deploy-release
    params:
      manifest: task-output/manifest.yml
```

Пример 15.5. Задача unit, при выполнении которой будет создан и протестирован микросервис учетных записей

```
# This task will build and run the unit tests specified in the source code
# of the microservice.
---
platform: linux
image_resource:
  type: docker-image
  source:
    repository: maven
    tag: alpine
inputs:
- name: account-microservice
outputs:
- name: release
run:
  path: sh
  args:
```

```
  - -exc
  - |
    cd account-microservice/source \
    && mvn clean package \
    && mv target/cd-account-microservice.jar \
    ../release/cd-account-microservice.jar
```

В примере 15.5 показано определение задачи `unit`, в результате выполнения которой произойдет сборка микросервиса учетных записей и его блочное тестирование с помощью `unit-test`. Каждая задача запускается внутри Docker-контейнера, и ее запуск приводит к подключению томов ввода и вывода, которые позволяют передавать состояние каждой задачи или задания, имеющихся в конвейере. В данном примере сборка JAR-компонента микросервиса учетных записей происходит только при условии прохождения тестов. В результате получается, что JAR-компонент станет вводом для следующей задачи, которая подготовит предварительную сборку (snapshot build) и развернет ее в хранилище предварительных сборок Maven.

Для каждого созданного вами конвейера у вас, вероятно, будут параметры, содержащие регистрационную и секретную информацию. При создании нового конвейера у вас будет возможность предоставить в качестве входного параметра файл конфигурации. В проекте, приводимом в качестве примера, мы включили в каталог `account-parent/concourse` сценарий `set-pipeline.sh`. Содержимое последнего для микросервиса учетных записей показано в примере 15.6.

Пример 15.6. При выполнении кода файла set-pipeline.sh создается конвейер микросервиса учетных записей

```
#!/usr/bin/env bash

fly -t lite set-pipeline -p account-microservice -c pipeline.yml \
  -l .pipeline-config.yml
```

При создании нового конвейера вы получите определение вашего конвейера и файл конфигурации, включающий регистрационные и секретные данные. Содержимое файла `.pipeline-config.yml` показано в примере 15.7.

Пример 15.7. Файл конфигурации .pipeline-config.yml для микросервиса учетных записей

```
# The Cloud Foundry credentials where the microservice will be deployed
cf-url: https://api.run.pivotal.io
cf-username: replace
cf-password: replace
cf-org: replace
cf-space: replace

# The Maven repository where versioned artifacts will be published
artifactory-url: https://cloudnativejava.jfrog.io/cloudnativejava/libs-
release-local/
artifactory-username: replace
```

```
artifactory-password: replace

# The git repository containing the microservice source code
git-source-repository-url: https://github.com/cloud-native-java/continuous-
delivery

# The git repository containing the version file
git-version-repository-url: git@github.com:cloud-native-java/continuous-
delivery.git
git-version-private-key: |
  -----BEGIN RSA PRIVATE KEY-----
  REPLACE
  -----END RSA PRIVATE KEY-----
```

В примере 15.7 показан файл конфигурации, содержащий параметры конфигурации для конвейера микросервиса учетных записей. Прежде чем запускать конвейер, вам следует обновить эти регистрационные данные, или же конвейер не сможет принять завершенный вид. Первые три элемента регистрационных данных могут сказать все сами за себя, а вот `git-version-private-key` нуждается в дополнительных пояснениях. Параметр `git-version-private-key` предоставляется конвейеру для того, чтобы задача по развертыванию могла штамповать версию выпуска при каждом запуске конвейера.

На рис. 15.5 показана ветвь `version` созданного для нашего примера хранилища, содержащая два файла версий: один для микросервиса учетных записей, а другой — для микросервиса пользователей.

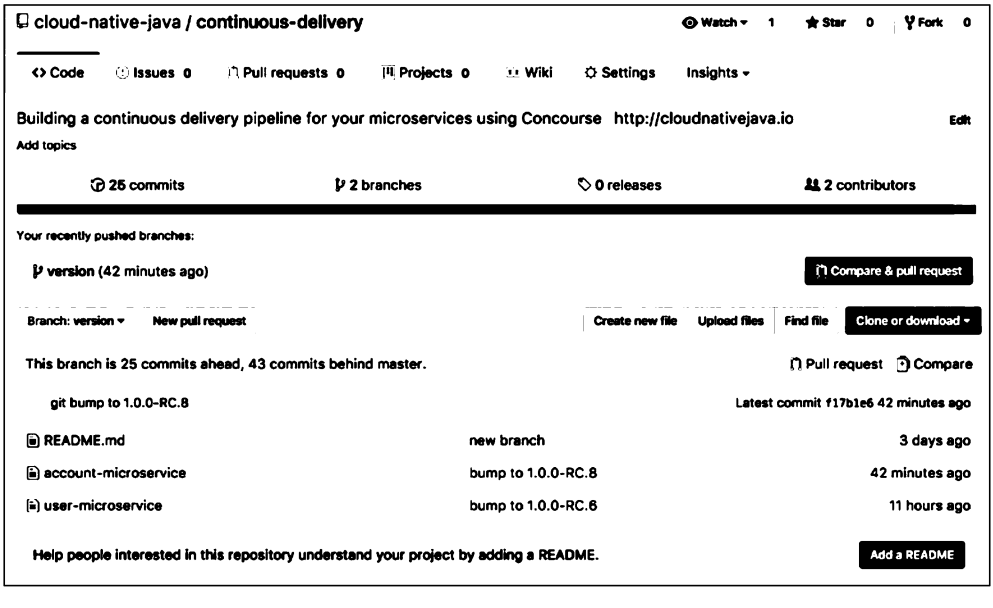


Рис. 15.5. Ветвь version в GitHub-хранилище, созданном для нашего примера

Теперь каждый из этих конвейеров будет запускаться независимо от другого и при каждом прохождении по конвейерам этапа тестирования соответствующий файл будет «перештампован» из текущей версии в новую. Данный метод присваивания версий даст событие регистрационной записи, которым можно будет воспользоваться для автоматического отката развертывания при любом неблагоприятном развитии событий. Именно поэтому нам нужно предоставить `git-version-private-key`: это позволит задаче конвейера увеличить на единицу значение версии микросервиса при каждом запуске конвейера.

Запустим конвейер из имеющегося в Concourse веб-интерфейса администратора. Перейдите в используемом вами браузере на веб-интерфейс вашего экземпляра Concourse (который будет изменяться в зависимости от вашей установки).

Запуск конвейера

После перехода на веб-интерфейс Concourse вам будет предоставлено боковое меню со списком созданных вами конвейеров. После создания конвейера `account-microservice` вы увидите, что он находится в готовности к запуску, о чем свидетельствует синяя подсветка (рис. 15.6).

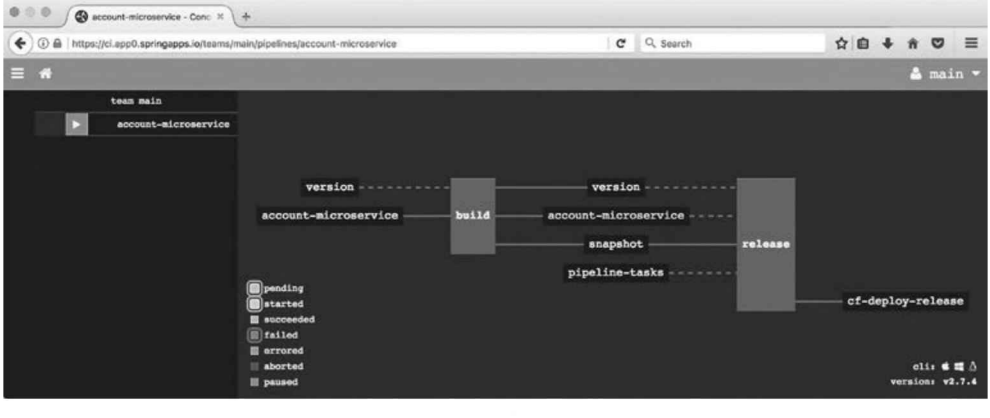


Рис. 15.6. Конвейер account-microservice готов к запуску

На рис. 15.6 показан конвейер перед запуском. Когда конвейер запущен, это означает, что любой внешний инициатор, например новая отправка данных в Git-хранилище, побудит конвейер к началу выполнения первой задачи.

Непрерывное развертывание

Каждая задача в задании может быть автоматически запущена с помощью ресурса Concourse, такого как новая отправка данных в Git-хранилище. На рис. 15.7 задание сборки `build` обведено светлым (в программе — желтым) контуром. Это

значит, что новая отправка данных в хранилище `account-microservice` инициировала новую сборку.

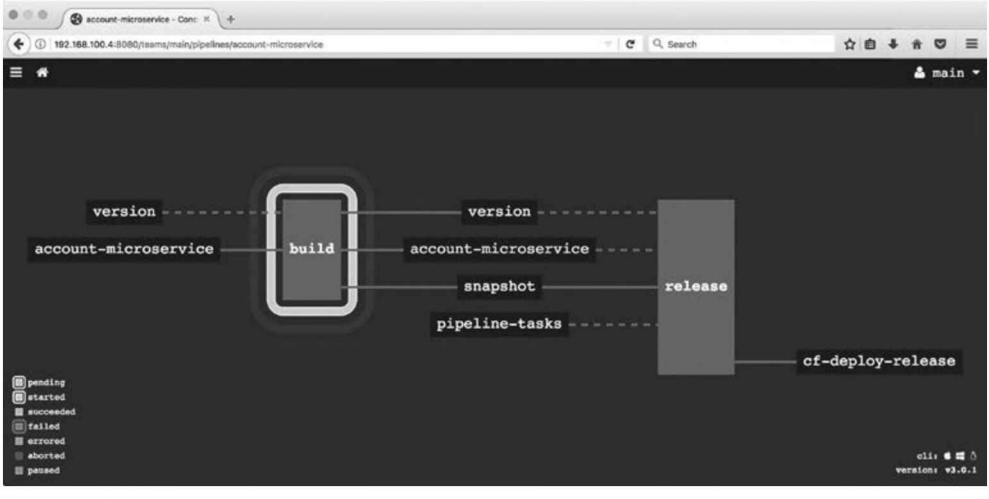


Рис. 15.7. Запуск конвейера account-microservice после новой отправки данных в Git-хранилище

Еще раз посмотрим на определение Concourse-конвейера микросервиса учетных записей, чтобы разобраться в том, как все работает (пример 15.8).

Пример 15.8. Фрагмент кода из определения конвейера account-microservice

```
resources:
- name: account-microservice    ❶
  type: git
  source:
    uri: https://github.com/cloud-native-java/continuous-delivery
    branch: master
jobs:
- name: build
  plan:
  - get: account-microservice    ❷
    trigger: true    ❸
  - task: unit
    file: account-microservice/concourse/unit/unit.yml
```

❶ Описание Git-ресурса, который будет инициировать новую сборку.

❷ Извлечение самой последней отправки данных на главную ветвь из Git.

❸ Установка значения `true` говорит о том, что инициироваться эта задача будет при любой новой отправке данных.

В примере 15.8 показан фрагмент кода из файла `pipeline.yml`, в котором дается описание Concourse-конвейера для микросервиса учетных записей. В этом

фрагменте показано, как можно сообщить Concourse о необходимости автоматически инициировать задачу из задания по факту новой отправки данных в Git-хранилище.

Описанное выше — пример конвейера *непрерывного развертывания*, а это значит, что конвейер будет автоматически пытаться выполнить сборку, тестирование и развертывание нового кода после каждой отправки данных. Процесс отличается от непрерывной поставки тем, что после каждой новой отправки будет подготовлена для развертывания новая сборка, которая не подвергнется автоматическому развертыванию. Затем последняя задача будет ожидать нажатия кнопки, запускающей развертывание, что сообщит Concourse о необходимости подготовить самую последнюю сборку к развертыванию в производственной среде.

Сборка и тестирование

После инициирования сборки задание `unit` получит входные данные, указанные в файле определений `pipeline.yml`. Если в веб-интерфейсе Concourse щелкнуть на задании `build`, то появится возможность наблюдать состояние сборки, включая поток стандартного устройства вывода (`stdout`) из Docker-контейнера, выполняющего задачу сборки.

На рис. 15.8 показано, что сборка стартовала с входными данными для исходного кода микросервиса учетных данных и текущей версии выпуска. Эти входные данные будут переданы задаче `unit`, в рамках которой выполняются сборка и тестирование исходного кода проекта Spring Boot с помощью Maven.

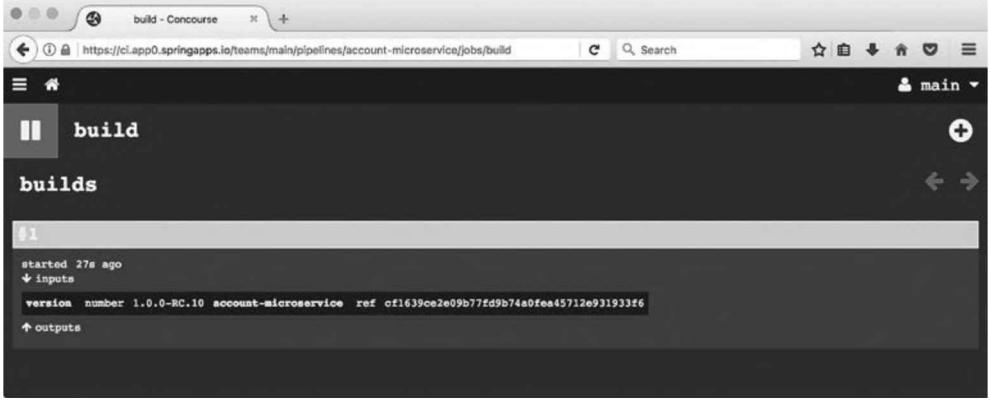


Рис. 15.8. Первая сборка инициируется в качестве входных данных исходным кодом

На рис. 15.9 показан поток вывода из Docker-контейнера сразу после того, как процесс сборки Maven приступит к загрузке зависимостей микросервиса учетных записей.

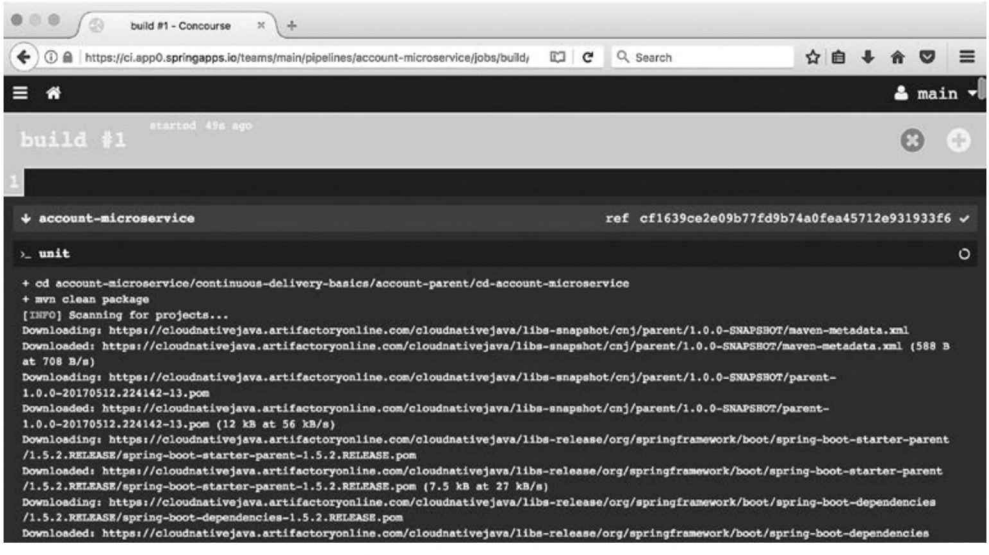


Рис. 15.9. Задача unit выполняет сборку и блочное тестирование проекта Spring Boot

Компонент Maven с присвоенной версией

После запуска и прохождения блочных тестов получившаяся сборка может быть автоматически подготовлена в качестве компонента выпуска с присвоенной ему версией. Как уже ранее упоминалось, данный конвейер использует хранилище компонентов Maven для содержания компонентов с присвоенными им версиями, которые будут развернуты на платформе Cloud Foundry. На рис. 15.10 показано конечное состояние успешно завершенного задания сборки.

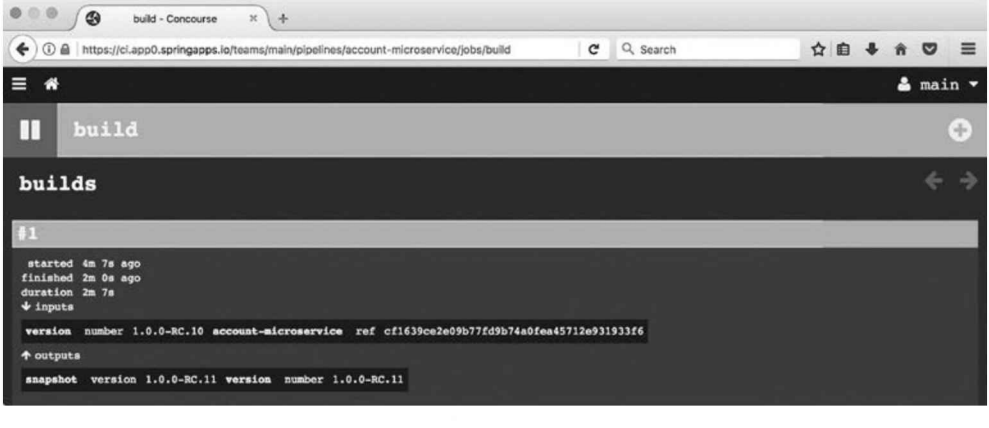


Рис. 15.10. Сборка завершается, и на выходе получается компонент развертывания с присвоенной ему версией

Если сборка сорвется, то вы получите уведомление об ошибке, а задание будет выделено красным цветом. В данном случае мы видим, что сборка и тестирование прошли успешно и компонент выпуска был выгружен в хранилище компонентов с версией, чье значение было увеличено на единицу.

Развертывание на платформе Cloud Foundry

Финальным этапом на конвейере `account-microservice` станет подготовка выпуска к развертыванию на платформе Cloud Foundry. В последнем задании была произведена сборка компонента развертывания нашего приложения Spring Boot с присвоенной ему версией и его выкладывание в хранилище. Задача `release` будет инициирована появлением новой версии в хранилище Maven и приступит к развертыванию на платформе Cloud Foundry (рис. 15.11).

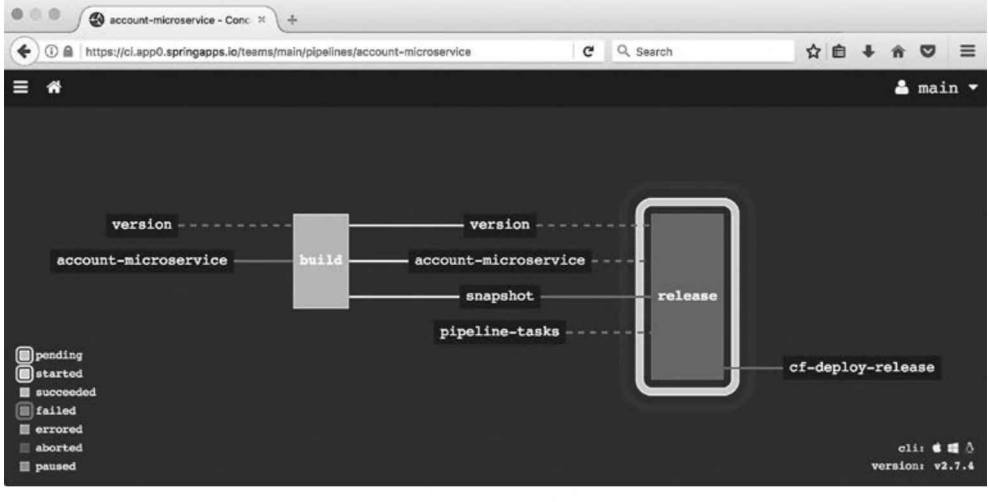


Рис. 15.11. После выполнения задачи сборки (build) новый выпуск развертывается на платформе Cloud Foundry

На рис. 15.11 показано, что задание выпуска (`release`) уже запущено. Как часть этого задания, текущий опубликованный и снабженный версией компонент из задания сборки (`build`) будет подготовлен к выпуску на платформе Cloud Foundry. Завершающим выводом задания `release` является `cf-deploy-release`. Он станет принимать файл манифеста Cloud Foundry, созданный с помощью параметров, предоставленных в определении конвейера. В примере 15.9 показано, как выглядит план задания `release`.

Пример 15.9. Задание release из определения конвейера account-microservice

```
resources:
# ...
- name: snapshot
```

```
  type: maven-resource
  source:
    url:
    artifact: cnj:cd-account-microservice:jar
    username:
    password:
- name: cf-deploy-release
  type: cf # the Concourse resource for deploying to Cloud Foundry
  source:
    api: # \
    username: # \
    password: # \
    organization: # \
    space: # \
    skip_cert_check: false
# ...
jobs:
- name: release
  serial: true
  plan:
  - get: snapshot
    trigger: true (1)
    passed: [build] (2)
  - get: account-microservice
    passed: [build]
  - get: version (3)
    passed: [build]
  - get: pipeline-tasks # Resource used to create a custom manifest.yml
  - task: prepare-manifest (4)
    file: account-microservice/concourse/release/prepare.yml
    params:
      MF_PATH: ../release-output/cd-account-microservice.jar
      MF_BUILDPACK: java_buildpack
  - task: prepare-release
    file: account-microservice/concourse/release/release.yml (5)
  - put: cf-deploy-release
    params:
      manifest: task-output/manifest.yml (6)
```

(1) Задание `release` инициируется появлением новой версии в хранилище компонентов.

(2) Выполнение `release` может быть начато только после успешного прохождения задания `build`.

(3) Получение новой версии, выведенной при выполнении задания `build`.

(4) Подготовка `manifest.yml`, пользовательского файла развертывания на платформе Cloud Foundry.

(5) Перемещение компонента с присвоенной версией в каталог подготовленных выпусков.

(6) Развертывание компонента с присвоенной версией, описание которого дано в файле `manifest.yml`, на платформе Cloud Foundry.

В примере 15.9 можно найти фрагмент кода с описанием задания `release` в определении `pipeline.yml`, относящемся к конвейеру `account-microservice`. Задание `release` управляет серией сложных действий, для чего требуется, чтобы каждый из вводов проходил как вывод из задания `build`. Следующий этап заключается в создании пользовательского файла `manifest.yml`, востребуемого Concourse-ресурсом `CF` для описания нового развертывания на платформе Cloud Foundry. `CF`-ресурс, названный `cf-deploy-release`, станет выводом задания `release` и приведет к развертыванию новой версии `account-microservice` в производство.

На рис. 15.12 показан пример успешно выпущенной версии `account-microservice`.

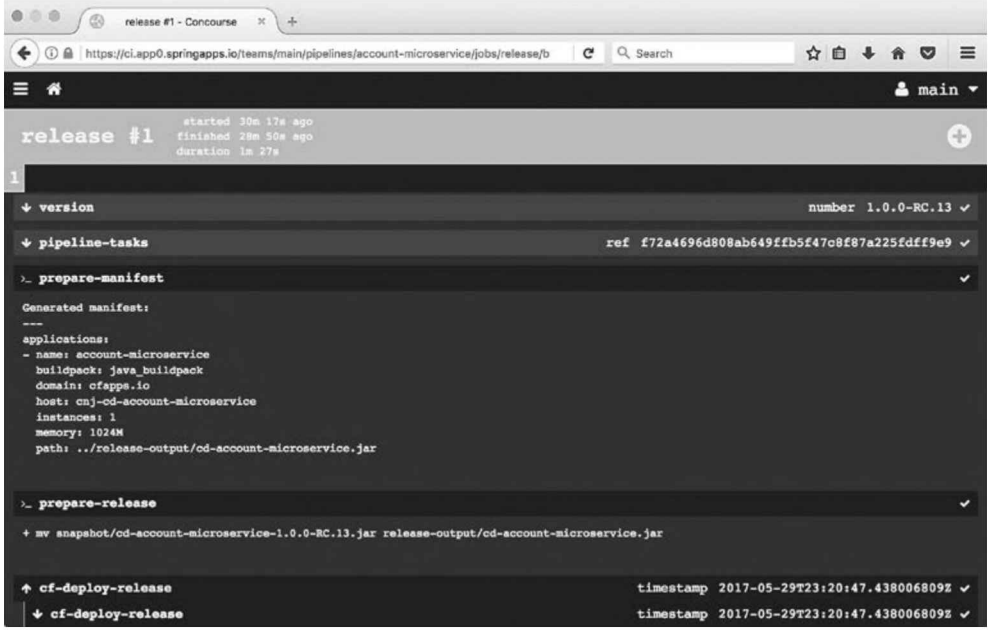


Рис. 15.12. Задание release успешно приготовило новое развертывание на платформе Cloud Foundry

Этот выпуск не был развертыванием с нулевым временем простоя. Задание release можно настроить под сине-зеленое развертывание с нулевым временем простоя, установив новый компонент с присвоенной версией на подготовленное место с последующей заменой маршрута, который ведет к приложению Cloud Foundry.

Непрерывная интеграция

Мы изучили порядок установки основного конвейера развертывания для микросервиса учетных записей. Но у этого сервиса не было никакого комплексного тестирования. Concourse может использоваться для установки комплексной среды

интеграции в целях выполнения сквозного тестирования или проведения основного комплексного теста с помощью опорных сервисов, например базы данных. В следующем разделе мы намерены установить новый конвейер для `user-service`, от которого будет зависеть `account-microservice`. В этом новом конвейере мы настроены изучить порядок установки дополнительного задания `integration`, в ходе которого выполнится тестирование на основе потребительского контракта для наших двух микросервисов.

Тестирование контрактов, ориентированных на потребителя

Непрерывное развертывание микросервисов в отсутствие системы сдержек и противовесов, гарантирующих отсутствие бесконечных выпусков командами каких-либо критических изменений, быстро приведет к абсолютно неприемлемым результатам. В главе 4 рассматривался пример тестирования контрактов, ориентированных на потребителя. Теперь же мы собираемся повторить его на простом конвейере Concourse, созданном для сервиса учетных данных. Мы включим этап тестирования контрактов, ориентированных на потребителя, чтобы убедиться, что изменения не нарушат работы потребителей. Целью этого нового конвейера станет непрерывная интеграция *сервиса пользователей* с тестированием контракта, ориентированного на потребителя, опубликованного *сервисом учетных записей*.

Для этого нужно создать хранилище исходного кода, где *сервис учетных записей* сможет публиковать заглушки контрактов, ориентированных на потребителя *сервиса пользователей*. Как уже упоминалось в главе 4, тесты контрактов, ориентированных на потребителя, создаются потребителями в зависимости от сервиса. Каждый из наших потребителей будет отвечать за публикацию их тестов контрактов, ориентированных на потребителя в назначенном хранилище. Приведем такой пример.

Мы являемся владельцами *сервиса учетных записей* и зависим от *сервиса пользователей*. Будучи потребителями, мы хотим убедиться, что в сервисе пользователей не было выпущено каких-либо нарушающих нашу работу изменений. Чтобы автоматизировать это комплексное тестирование, опубликуем наши потребительские тесты в назначенное хранилище, владельцем которого является *сервис пользователей*. Тогда наша ответственность как потребителя заключается в публикации наших тестов, ориентированных на потребителя, в комплексном хранилище того сервиса, от которого мы зависим. В противном случае *сервис пользователей* не будет знать об ожиданиях с нашей стороны, то есть со стороны *сервиса учетных записей*.

Конвейер микросервиса пользователей. А теперь возьмем конструкцию конвейера, которую мы создали ранее для *сервиса учетных записей*, и изменим набор правил, включив тестирование контрактов, ориентированных на потребителя для *сервиса пользователей*. Конвейер для микросервиса пользователей показан на рис. 15.13.

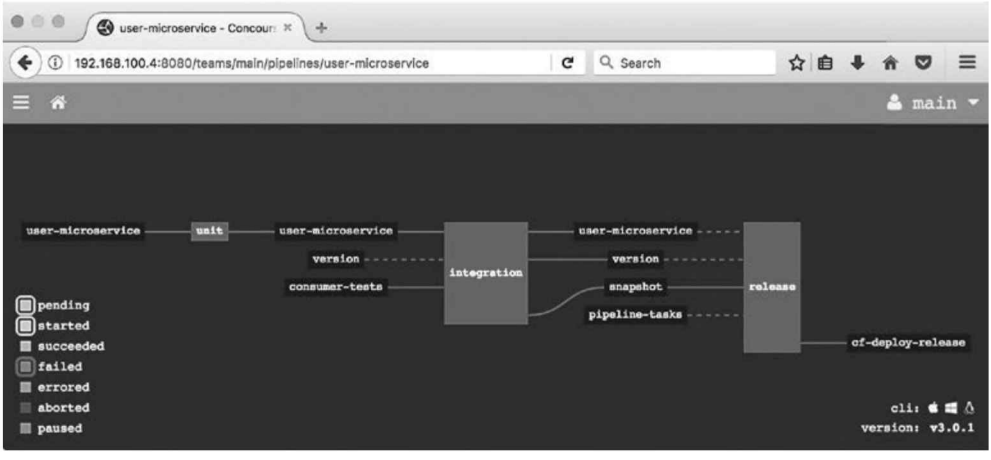


Рис. 15.13. Concourse-конвейер для микросервиса пользователей

В данном сервисе у нас есть дополнительный этап конвейера: задание `integration`. В нем, прежде чем собрать в единое целое новый компонент выпуска и выложить его в хранилище компонентов Maven, в отношении микросервиса пользователей будет проведен набор тестов, ориентированных на потребителя.

На рис. 15.14 показан каталог исходного кода микросервиса пользователей.

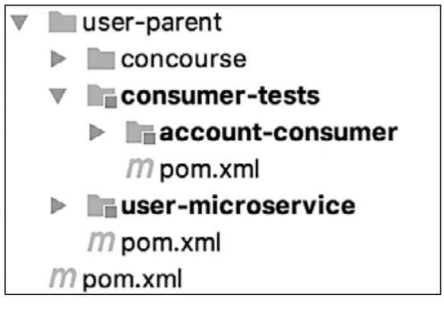


Рис. 15.14. Каталог исходного кода микросервиса пользователей

Для микросервиса пользователей у нас будет еще один дополнительный каталог, который содержит коллекцию тестов контрактов, ориентированных на потребителя, опубликованных потребителями. Прежде чем это сможет произойти, нам нужно опубликовать заглушку контракта из микросервиса пользователей, чтобы другие сервисы могли записать свои тесты в отношении данного контракта.

На рис. 15.15 показана заглушка контракта, ориентированного на потребителя, которая использует проект Spring Cloud Contract, рассмотренный в главе 4.

При публикации микросервиса пользователей сервис может применять эту заглушку для запуска встроенного сервера приложений, который имитирует поведение микросервиса пользователей.

На рис. 15.16 показан тест контракта, ориентированного на потребителя, который создан микросервисом учетных записей. Задание `integration` в конвейере микросервиса пользователей будет запускать каждый из потребительских тестов, имеющихся в данной папке, для гарантии того, что новые изменения не нарушат

работу потребителей. Ответственность за написание этих тестов возлагается на каждого потребителя, и если тест не будет пройден, то микросервис пользователей не сможет выпустить критические изменения в производство.

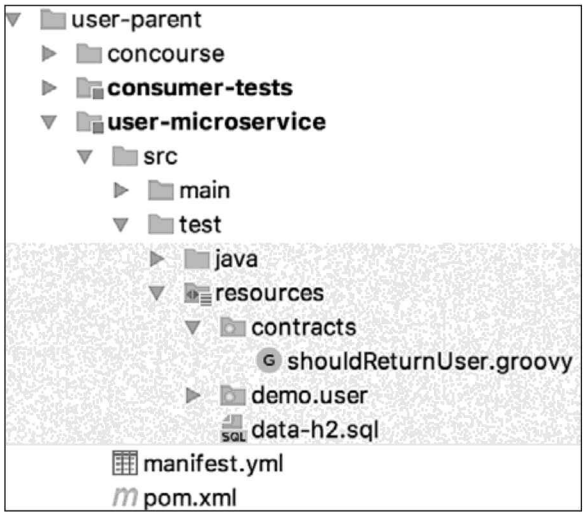


Рис. 15.15. Заглушка контракта, ориентированного на потребителя, опубликованная микросервисом пользователей

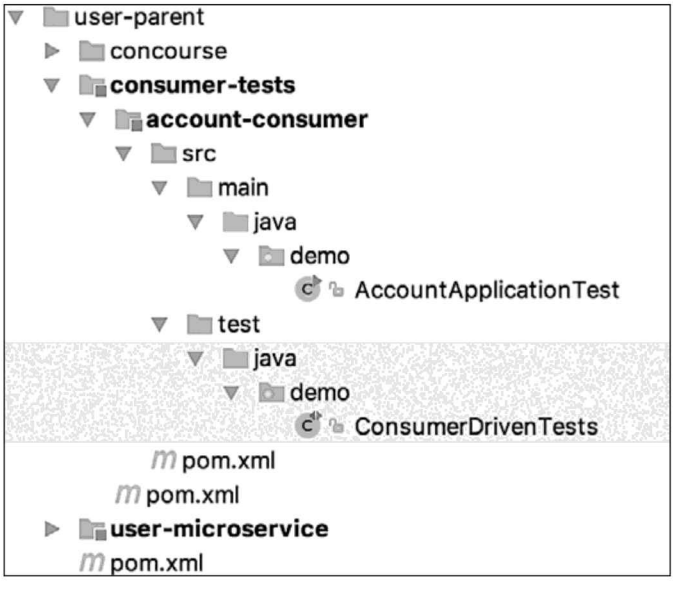


Рис. 15.16. Микросервис учетных записей создает потребительские тесты в отношении микросервиса пользователей

Теперь посмотрим на задачу `consumer-tests`, являющуюся частью задания `integration` в Concourse-конвейере микросервиса пользователей (пример 15.10).

Пример 15.10. Задание integration для микросервиса пользователей

```
# This task will run the integration tests and consumer-driven contract tests of
# the user microservice and its consumers.
---
platform: linux
image_resource:
  type: docker-image
  source:
    repository: maven
    tag: alpine
inputs:
- name: consumer-tests
outputs:
- name: release
run:
  path: sh
  args:
  - -exc
  - |
    cd consumer-tests/user-microservice \                    (1)
    && mvn clean install -DskipTests \                       (2)
    && mv target/user-microservice.jar \
      ../../release/user-microservice.jar \                  (3)
    && cd ../consumer-tests \                                (4)
   && mvn clean package                                      (5)
```

(1) Переход в каталог приложения Spring Boot, являющегося микросервисом пользователей.

(2) Установка заглушек контрактов микросервиса пользователей в локальное хранилище Maven.

(3) Размещение JAR-компонента в выходном каталоге выпусков для следующей задачи.

(4) Переход к каталогу потребительских тестов.

(5) Запуск всех потребительских тестов в отношении самых последних заглушек контрактов, ориентированных на потребителя.

В примере 15.10 показано определение задачи `consumer-tests.yml`, включенной в каталог `concourse` микросервиса пользователей. В этой задаче тесты, ориентированные на потребителя, которые были созданы потребителями микросервиса, будут запущены в отношении новых изменений. Если один из этих тестов не будет завершен успешно, то конвейер даст сбой, что помешает микросервису пользователей выпустить изменения. Применяя конвейер непрерывной поставки для автоматизации такого рода комплексных тестов, владельцы микросервисов

должны заранее продумывать свои решения по критическим изменениям, прежде чем получить возможность развернуть свои приложения в среде эксплуатации.

Пути папок в примерах, использованных в данной главе, могут отличаться от путей, показанных в примерах кода, сопровождающих эту книгу. Это сделано для того, чтобы не затруднять чтение приводимых здесь фрагментов кода. Во входных ресурсах задачи будет задействован полный путь к микросервису. Чтобы не усложнять поиск, в данной книге предусмотрено применение монолитных хранилищ. Для ваших микросервисов, предназначенных для работы в производственной среде, не рекомендуется включать несколько микросервисов в одно и то же Git-хранилище.

Данные

Упоминая о сине-зеленом развертывании, мы не затрагивали вопрос миграции баз данных. Поскольку единого решения не существует, указанную проблему здесь можно рассмотреть, не прибегая к особому углублению в подробности.

Инструменты миграции схемы СУРБД, такие как Flyway (https://flywaydb.org/) и Liquibase (http://www.liquibase.org/), обслуживают манифест в каждой базе данных, из которой были применены мутации схемы, что позволяет вполне определенно обновить версию схемы при условии определения доступности новой версии. Если все, что вы делали, сводилось к добавлениям новых столбцов, таблиц и записей, то при необходимости выполнить откат к прежнему состоянию данных никаких проблем возникать не должно. У вас по-прежнему будут все прежние записи и схема.

А что произойдет при развертывании новой версии сервиса (v2) и использовании Flyway или Liquibase, инструментов, поддерживающих дельта-схемы управления версиями и миграцию существующих баз данных из одной версии в другую для развертывания схемы в версию 2.0? Как разовьются события, если что-то пойдет не так и схема в результате обновления подвергнется разрушению (например, будут изменены столбцы или удалены записи)?

В указанных вопросах при разработке сине-зеленых развертываний следует проявлять особую осмотрительность. Вместо удаления данных нужно предусмотреть *«захоронение» записей*, при котором приложение логически удаляет запись, скажем устанавливая в столбце `DELETED` значение `TRUE`, но оставляет данные, чтобы приложение можно было «воскресить» путем снятия флага в столбце «захоронения». Можете ли вы при изменении столбца продублировать данные из старого столбца в новый, сохраняя при этом оба?

Делайте все возможное, чтобы позволить откат кода к прежней версии, поскольку никто не знает, где и когда может произойти сбой. С практической точки зрения вам не следует поддерживать откат к каждой предыдущей версии, достаточно

иметь такую возможность относительно предпоследней версии. Команды зачастую оставляют себе достаточно широкое поле для отката к последней известной версии схемы, например к версии 1. В версии 2 *устаревшие* столбцы и записи остаются нетронутыми. В версии 3 можно применять все ликвидационные действия относительно версии 1, что делает миграцию из нее необратимой.

К производству!

> Программное обеспечение, о котором можно сказать, что оно «завершено», похоже на газон, о котором говорят, что он «скошен».
>
> *Джим Бенсон (Jim Benson)*

В данной книге мы рассмотрели порядок создания приложений, ориентированных на выполнение в облачной среде. Такие приложения предназначены для запуска на облачной платформе и использования облачной парадигмы. По своей конструкции они оптимизированы таким образом, чтобы в них легче было вносить улучшения и как можно быстрее получать отклики, что не делалось еще никогда. Приложения, оптимизированные для выполнения в облачной среде, относятся (как мы надеемся) к разряду тех, в которых произойдет множество революционных изменений на стадиях от концепции до ввода в эксплуатацию, при каждом из которых они будут становиться все ценнее для своих клиентов. В данной главе был рассмотрен вопрос о том, как можно свести к минимуму затраты на эти революционные преобразования, воплотив в жизнь технологию непрерывной поставки.

Приложение. Использование Spring Boot с Java EE

В данном приложении мы изучим способы интеграции приложений Spring Boot с Java EE. В нашем случае целесообразно считать Java EE обобщающим названием для набора API, а в некоторых случаях для сред выполнения кода — *серверов приложений Java EE*. Данные серверы (например, сервер приложений разработки Red Hat под названием WildFlyAS (http://wildfly.org/), прежде именовавшийся JBoss Application Server) предоставляют реализации этих API. Мы рассмотрим способы создания приложений, использующих API Java EE вне сервера приложений Java EE. Если вы сейчас создаете качественно новое приложение, то этот дополнительный материал не понадобится. Главным образом он предназначен для разработчиков, имеющих функциональные решения, зафиксированные в сервере приложений, которые нужно поместить в архитектуру микросервисов. Расширенное представление аспектов перемещения (миграции) ранее разработанных приложений на такую облачную платформу, как Cloud Foundry, с минимумом реструктуризаций, дается в главе 5.

Там, где это кажется осуществимым, Spring выступает в роли потребителя API Java EE. Самой среде ни один из них *не требуется*. По возможности Spring поддерживает использование API Java EE «по выбору», независимо от полноценного сервера приложений Java EE. Приложения Spring в идеале должны быть переносимыми между разными средами, включая встроенные веб-приложения, серверы приложений и практически любые предлагаемые платформы в виде сервисов (Platform as a Service, PaaS).

Совместимость и стабильность

Spring 4.2 (базовый выпуск для Spring Boot 1.3 и последующих версий) поддерживает Java SE 6 или более поздние версии (в частности, минимальным уровнем API является JDK 6u18, выпущенный в начале 2010 года). Для Java SE 6 и Java SE 7

компания Oracle прекратила публикацию обновлений с исправлениями проблем безопасности. Поэтому нужно переходить на использование Java SE 8.

В Spring также поддерживается версия Java EE 6+ (выпущенная в 2009 году). В практическом плане Spring 4 нацелена на Servlet 3.0+, но совместима с Servlet 2.5. Среда Spring Boot, построенная на основе Spring, лучше всего работает с Servlet 3.1 или более поздними версиями. Подход к зависимостям, практикуемый в Spring Boot, гарантирует получение по возможности самых последних и наиболее качественных версий, хотя при желании вы можете вернуться к веб-серверам более раннего поколения Servlet 3.0.

По популярности на рынке с большим отрывом неизменно лидируют Tomcat (проект, в который вложено много усилий со стороны Pivotal) компании Apache (http://tomcat.apache.org/), Jetty компании Eclipse и Wildfly компании Red Hat. Большинство разработчиков Java не применяют развертывание в полные Java EE-совместимые контейнеры, и поэтому для все возрастающего их количества, не запускающего серверы приложений Java EE, среда Spring предоставляет весьма удобные API Java EE. Просто Java EE предлагает весьма привлекательные (и практичные) API. Среди них можно упомянуть такие полезные и удобные для пользователей интерфейсы, как Servlet API, JSR 330 (`javax.inject`) и JSR 303 (`javax.validation`), JCache, JDBC, JPA, JMS, JAX-RS и т. д., поддерживаемые Spring и Spring Boot.

По мере повышения внимания со стороны разработчиков к облачной среде и, как следствие, к архитектуре, ориентированной на облачные вычисления, наблюдается снижение зависимости от использования серверов приложений Java EE. Подход по принципу «или все, или ничего» идет вразрез с технологией создания облачно-ориентированных систем, составленных из небольших узкоспециализированных масштабируемых сервисов.

Некоторые из API Java EE весьма удобны, но, как известно, развиваются крайне медленно. Из этого следует, что полезные свойства медленно развивающихся, стандартизованных API Java EE лучше всего использовать на уровнях, редко подвергающихся изменениям. Примером вполне разумного применения API может послужить спецификация JDBC, которая обеспечивает связующее звено для десятилетиями применяющихся баз данных на основе SQL. Кроме того, очень эффективным интерфейсом, предоставляющим связующее средство для приложений HTTP, является Servlet API. Спецификации JDBC и Servlet служат примерами API, не требующего частых изменений.

Исключение составляют те случаи, когда это делается *по острой необходимости*. Новые парадигмы разработки появляются непрерывно. По состоянию на середину 2017 года протокол HTTP 2, предлагающий практически бесплатное повышение производительности, уже несколько лет поддерживался большинством других платформ и технологий, но не спецификацией Servlet. Для доступа к этой функциональной возможности (на данный момент) разработчики должны использовать

в базовых Servlet-контейнерах платформозависимые API. По мере роста спроса на приложения и укрупнения наборов данных становится неприемлемой блокировка потоков, ожидающих ввода и вывода, когда работа может быть проделана незамедлительно (reactively), высвобождая ценные ресурсы. Чтобы программирование с асинхронными потоками информации (reactive programming) полностью заработало в отношении базы данных JDBC, каждая ссылка в цепочке обработки запроса должна поддерживать асинхронные потоки данных со стороны HTTP-запроса на Servlet-основе. Сегодня ничего похожего не происходит. В компании Oracle намекнули, что им хотелось бы видеть поддержку программирования с асинхронными потоками данных в последующей переработке JDBC и Java EE 8 станет поставляться с поддержкой HTTP 2. И в конечном итоге вопрос будет решен.

Программное обеспечение служит достижению определенной цели и создается, как правило, в коммерческих целях, и компании не конкурируют, полагаясь на низкокачественные решения с возрастом в несколько десятилетий. И конкретные стандарты следует выбирать там и только там, где они имеют смысл.

Рассмотрим несколько распространенных API, которые неплохо работают в приложениях Spring Boot.

Внедрение зависимостей с помощью JSR 330 (и JSR 250)

В среде Spring уже давно предлагаются различные подходы для предоставления конфигурации. Фактически среде совершенно безразлично, где именно она узнает о ваших объектах и как именно вы их соединяете. Вначале использовалась конфигурация в формате XML. Затем в Spring было введено сканирование компонентов для обнаружения и регистрации компонентов с обозначенными аннотациями наподобие `@Component` или любыми другими аннотациями, которые сами имеют эту аннотацию, такими как `@RestController` или `@Service`. В 2006 году появился проект Spring Java Configuration. В данном подходе bean-компоненты раскрывались для среды Spring в понятиях объектов, возвращаемых из методов, имеющих аннотацию `@Bean`. Эти *методы поставщика* bean-определений вы можете вызывать.

Тем временем в компании Google Боб Ли (Bob Lee) представил среду Guice, в которой так же, как и проекте конфигурации Spring Java, поддерживаются методы поставщика bean-определений. Эти два варианта, наряду с некоторыми другими идеями, развивались независимо друг от друга, и у каждого из них появилось большое сообщество приверженцев.

Пропустим 2007, 2008 и 2009 годы и период становления Java EE 6. Команда, занимавшаяся разработкой среды JBoss Seam и создавшая собственную технологию внедрения зависимостей, предприняла попытку определить стандарт под названием JSR 299 (CDI) с целью установить, что такое технология внедрения зависимостей.

Но ни Seam, ни JSR 299 *ничем* не походили на Spring или Guice, то есть на две наиболее востребованные и популярные технологии, поэтому создатель Spring Род Джонсон и создатель Guice Боб Ли предложили спецификацию JSR 330. В ней определяется основной набор аннотаций для тех сред внедрения зависимостей, *которые оказывают влияние на код бизнес-компонентов*. Тем самым отличия каждого контейнера внедрения зависимостей друг от друга остались в реализациях самого механизма.

Вполне естественно, что JSR 330 поддерживается Spring и Guice и в конечном итоге реализациями JSR 299. Эту спецификацию зачастую фактически поддерживают и другие технологии внедрения зависимостей, например технология Dagger, оптимизированная под создание кода в ходе компилирования, и мобильные среды, подобные Android. Имея настоятельную потребность в переносимом внедрении зависимостей, воспользуйтесь спецификацией JSR 330.

При условии, что для определения, разрешения и внедрения ссылок на bean-компоненты в путях к классам имеется запись `javax.inject : javax-inject : 1`, обычно с JSR 330 будут применяться следующие аннотации.

- `@Inject` — эквивалент имеющейся в Spring аннотации `@Autowired`. С ее помощью идентифицируются внедряемые конструкторы, методы и поля.
- `@Named` в какой-то мере является эквивалентом имеющихся в Spring различных стандартных аннотаций наподобие `@Component`. Этой аннотацией помечают bean-компонент, предназначенный для регистрации с контейнером, и она может использоваться для предоставления bean-компоненту идентификатора на основе строкового значения, под которым он может быть зарегистрирован.
- `@Qualifier` может использоваться для *определения* bean-компонентов по типу (или по строковому идентификатору, String ID). По сути, она почти идентична имеющейся в Spring аннотации `@Qualifier`.
- `@Scope` — в некотором смысле аналог имеющейся в Spring аннотации `@Scope` и используется для сообщения контейнеру о характере жизненного цикла bean-компонента. Можно, к примеру, указать, что bean-компонент относится к области видимости сеанса работы, то есть должен появляться и исчезать в соответствии с HTTP-запросом.
- `@Singleton` сообщает контейнеру о том, что bean-компонент должен быть создан в виде экземпляра только один раз. В Spring это делается по умолчанию, но существует общая концепция, что стоит обеспечить поддержку этого подхода во всех реализациях.

В JSR 330 также определяется интерфейс `javax.inject.Provider<T>`. Бизнес-компоненты могут внедрять экземпляры заданного bean-компонента, `Foo`, непосредственно или с помощью `Provider<Foo>`. Это в какой-то мере служит аналогией имеющемуся в Spring типу `ObjectFactory<T>`. По сравнению с непосредственным внедрением экземпляра экземпляр `Provider<T>` может использоваться для извлечения нескольких экземпляров заданного bean-компонента, обрабатывая

«ленивые» семантические выражения, прерывая циклические зависимости и абстрагируя охватывающую область видимости.

Спецификация JSR 250, исходящая из Java EE 5, также исходно поддерживается в Spring, если типы указаны в путях к классам (в самых новых версиях JDK). Возможно, аннотации из нее вам уже встречались, если, к примеру, приходилось использовать @javax.annotation.Resource. Эти аннотации широко применяются в коде EJB 3, но нам никогда не приходилось сталкиваться с ними где-либо еще. Они могут облегчить разработчикам задачу перемещения кода из сред EJB 3, где разрешения ссылок обозначены с помощью @Resource.

Использование API Servlet в приложениях Spring Boot

Если привлечь запись `spring-boot-starter-web`, то среда Spring Boot по умолчанию совершает автоматическое конфигурирование среды выполнения веб-приложений. Эта среда в версиях Spring, предшествующих версии 5.0, представляет собой такой Servlet-контейнер, как Apache Tomcat, Undertow компании Red Hat и Jetty компании Eclipse.

В среде Spring, начиная с версии 5.0, также поддерживается оперативная среда выполнения, построенная на основе использования библиотеки Netty. Но мы отвлекаемся.

Все эти варианты прекрасны. Наиболее распространен Apache Tomcat, но вам ничто не мешает, сообразуясь с имеющимися потребностями, воспользоваться чем-нибудь другим. В Spring Boot поддерживаются решения таких общих вопросов, как указание порта с помощью свойства `server.port`, пути к контексту с применением `server.context-path`, конфигурирование поддержки SSL благодаря `server.ssl.*` и включение GZip-сжатия с помощью `server.compression.*` . В частности, GZip-сжатие принимает нерационально использующее пространство информационного наполнения, подобное данным в формате JSON, доставленным по протоколу HTTP, и выполняет его сжатие. Браузеры будут знать, как распаковать такие ресурсы. Это добавляет эффективности в работе с объемным и статическим информационным наполнением, таким как фрагменты кода JavaScript или CSS. При необходимости сжать все статические ресурсы (имеются в виду те, что среда Spring Boot будет обслуживать из каталогов с общим обозначением `src/main/resources/*`) следует всего лишь дать указание `spring.resources.chain.gzipped=true`. Если нужно сжать динамические конечные точки, такие как информационное наполнение в формате JSON, можно воспользоваться свойствами вида `server.compression.*`. В примере П.1 показано, как можно сжать все результаты, выданные в формате JSON.

Пример П.1. GZip-сжатие всего информационного наполнения, выданного в формате JSON

```
server.compression.enabled=true
server.compression.mime-types=application/json
```

Убедиться в том, что это работает, можно при посещении конечной точки без указания о допустимости кодирования, `accept-encoding`, и с таким указанием. Наше учебное приложение отдает конечную точку Spring MVC под `/mvc/hi`, что приводит к возврату выходной информации в формате JSON. Вызовите эту конечную точку без указания заголовка `accept-encoding` и получите чистый блок информации в формате JSON. А теперь вызовите ее с заголовком `accept-encoding`, указав `gzip`, и получите двоичные (сжатые) данные, которые для прочтения придется передать какому-нибудь средству типа `gunzip` (пример П.2).

Пример П.2. Вызов конечной точки REST с указанием и без указания заголовка accept-encoding

```
> curl http://localhost:8080/mvc/hi
{"greetings":"Hello, world!"}

> curl -H"accept-encoding: gzip" http://localhost:8080/mvc/hi
?VJ/JM-??K/V?R?H????Q(?/?IQT????

> curl -H"accept-encoding: gzip" http://localhost:8080/mvc/hi | gunzip
{"greetings":"Hello, world!"}
```

Существуют также свойства, характерные для того или иного контейнера. Разумеется, они никак не перекликаются со свойствами других контейнеров. Если развертывание проводится в Undertow, то можно указать, к примеру, `server.undertow.worker-threads=10`, чтобы ограничить количество созданных рабочих потоков. При проведении развертывания в Tomcat можно воспользоваться указанием `server.tomcat.max-threads=10`, установив максимальное количество запущенных потоков. Если развертывание осуществляется в проект Eclipse Jetty, то можно задействовать указания `server.jetty.acceptors` и `server.jetty.selectors` с целью ограничить количество обслуживаемых Jetty потоков-получателей и потоков-селекторов.

Общие свойства и свойства, характерные для конкретного контейнера, позволяют выразить основную часть тех настроек, которые могли быть у вас до этого. Если же ваши надежды на присутствие необходимых настроек не оправдываются и вы желаете программным путем настроить и опросить встроенный механизм сервлет-контейнера, то обратитесь к реализации `EmbeddedServletContainerCustomizer`, интерфейсу обратного вызова, который предоставляет доступ к реализации контейнера, создаваемого в соответствии с настройками Spring Boot. В этой среде поддерживаются также интерфейсы обратного вызова, нацеленные на работу с контейнерами. Эти обратные вызовы, характерные для контейнеров, дают возможность переопределить или дополнить конфигурацию компонентов, задействованных в создании самого приложения.

В примере П.3 будет произведено разыменование пользовательских обратных вызовов, относящихся к контейнерам, и выполнен их перебор. Вы можете внести

собственные настройки, добавив их к указанным коллекциям экземпляра контейнера. Это должен быть метод вида `container#add*` для той настройки, к которой вам нужно получить доступ.

Пример П.3. Работа со встроенным контейнером в EmbeddedServletContainerCustomizer

```
package servlets;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
//@formatter:off
import org.springframework.boot.context.embedded.AbstractEmbeddedServletContainer
  Factory;
import org.springframework.boot.context.embedded.ConfigurableEmbeddedServlet
  Container;
import org.springframework.boot.context.embedded.EmbeddedServletContainer
  Customizer;
import org.springframework.boot.context.embedded.jetty.JettyEmbeddedServlet
  ContainerFactory;
import org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedServlet
  ContainerFactory;
import
    org.springframework.boot.context.embedded.undertow.UndertowEmbeddedServlet
      ContainerFactory;
//@formatter:on
import org.springframework.stereotype.Component;

@Component
class ContainerAnalyzer implements EmbeddedServletContainerCustomizer {

 private final Log log = LogFactory.getLog(getClass());

 @Override
 public void customize(ConfigurableEmbeddedServletContainer c) {

  this.log.info("inside " + getClass().getName());

  ❶
  AbstractEmbeddedServletContainerFactory base =
    AbstractEmbeddedServletContainer Factory.class
   .cast(c);
  this.log.info("the container's running on port " + base.getPort());
  this.log.info("the container's context-path is " + base.getContextPath());

  ❷
  if (UndertowEmbeddedServletContainerFactory.class.isAssignableFrom(c
   .getClass())) {
   UndertowEmbeddedServletContainerFactory undertow = UndertowEmbeddedServlet
     ContainerFactory.class
    .cast(c);
   undertow.getDeploymentInfoCustomizers().forEach(
```

```
    dic -> log.info("undertow deployment info customizer " + dic));
   undertow.getBuilderCustomizers().forEach(
    bc -> log.info("undertow builder customizer " + bc));
  }

  ❸
  if (TomcatEmbeddedServletContainerFactory.class
   .isAssignableFrom(c.getClass())) {
   TomcatEmbeddedServletContainerFactory tomcat = TomcatEmbeddedServlet
     ContainerFactory.class
    .cast(c);
   tomcat.getTomcatConnectorCustomizers().forEach(
    cc -> log.info("tomcat connector customizer " + cc));
   tomcat.getTomcatContextCustomizers().forEach(
    cc -> log.info("tomcat context customizer " + cc));
  }

  ❹
  if (JettyEmbeddedServletContainerFactory.class.isAssignableFrom(
      c.getClass())) {
   JettyEmbeddedServletContainerFactory jetty = JettyEmbeddedServletContainer
     Factory.class
    .cast(c);
   jetty.getServerCustomizers().forEach(
    cc -> log.info("jetty server customizer " + cc));
  }
 }

}
```

❶ Для абстрагирования базового типа и получения более широких возможностей, чем вызов различных методов `set*`, полезно будет выполнить нисходящее преобразование типа.

❷ Существуют реализации, применяемые к конкретному контейнеру для Undertow...

❸ ...и Apache Tomcat...

❹ ...и Eclipse Jetty.

В Spring Boot выполняется автоматическая поддержка встроенного сервлет-контейнера с последующей регистрацией соответствующего пакета средств среды Spring с помощью программного механизма регистрации Servlet 3.0. В Spring Boot будет зарегистрирован диспетчер сервлетов Spring MVC `DispatcherServlet`, любые необходимые фильтры Spring Security и по мере надобности любые другие механизмы, востребованные другими частями экосистемы Spring. Все это делается без вашего участия и работает к тому моменту, когда вы приступите к написанию кода. Вам не нужно выполнять конфигурацию `web.xml` или управлять конфигурацией для самого контейнера. Если используются проекты Spring (такие как Spring MVC), то, значит, все уже настроено. К тому же вы можете передавать собственные сервлеты или фильтры. В Spring Boot будут подобающим образом зарегистри-

рованы любые bean-компоненты Spring (поступающие из методов поставщика `@Bean` либо снабженные стандартными аннотациями) типа `javax.servlet.Filter` или `javax.servlet.Servlet`, воспринимаемые в качестве фильтров или сервлетов соответственно.

Если нужно установить контроль над тем, как регистрируются сервлеты или фильтры (как определяется порядок их следования, отображения на URL, параметры и т. д.), то для этого есть два дополнительных варианта. Ваши сервлеты могут быть заключены в экземпляры `ServletRegistrationBean`, и среде Spring будет позволено взять управление на себя. Того же самого можно добиться для фильтров, воспользовавшись экземплярами `FilterRegistrationBean`. Предположим, что у нас имеется `javax.servlet.Filter` по имени `LoggingFilter` (пример П.4).

Пример П.4. И опять довольно распространенный пример регистрации

```
package servlets;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import javax.servlet.*;
import java.io.IOException;
import java.time.Instant;

class LoggingFilter implements Filter {

 private final Log log = LogFactory.getLog(getClass());

 @Override
 public void init(FilterConfig config) throws ServletException {
  this.log.info("init()");
  String initParameter = config.getInitParameter("instant-initialized");
  Instant initializationInstant = Instant.parse(initParameter);
  this.log.info(Instant.class.getName() + " initialized "
   + initializationInstant.toString());
 }

 @Override
 public void doFilter(ServletRequest req, ServletResponse resp,
  FilterChain chain) throws IOException, ServletException {
  this.log.info("before doFilter(" + req + ", " + resp + ")");
  chain.doFilter(req, resp);
  this.log.info("after doFilter(" + req + ", " + resp + ")");
 }

 @Override
 public void destroy() {
  log.info("destroy()");
 }
}
```

Экземпляр можно зарегистрировать с помощью определения bean-компонента `FilterRegistrationBean`, указывая, нужно ли фильтру поддерживать асинхронные операции, к каким путям URL применяется фильтр, и др. (пример П.5).

Пример П.5. Регистрация фильтров и сервлетов в явном виде

```
package servlets;

import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.Ordered;

import java.time.Instant;

@Configuration
public class FilterConfiguration {

 @Bean
 FilterRegistrationBean filter() {
  LoggingFilter filter = new LoggingFilter(); ❶
  FilterRegistrationBean registrationBean =
     new FilterRegistrationBean(filter);
  registrationBean.setOrder(Ordered.HIGHEST_PRECEDENCE);
  ❷
  registrationBean.addInitParameter("instant-initialized", Instant.now()
   .toString());
  return registrationBean;
 }
}
```

❶ Здесь экземпляр `Filter` создается для наших собственных потребностей. Но ничто не мешает вам управлять жизненным циклом bean-компонента также в качестве bean-компонента Spring.

❷ Здесь параметры инициализации указываются программным способом. В свою очередь, они могут поступать из конфигурации, указанной в командной строке или из сервера Spring Cloud Config Server.

Среда Spring обеспечит вызов в подходящее время методов init и destroy.

В Spring Boot даже поддерживается автоматическое обнаружение и управление компонентами, имеющими аннотацию, предоставленную Servlet 3.0: `@javax.servlet.annotation.WebServlet`, `@javax.servlet.annotation.WebFilter` и `@javax.servlet.annotation.WebListener`. Чтобы активировать эту поддержку, добавьте

к классу конфигурации Spring `@org.springframework.boot.web.servlet.ServletComponentScan`, а в идеале к тому же классу, в котором находится `@SpringBootApplication`. Среда Spring станет управлять обнаруженными классами, как будто они были зарегистрированы с помощью Spring напрямую. Вам не нужно снабжать компоненты какой-нибудь другой стандартной аннотацией типа `@Component` или использовать bean-компонент регистрации (пример П.6).

Конечно же, мы не рекомендуем создавать новый код в виде Servlet-экземпляров, поскольку тех же самых результатов можно добиться с применением Spring MVC, но если нужно заставить работать уже существующий код, то данный прием может помочь это сделать.

Пример П.6. Компонент @WebServlet, которым Spring будет управлять в наших интересах

```
package servlets;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

❶
@WebServlet(urlPatterns = "/servlets/hi", asyncSupported = false)
class GreetingsServlet extends HttpServlet {

 private final Log log = LogFactory.getLog(getClass());

 @Override
 protected void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException {
  this.log.info("doGet(" + req + ", " + resp + ")");
  resp.setStatus(200);
  resp.setHeader("Content-Type", "application/json");
  resp.getWriter().println("{ \"greeting\" : \"Hello, world\"}");
  resp.getWriter().close();
 }
}
```

❶ Основные необходимые вам указания можно разместить в файле `web.xml` или в регистрационном bean-компоненте в соответствующем типе аннотации Servlet API.

Создание REST API с помощью JAX-RS (Jersey)

В Spring Boot установка REST API с применением JAX-RS существенно упрощается. JAX-RS — стандарт, требующий реализации. В примере П.7 показано присущее Boot автоматическое конфигурирование для Jersey 2.x (https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jersey) в `GreetingEndpoint`. В данном примере используется Spring Boot-стартер `org.springframework.boot : spring-boot-starter-jersey`. Если нужно задействовать альтернативную реализацию JAX-RS, такую как REST Easy, то с учетом только что полученных знаний о регистрации фильтров и сервлетов установку этого средства в приложении Spring Boot можно выполнить без особых затруднений.

Пример П.7. JAX-RS GreetingEndpoint

```
package demo;

import javax.inject.Inject;
import javax.inject.Named;
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;

@Named
❶
@Path("/hello")
❷
@Produces({ MediaType.APPLICATION_JSON })
❸
public class GreetingEndpoint {

 @Inject
 private GreetingService greetingService;

 @POST
 ❹
 public void post(@QueryParam("name") String name) {  ❺
  this.greetingService.createGreeting(name);
 }

 @GET
 @Path("/{id}")
 public Greeting get(@PathParam("id") Long id) {
  return this.greetingService.find(id);
 }
}
```

❶ `@Inject`-аннотация спецификации JSR 330.

❷ `@Path`-аннотация спецификации JAX-RS, являющаяся полным функциональным эквивалентом `@RequestMapping` из Spring MVC. Она сообщает контейнеру о том, на каком пути должна быть показана эта конечная точка.

❸ `@RequestMapping`-аннотация из Spring MVC предоставляет атрибуты `produces` и `consumes`, позволяющие указать, содержимое каких типов может потребляться или создаваться заданной конечной точкой. В JAX-RS это отображение выполняется с помощью отдельных аннотаций.

❹ В иных случаях HTTP-операция также указывается в Spring MVC-аннотации `@RequestMapping`, но здесь она задана в отдельной аннотации.

❺ Аннотация `@QueryParam` предписывает JAX-RS выполнить внедрение любых параметров поступающих запросов (`?name==..`) в качестве аргументов метода. В Spring MVC для этого использовались бы аргументы метода, имеющие аннотацию `@RequestParam`.

Для включения основных функций и регистрации компонентов в Jersey требуется наличие подкласса `ResourceConfig` (пример П.8).

Пример П.8. Подкласс ResourceConfig, в котором выполняется конфигурирование Jersey

```
package demo;

import org.glassfish.jersey.jackson.JacksonFeature;
import org.glassfish.jersey.server.ResourceConfig;

import javax.inject.Named;

❶
@Named
public class JerseyConfig extends ResourceConfig {

 public JerseyConfig() {
  this.register(GreetingEndpoint.class); ❷
  this.register(JacksonFeature.class); ❸
 }
}
```

❶ Мы зарегистрировали конечную точку с помощью JSR 330, хотя здесь точно так же легко можно было бы воспользоваться любой стандартной аннотацией среды Spring. Соответствующие аннотации взаимозаменяемы.

❷ Наша конечная точка JAX-RS нуждается в явно выраженной регистрации.

❸ Нам нужно сообщить JAX-RS, что требуется управление JSON-маршализацией. В Java EE нет встроенного API для маршализации JSON (как уже ранее сообщалось, ориентировочно она будет доступна в Java EE 8), но можно воспользоваться характерными для Jersey реализациями *функций*, чтобы подключить популярные реализации JSON-маршализации, например Jackson.

В Spring Boot выполняется автоматическое конфигурирование Jersey-контейнера `org.glassfish.jersey.servlet.ServletContainer` для отслеживания всех запросов, относящихся к корневому пути приложения. В JAX-RS кодирование и декодирование сообщений производятся через интерфейсы SPI `javax.ws.rs.ext.MessageBodyWriter`

и `javax.ws.rs.ext.MessageBodyReader` соответственно, что несколько сродни иерархии `org.springframework.http.converter.HttpMessageConverter` из Spring MVC. Изначально в JAX-RS не содержится множество полезных, готовых к работе и доступных средств чтения и записи тела сообщения. В нашем примере для поддержки JSON-кодирования и декодирования в подклассе `ResourceConfig` регистрируется `JsonFeature`.

Управление транзакциями с помощью JTA и XA

Тема управления транзакциями у многих вызывает затруднения, в частности, из-за богатства выбора!

Выполнение транзакций, локальных по отношению к ресурсу, с помощью PlatformTransactionManager

По своей сути, транзакции функционируют в той или иной степени одинаково: клиент приступает к работе, выполняет определенный ее объем, отправляет работу, если что-то пошло не так, возвращает транзакцию в исходное состояние (выполняет ее откат) и восстанавливает состояние базового ресурса транзакций, приводя его к исходному, которое было до начала работы. Реализации в многочисленных ресурсах существенно различаются. JMS-клиенты создают объект `Session` с *поддержкой транзакций*, который затем отправляется. JDBC-объект `Connection` может быть сделан таким образом, чтобы не выполнять автоматическую отправку работы, что аналогично предписанию на накопление работы с ее последующей отправкой при поступлении соответствующей конкретной инструкции.

Выполнение транзакций, *локальных по отношению к ресурсу*, должно стать вашим исходным подходом к управлению транзакциями. Использоваться он будет с различными API транзакций, имеющимися в Java EE, такими как JMS, JPA, JMS, CCI и JDBC. В Spring поддерживается множество других ресурсов транзакций, подобных AMQP-брокерам, например RabbitMQ (http://www.rabbitmq.com/), графовая база данных Neo4j (https://neo4j.com/) и Pivotal Gemfire (https://pivotal.io/pivotal-gemfire). К сожалению, каждый из этих ресурсов транзакций предлагает для инициирования, отправки или отката работы различный API. Чтобы упростить ситуацию, в Spring предоставляется иерархия `PlatformTransactionManager`. Существует множество подключаемых реализаций `PlatformTransactionManager`, которые адаптируют разные представления о транзакциях к общему API. Spring позволяет управлять транзакциями с помощью реализаций этой иерархии.

В Spring обеспечивается многоуровневая поддержка транзакций. На самом нижнем уровне в Spring предоставляется шаблон `TransactionTemplate`. В нем заключается bean-компонент `PlatformTransactionManager`, используемый для

управления транзакцией применительно к блоку работы, который предоставлен клиентом в качестве реализации `TransactionCallback`. Сначала соберем всю схему. У нас есть нечто общее для следующих двух примеров: база данных `DataSource`, компоненты `DataSourceInitializer`, `JdbcTemplate`, `PlatformTransactionManager` и т. д., в связи с чем дадим им определения в общем классе конфигурации Java. В этой конфигурации Java также определяется объект `RowMapper`, который является интерфейсом обратного вызова `JdbcTemplate`, отображающим строки результатов на Java-объекты (пример П.9).

Пример П.9. Конфигурация Java для общих типов в сервисе JDBC, выполняющем транзакции

```
package basics;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.Resource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.SimpleDriverDataSource;
import org.springframework.jdbc.datasource.init.DataSourceInitializer;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;
import org.springframework.transaction.PlatformTransactionManager;

import javax.sql.DataSource;

@Configuration
public class TransactionalConfiguration {

❶
 @Bean
 DataSource dataSource() {
  SimpleDriverDataSource dataSource = new SimpleDriverDataSource();
  dataSource
   .setUrl("jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE");
  dataSource.setDriverClass(org.h2.Driver.class);
  dataSource.setUsername("sa");
  dataSource.setPassword("");
  return dataSource;
 }

 ❷
 @Bean
 DataSourceInitializer dataSourceInitializer(DataSource ds,
  @Value("classpath:/schema.sql") Resource schema,
  @Value("classpath:/data.sql") Resource data) {
  DataSourceInitializer init = new DataSourceInitializer();
  init.setDatabasePopulator(new ResourceDatabasePopulator(schema, data));
  init.setDataSource(ds);
```

```
  return init;
 }

 ❸
 @Bean
 JdbcTemplate jdbcTemplate(DataSource ds) {
  return new JdbcTemplate(ds);
 }

 ❹
 @Bean
 RowMapper<Customer> customerRowMapper() {
  return (rs, i) -> new Customer(rs.getLong("ID"), rs.getString("FIRST_NAME"),
   rs.getString("LAST_NAME"));
  }

 ❺
 @Bean
 PlatformTransactionManager transactionManager(DataSource ds) {
  return new DataSourceTransactionManager(ds);
 }
}
```

❶ Определение `DataSource`, общающегося только со встроенной в память базой данных H2.

❷ Определение `DataSourceInitializer`, запускающего схемы и инициализации данных DDL.

❸ Spring-шаблон `JdbcTemplate`, уменьшающий общие однострочные JDBC-вызовы.

❹ `RowMapper` — интерфейс обратного вызова, используемый `JdbcTemplate` для отображения SQL-объектов `ResultSet` в объекты, в данном случае относящиеся к типу `Customer`.

❺ Bean-компонент `DataSourceTransactionManager` адаптирует транзакции `DataSource` к Spring-иерархии `PlatformTransactionManager`.

Низкоуровневый Spring-шаблон `TransactionTemplate` может использоваться для явного проведения границ транзакции:

```
package basics.template;

import basics.Customer;
import basics.TransactionalConfiguration;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.*;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
```

```
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;

@Configuration
@ComponentScan
@Import(TransactionalConfiguration.class)
public class TransactionTemplateApplication {

 public static void main(String args[])
 {
  new AnnotationConfigApplicationContext(TransactionTemplateApplication.class);
 }

 ❶
 @Bean
 TransactionTemplate transactionTemplate(PlatformTransactionManager txManager)
 {
  return new TransactionTemplate(txManager);
 }
}

@Service
class CustomerService {

 private JdbcTemplate jdbcTemplate;

 private TransactionTemplate txTemplate;

 private RowMapper<Customer> customerRowMapper;

 @Autowired
 public CustomerService(TransactionTemplate txTemplate,
  JdbcTemplate jdbcTemplate, RowMapper<Customer> customerRowMapper) {
  this.txTemplate = txTemplate;
  this.jdbcTemplate = jdbcTemplate;
  this.customerRowMapper = customerRowMapper;
 }

 public Customer enableCustomer(Long id) {

  ❷
  TransactionCallback<Customer> customerTransactionCallback = (
   TransactionStatus transactionStatus) -> {

   String updateQuery = "update CUSTOMER set ENABLED = ? WHERE ID = ?";
   jdbcTemplate.update(updateQuery, Boolean.TRUE, id);

   String selectQuery = "select * from CUSTOMER where ID = ?";
```

```
    return jdbcTemplate.queryForObject(selectQuery, customerRowMapper, id);
  };

  ❸
  Customer customer = txTemplate.execute(customerTransactionCallback);

  LogFactory.getLog(getClass())
    .info("retrieved customer # " + customer.getId());

  return customer;
 }

}
```

❶ Шаблону `TransactionTemplate` требуется только `PlatformTransactionManager`.

❷ `TransactionCallback` определена с помощью лямбда-выражения Java 8. Тело этого метода будет запущено в подтвержденной транзакции и отправлено и закрыто по завершении.

❸ Возвращаемое значение может быть чем угодно, но думаем, что это сама по себе интересная информация: в конечном итоге транзакции должны быть направлены на получение одного подходящего побочного продукта.

Шаблон `TransactionTemplate` должен быть знаком, если вам когда-либо приходилось пользоваться любой из имеющихся в Spring других `*Template`-реализаций. Он определяет границы транзакции явным образом. Это пригодится в том случае, когда вам нужно огородить логические секции в транзакции и контролировать ее начало и остановку.

Чтобы выполнить декларативную демаркацию транзакционных границ, задействуйте аннотации. В Spring уже давно поддерживаются декларативные правила транзакций, как внешние для бизнес-компонентов, к которым применяются правила, так и внутренние. Наиболее популярный способ определить правила транзакций, начиная с Spring 1.2, выпущенной в мае 2005 года сразу после Java SE 5, — использовать Java-аннотации.

В Spring поддержка декларативного управления транзакциями выполняется с помощью аннотации `@Transactional` (https://docs.spring.io/spring-framework/docs/current/javadoc-api//org/springframework/transaction/annotation/Transactional.html). Ею могут снабжаться типы или отдельные методы. Аннотация может использоваться для указания качеств транзакций, например таких, как распространение, исключения, относящиеся к откатам, и т. д.

И какое отношение все это имеет к Java EE? Версия Java EE 5, выпущенная в 2006 году, включала компонент EJB 3, который определял способ демаркации границ транзакций на основе его аннотации `javax.ejb.TransactionAttribute`. В Spring данная аннотация тоже действует, если обнаруживается в bean-компонентах

Spring. Элемент `TransactionAttribute` подходит для бизнес-компонентов на основе EJB, но вне таких компонентов действует ограниченно. В JTA 1.2 имеется определение `javax.transaction.Transactional`, аннотации границ транзакции общего назначения, которая более или менее похожа на Spring-аннотацию `@Transactional` восьмилетней давности. В Spring данная аннотация также действует, если она представлена в этой среде.

Конфигурация, приводящая все это в рабочее состояние, в основном является такой же, как и при использовании `TransactionTemplate`, за исключением того, что границы транзакций на основе аннотаций включаются с помощью аннотации `@EnableTransactionManagement` и больше не нуждаются в bean-компоненте `TransactionTemplate`. В случае применения Spring Boot не следует добавлять `@EnableTransactionManagement`; все будет работать, если сконфигурировать bean-компонент `PlatformTransactionManager` (пример П.10).

Пример П.10. Конфигурация Java для простого приложения на основе использования аннотации @Transactional

```
package basics.annotation;

import basics.Customer;
import basics.TransactionalConfiguration;
import basics.template.TransactionTemplateApplication;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.
AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@ComponentScan
@Import(TransactionalConfiguration.class)
@EnableTransactionManagement
(1)
public class TransactionalApplication {

 public static void main(String args[]) {
  new AnnotationConfigApplicationContext(TransactionTemplateApplication.class);
 }
}

@Service
class CustomerService {
```

```
 private JdbcTemplate jdbcTemplate;

 private RowMapper<Customer> customerRowMapper;

 @Autowired
 public CustomerService(JdbcTemplate jdbcTemplate,
  RowMapper<Customer> customerRowMapper) {
  this.jdbcTemplate = jdbcTemplate;
  this.customerRowMapper = customerRowMapper;
 }

 // @org.springframework.transaction.annotation.Transactional
 ❷
 // @javax.ejb.TransactionAttribute  ❸
 @javax.transaction.Transactional
 ❹
 public Customer enableCustomer(Long id) {

  String updateQuery = "update CUSTOMER set ENABLED = ? WHERE ID = ?";
  jdbcTemplate.update(updateQuery, Boolean.TRUE, id);

  String selectQuery = "select * from CUSTOMER where ID = ?";
  Customer customer = jdbcTemplate.queryForObject(selectQuery,
   customerRowMapper, id);
  LogFactory.getLog(getClass())
   .info("retrieved customer # " + customer.getId());
  return customer;
 }
}
```

❶ `@EnableTransactionManagement` включает обработку транзакции. Для этого требуется наличие где-либо подходящего компонента `PlatformTransactionManager`.

❷ Можно воспользоваться Spring-аннотацией `@org.springframework.transaction.annotation.Transactional`...

❸ ...или EJB-аннотацией `@javax.ejb.TransactionAttribute`...

❹ ...или аннотацией JTA 1.2 `@javax.transaction.Transactional`.

Предпочтение следует отдавать варианту Spring-аннотации `@Transactional`. Она эффективнее EJB3-альтернативы, поскольку позволяет предоставлять транзакционные семантики (такие операции, как приостановка и возобновление транзакции), чего не может сделать относящаяся к EJB аннотация `@TransactionAttribute` (и разумеется, сама технология EJB), и не требует дополнительной зависимости от JTA или EJB. Но приятно осознавать, что это в любом случае будет работать.

В этих примерах требуются только драйвер `DataSource` и строка кода `org.springframework.boot : spring-boot-starter-jdbc`.

Глобальные транзакции, выполняемые с помощью Java Transaction API (JTA)

Среда Spring упрощает работу с одним транзакционным ресурсом. А что делать проектировщику, если предпринимается попытка манипулировать через транзакции более чем одним транзакционным ресурсом, скажем глобальной транзакцией? Например, как разработчик с помощью транзакций должен вести запись в базу данных и подтверждать получение JMS-сообщения? Глобальные транзакции являются альтернативой тем транзакциям, которые по отношению к ресурсам носят локальный характер; в них вовлекаются несколько ресурсов за одну транзакцию. Для обеспечения целостности глобальной транзакции координатор должен быть агентом, независимым от ресурсов, принимающих участие в транзакции. Он должен гарантировать возможность повторного воспроизведения транзакции, потерпевшей неудачу, и проявить собственную невосприимчивость к отказам. JTA неизбежно усложняет процесс и добавляет к нему состояние, чего избегает транзакция, локальная по отношению к ресурсу. Большинство диспетчеров глобальных транзакций ведут общение по протоколу X/Open XA (https://en.wikipedia.org/wiki/X/Open_XA), позволяющему транзакционным ресурсам (таким как база данных или брокер сообщений) привлекаться к глобальным транзакциям и участвовать в них. Промежуточное программное средство Java EE для протокола X/Open называется JTA (https://en.wikipedia.org/wiki/Java_Transaction_API).

Для решения наших задач важно знать, что JTA предоставляет два ключевых интерфейса: `javax.transaction.UserTransaction` (https://docs.oracle.com/javaee/7/api/javax/transaction/UserTransaction.html) и `javax.transaction.TransactionManager` (https://docs.oracle.com/javaee/7/api/javax/transaction/TransactionManager.html).

Объект `UserTransaction` поддерживает вполне предполагаемые методы: `begin()`, `commit()`, `rollback()` и т. д. Клиенты используют его для начала глобальной транзакции. Как только глобальная транзакция откроется, к JTA-транзакции *могут быть привлечены* такие поддерживающие JTA-ресурсы, как JDBC-ресурс `javax.sql.XADataSource` или JMS-ресурс`javax.jms.XAConnectionFactory`. Они ведут обмен данными с координатором глобальной транзакции и следуют протоколу, чтобы либо отправить работу в атомарном режиме во все привлеченные ресурсы, либо выполнить откат. Каждый атомарный ресурс может поддерживать, а может и не поддерживать JTA-протокол, который сам ведет обмен данными по протоколу XA.

Объекту `UserTransaction` в контейнере Java EE требуется присутствие в JNDI-контексте под привязкой `java:comp/UserTransaction`, поэтому определить его местонахождение Spring-диспетчеру `JtaTransactionManager` не составит труда. Этого простого интерфейса вполне хватает для обработки основных операций по управлению транзакциями. Примечательно, что он недостаточно развит для обработки подчиненных транзакций, приостановки и возобновления транзакций или для

выполнения других функций, обычных для современных реализаций JTA. Вместо него лучше воспользоваться диспетчером `TransactionManager`, если таковой будет доступен. Серверам приложений Java EE не нужно предоставлять этот интерфейс, и они весьма редко его предоставляют под одной и той же JNDI-привязкой. В среде Spring известен широко распространенный контекст многих популярных серверов приложений Java EE, и она будет стараться автоматически определить для вас их местонахождение.

Зачастую bean-компонент, реализующий javax.transaction.UserTransaction, реализует и javax.transaction.TransactionManager!

JTA также можно запускать за пределами контейнера Java EE. Существует множество популярных реализаций JTA сторонних разработчиков (в том числе и с открытым кодом), среди которых Atomikos и Bitronix. В Spring Boot предоставляется автоматическое конфигурирование обеих реализаций (https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle//#boot-features-jta). Atomikos (https://www.atomikos.com/Main/WebHome) имеет коммерческую поддержку и обеспечивает базовую редакцию с открытым кодом. Рассмотрим пример в сервисе приветствий `GreetingService`, где в качестве элемента глобальной транзакции JMS используется для отправки уведомлений, а JPA — для сохранения записей в СУРБД (пример П.11).

Пример П.11. Учебный сервис, работающий как с JDBC, так и с ресурсом JMS посредством JTA

```
package demo;

import org.springframework.jms.core.JmsTemplate;

import javax.inject.Inject;
import javax.inject.Named;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.Collection;

@Named
@javax.transaction.Transactional
❶
public class GreetingService {

 @Inject
 private JmsTemplate jmsTemplate;  ❷

 @PersistenceContext
 ❸
 private EntityManager entityManager;

 public void createGreeting(String name, boolean fail) {  ❹
```

```
  Greeting greeting = new Greeting(name);
  this.entityManager.persist(greeting);
  this.jmsTemplate.convertAndSend("greetings", greeting);
  if (fail) {
   throw new RuntimeException("simulated error");
  }
 }

 public void createGreeting(String name) {
  this.createGreeting(name, false);
 }

 public Collection<Greeting> findAll() {
  return this.entityManager.createQuery(
   "select g from " + Greeting.class.getName() + " g", Greeting.class)
   .getResultList();
 }

 public Greeting find(Long id) {
  return this.entityManager.find(Greeting.class, id);
 }
}
```

1. Среде Spring сообщается, что все открытые методы компонента являются транзакционными.
2. В этом коде для работы с JMS-ресурсом используется Spring JMS-клиент `JmsTemplate`.
3. В данном коде применяется поддержка `spring-boot-starter-data-jpa`, позволяющая внедрить диспетчер JPA `EntityManager` с JPA-аннотацией `@PersistenceContext` в соответствии с обычными соглашениями Java EE.
4. Этот метод получает булево значение, и если оно истинно (`true`), то выдается исключение. Данное исключение инициирует откат JTA-транзакции. После запуска кода станут видны доказательства того, что только два из трех тел работы будут завершены.

Мы продемонстрируем это в `GreetingServiceClient`, создав три транзакции и имитировав откат на третьей из них. На выводе консоли будет показано наличие двух записей, возвратившихся из источника данных JDBC `javax.sql.DataSource`, и двух записей, полученных из встроенного JMS-получателя `javax.jms.Destination` (пример П.12).

Пример П.12. Пошаговое прохождение нашего транзакционного сервиса

```
package demo;

import javax.annotation.PostConstruct;
import javax.inject.Inject;
import javax.inject.Named;
```

```
import java.util.logging.Logger;

@Named
public class GreetingServiceClient {

 @Inject
 private GreetingService greetingService;

 ❶
 @PostConstruct
 public void afterPropertiesSet() throws Exception {
  greetingService.createGreeting("Phil");
  greetingService.createGreeting("Dave");
  try {
   greetingService.createGreeting("Josh", true);
  }
  catch (RuntimeException re) {
   Logger.getLogger(Application.class.getName()).info("caught exception...");
  }
  greetingService.findAll().forEach(System.out::println);
 }
}
```

❶ До сих пор аннотация `@PostConstruct` нам еще не попадалась. Она является частью JSR 250 и семантически аналогична Spring-интерфейсу `InitializingBean`. Эта аннотация определяет обратный вызов, инициирование которого произойдет *после* bean-зависимостей, будь они аргументами конструктора, JavaBean-свойствами или полями.

Spring Boot выполнит автоматическую установку JPA на основе сконфигурированного источника данных DataSource. В этом примере используется встроенная в Spring Boot поддержка DataSource (https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle//#howto-configure-a-datasource). Если встроенная база данных (подобная H2 (http://www.h2database.com/html/main.html), которую мы здесь и применяем, или Derby и HSQL) присутствует в путях к классам и имеется явное определение nojavax.sql.DataSource, то среда Spring Boot создаст рабочий bean-компонент DataSource.

Вдобавок ко всему среда Spring Boot существенно упрощает установку JMS-подключения. Точно так же, как и со встроенным `DataSource`, Spring Boot может создать встроенную JMS-фабрику `ConnectionFactory` (https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle//#boot-features-hornetq), воспользовавшись имеющимся у Red Hat поставщиком сообщений `HornetQ` (http://hornetq.jboss.org/) во встроенном режиме при условии, что нужные типы прописаны в путях к классам (`org.springframework.boot : spring-boot-starter-hornetq`) и указан ряд свойств (пример П.13).

Пример П.13. Конфигурация, предназначенная для настройки встроенной JMS-фабрики ConnectionFactory

```
spring.hornetq.mode=embedded
spring.hornetq.embedded.enabled=true
spring.hornetq.embedded.queues=greetings
```

Для активации нужного автоматического конфигурирования добавьте требуемые Spring Boot-стартеры (`org.springframework.boot : spring-boot-starter-hornetq`) и поддержку `HornetQ` (`org.hornetq : hornetq-jms-server`). При необходимости подключиться к традиционным, невстроенным экземплярам можно просто либо указать свойства, такие как `spring.datasource.url` и `spring.hornetq.host`, либо предоставить `@Bean`-определения подходящих типов.

В данном примере для настройки ресурса Atomikos, а также поддерживающих XA-протокол JMS- и JDBC-ресурсов используется стартер Spring Boot Atomikos (`org.springframework.boot : spring-boot-starter-atomikos`).

В сегодняшнем распределенном мире диспетчеры глобальных транзакций должны рассматриваться с точки зрения архитектуры (http://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html). Распределенные транзакции регулируют возможности одного сервиса по обработке транзакций в независимом ритме. В распределенных транзакциях подразумевается поддержка состояния в нескольких сервисах, притом что в идеале она должна быть в одном сервисе. Желательно, чтобы сервисы могли совместно пользоваться состоянием на уровне не базы данных, а API, но в таком случае весь рассматриваемый вопрос приобретает чисто теоретический характер, поскольку REST API вообще не поддерживают протокол X/Open! Для синхронизации состояния имеются и другие шаблоны, способствующие горизонтальной масштабируемости и временной развязке, сфокусированные на рассылке сообщений. Более подробно вопрос рассылки сообщений рассматривается в главах 10 и 12, посвященных проблемам работы с сообщениями и интеграции.

Развертывание в среде Java EE

В примерах данного приложения вместо Apache Tomcat (используемого в Spring Boot по умолчанию) применялся замечательный Servlet-механизм Undertow (http://undertow.io/), встроенный в сервер приложений Red Hat Wildfly. При желании можно также обратиться к Jetty. Чтобы задействовать любой альтернативный вариант, следует явным образом определить зависимость для `org.springframework.boot : spring-boot-starter-tomcat`, а затем установить для его параметра `scope` значение `provided`. Тем самым будет гарантировано *отсутствие* Tomcat в путях к классам, даже если одна или несколько зависимостей приносят его в силу переходных процессов. Затем нужно добавить зависимость для `spring-boot-starter-jetty` или `spring-boot-starter-undertow`. В частности, задумка с применением Undertow возникла по запросу сторонних производителей, но авторам это

средство действительно понравилось из-за быстроты запуска. Но вы можете увидеть свою выгоду совершенно в другом.

Хотя в наших примерах использовано множество довольно знакомых API Java EE, они все же являются обычными приложениями Spring Boot, поэтому по определению их можно запускать командами вида `java -jar your.jar` или без труда развертывать в ориентированных на процессы платформах в виде сервисов (https://en.wikipedia.org/wiki/Platform_as_a_service), подобных Heroku или Cloud Foundry (https://www.cloudfoundry.org/).

Если требуется развернуть их в автономном сервере приложений (подобном Apache Tomcat, или Websphere, или в каком-нибудь из этой же серии), то нужно просто конвертировать сборку в файл с расширением `.war` и развернуть его соответственно в любом контейнере Servlet 3. В большинстве случаев все сводится к замене пакетирования Maven-сборки на файл с расширением `.war`, добавлению инициализатора сервлета и присваиванию статуса `provided`. Или же к применению какого-либо иного способа исключения зависимостей Spring Boot (таких как `spring-boot-starter-tomcat` или `spring-boot-starter-jetty`), предоставляющих сервлет-контейнер, из тех, что собирается предоставить контейнер Java EE. При использовании Spring Initializr (http://start.spring.io/) эти действия можно пропустить, выбрав пункт `war` из раскрывающегося списка Packaging (Пакетирование).

При конвертации существующего приложения Spring Boot на основе fat-jar придется добавить Servlet-инициализатор. Это программный эквивалент `web.xml`. Spring Boot предоставляет базовый класс `org.springframework.boot.context.web.SpringBootServletInitializer`, который установит Spring Boot и соответствующие механизмы в стандартной манере Servlet 3. Следует заметить, что если приложение будет развернуто подобным образом, то все встроенные свойства веб-контейнера, такие как SSL-конфигурации, HTTP-сжатие ресурсов и управление портами, работать не будут.

```
package demo;

import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.support.SpringBootServletInitializer;

public class GreetingServletInitializer extends SpringBootServletInitializer {

 @Override
 protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
  return builder.sources(Application.class);
 }
}
```

Если же развернуть приложение в более классическом сервере приложений, то Spring Boot сможет получить преимущества от использования возможностей сервера приложений. Можно напрямую задействовать привязанную к JNDI JMS-фабрику `ConnectionFactory`, `DataSource` от JDBC или `UserTransaction` от JTA.

Резюме

Многие из этих API будут подвергаться сомнениям. Вам действительно нужно работать с распределенными транзакциями из множества источников? В JPA имеется вполне подходящий API для общения с источником данных `javax.sql.DataSource` на основе SQL, но в хранилища Spring Data, конечно же, включена поддержка JPA и не только. В них существенно упрощен прямой JPA-интерфейс. В них также добавлена поддержка Cassandra, MongoDB, Redis, Riak, Couchbase, Neo4j и постоянно расширяющийся список альтернативных технологий. Более того, даже если вы ставите перед собой цель использовать ORM с SQL и СУРБД, то в вашем распоряжении по-прежнему имеется богатый выбор. Посмотрите, к примеру, на соответствующие варианты для MyBatis (https://spring.io/blog/2017/02/22/spring-tips-mybatis) и JOOQ (https://www.jooq.org/), представляющие собой функции как варианты в Spring Initializr (http://start.spring.io/). Надеемся, что API низкого уровня являются временными мерами на пути к архитектуре микросервисов, а не самоцелью.

В контексте микросервисов Java EE противоречит требованиям узкой специализации в архитектуре современных микросервисов, которые по своей сути являются небольшими сервисами, нацеленными на решение одной конкретной задачи и имеющими наименьшее количество движущихся частей. Отдельные современные серверы приложений могут быть весьма небольшими, но принцип тот же: зачем платить за то, что не требуется? Зачем заходить так далеко в стремлении отделить ваши сервисы друг от друга лишь для того, чтобы удовлетвориться ненужной связанностью инфраструктуры, запускающей код?

Серверы приложений создавались для другой эпохи (в конце 1990-х годов), когда оперативная память ценилась на вес золота. Сервер приложений обещал изоляцию приложения, а также возможность отслеживания и получения объединенной инфраструктуры, но сегодня нетрудно заметить, что ни с одной из этих задач данная технология достойно справиться так и не смогла. Сервер приложений не защищает приложения от конкуренции с другими приложениями за пределами загрузчика классов (но даже там не все проходит гладко, отсюда возникают потребности в модулях (Project JigSaw в Java 9) и технологиях типа OSGi). Не получается изолировать центральный процессор, оперативную память, файловые системы и даже сами JVM, один компонент может создать дефицит ресурсов для других компонентов. Вместо этого пусть необходимую изолированность обеспечит операционная система, которая отвечает за процессы, знает об их существовании и не занимается серверами приложений и загрузчиками классов.

Довольно часто встречаются приложения с конфигурацией сервера приложений (или даже сами серверы приложений!), зарегистрированные в системах управления исходным кодом наряду с приложением. Этот весьма сомнительный подход подразумевает, что кто-то все же пытается выжать из оперативной памяти своих машин как можно больше отдачи путем совместного размещения приложений на одном и том же сервере приложений. Делается это для сборки и выдачи единого

образа со всеми конфигурациями сервера приложений, настроенного вместе с приложением. Зачем это искусственное разделение? Разве нет никакой вероятности развертывания приложения в каком-нибудь другом сервере приложений, зачем рассматривать два компонента (приложение и сервер) как две физически разнесенные вещи?

Многие разработчики используют сервер приложения (или веб-серверы наподобие Apache Tomcat, продвигаемого в основном компанией Pivotal) по причине того, что он берет на себя постоянную операционную нагрузку. Операции знают, как выполнять развертывание, масштабирование и управление сервером приложений. Все понятно. Но сегодня наиболее подходящим является контейнер, выполненный по технологии Docker или Warden, поддерживающей Cloud Foundry. Контейнеры предоставляют согласованную область управления для рабочих операций. Все, что запускается внутри, совершенно скрыто.

Контейнеры, как и облака, озабочены только процессами. В них даже не выстраиваются предположения о том, что ваше приложение предоставит конечную точку HTTP. Более того, для этого зачастую нет никаких причин. Несмотря на присутствие в микросервисах HTTP и REST, в них нет ничего, что бы подразумевало данное присутствие. Благодаря отходу от сервера приложений и переходу к развертываниям на основе использования `.jar`-файлов появилась возможность применять HTTP (и сервлет-контейнер) исключительно в тех случаях, где это целесообразно. Мы склонны рассматривать данный вопрос с такой точки зрения: что, если сегодняшним протоколом был не HTTP, а FTP? Что, если весь мир обращался бы к сервисам, предоставляемым с помощью FTP-сервера? Имеются ли какие-либо непреодолимые архитектурные проблемы для развертывания наших приложений на FTP-сервере? Нет? Тогда почему мы развертываем их *на серверах* HTTP, EJB, JMS, JNDI и XA/Open?

Изучая данное приложение, вы убедились: Spring вполне успешно взаимодействует с сервисами и API Java EE и эти интерфейсы могут пригодиться даже в мире микросервисов. Мы надеемся, из всего изложенного вы сделаете вывод, что сам сервер приложений Java EE уже не нужен. Он лишь замедлит вашу работу.

Джош Лонг, Кеннет Бастани

Java в облаке. Spring Boot, Spring Cloud, Cloud Foundry

Перевел с английского *Н. Вильчинский*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*С. Заматевская*
Корректоры	*Е. Павлович, Т. Радецкая*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 07.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 17.07.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 50,310. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*